CS-1201 Object Oriented Programming

Relationship Between Classes

Arbish Akram

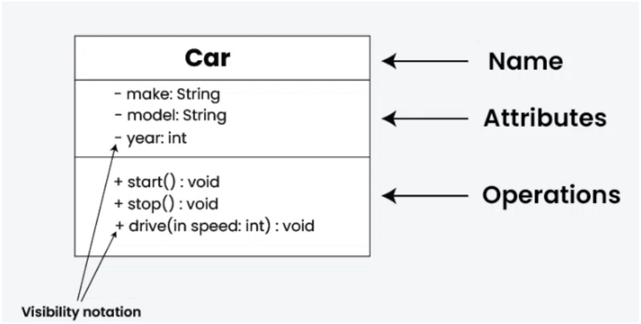
Department of Computer Science
Government College University

Class Diagrams

- Class diagrams are a type of UML (Unified Modeling Language) diagram.
- They represent the structure and relationships of classes within a system.
- Used to construct and visualize object-oriented systems.
- Components:
    - Classes (depicted as boxes)
    - Attributes
    - Methods
    - Relationships (associations, etc.)
- Provide a high-level overview of a system's design.

Class

- A blueprint or template for creating objects.
- Components of a Class:
  - Attributes: Characteristics or properties (data members).
  - Methods: Behaviors or actions (functions/procedures).
- Objects: Instances of classes.



UML Class Notation

- Class Name: Centered and bold in the top compartment of the class box.
- Attributes:
  - Listed in the second compartment.
  - Includes visibility (+, −, #, ~) and data type.
- Methods:
  - Listed in the third compartment.
  - Includes visibility, return type, and parameters.
- Visibility Notation:
  - + for public
  - − for private
  - # for protected
  - ~ for package

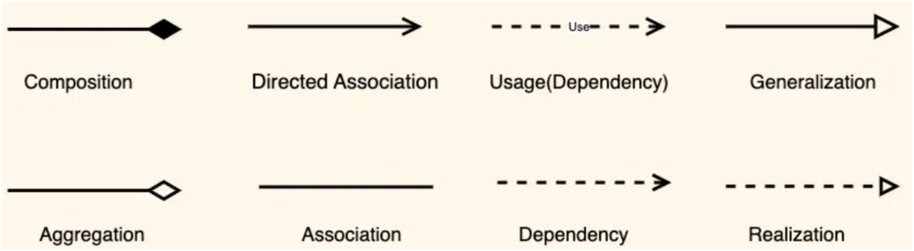
Relationships Between Classes

- **Association**: Bi-directional relationship.
  - *Example*: A `Teacher` and `Student` can interact with each other. A `Teacher` may know their `Students` and vice versa.
- **Directed Association**: One-way relationship.
  - *Example*: A `Customer` places an `Order`. The `Customer` is aware of the `Order`, but the `Order` does not need to know about the `Customer`.
- **Aggregation**: Whole-part relationship.
  - *Example*: A `Library` and `Books`. `Books` can exist without the `Library`.
- **Composition**: Stronger whole-part relationship.
  - *Example*: A `House` and its `Rooms`. If the `House` is destroyed, the `Rooms` are also destroyed.
- **Generalization (Inheritance)**: "Is-a" relationship.
  - *Example*: A `Dog` is an `Animal`.

Relationships Between Classes

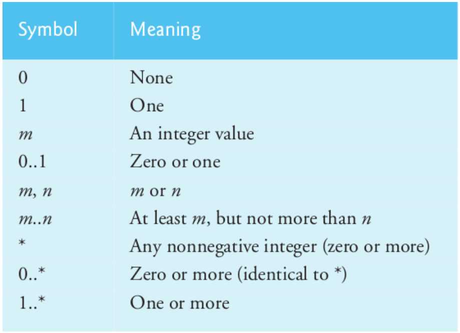
- **Realization (Interface Implementation)**:
  - Class implements interface.
  - *Example*: A `List` class implements an `Iterable` interface.
- **Dependency Relationship**:
  - Loosely coupled.
  - *Example*: A `Car` class may depend on a `Driver` class temporarily for a drive method.
- **Usage (Dependency) Relationship**:
  - Utilizes another class.
  - *Example*: A `Printer` uses a `Document` object to print.

Relationships Between Classes



Multiplicity in Association

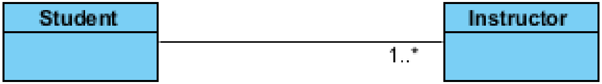
- **Multiplicity** defines the number of instances of one class that can be associated with a single instance of another class.
- It is used in associations to specify how many objects of one class relate to objects of another class.



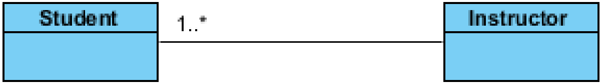
Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Multiplicity in Association

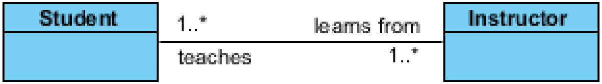
- A single student can associate with multiple teachers.



- This indicates that every instructor has one or more students.



- We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



Association

- Association represents a relationship between two classes where one class uses or interacts with another class.
- Each class can exist independently.

Association

```cpp
class Course {
public:
    string courseName;
    // Constructor to initialize the course name
    Course(string name) : courseName(name) {}
};
class Teacher {
public:
    string teacherName;
    Teacher(string name) : teacherName(name) {}

    void teachCourse(Course course) {
        cout << teacherName << " is teaching " << course.courseName << endl;
    }
};
int main() {
Course course("Mathematics");
Teacher teacher("Mr. Smith");
 // Teacher is associated with the Course through the teachCourse method
teacher.teachCourse(course);  // Association: Teacher uses course
return 0;
}
```

Composition

- Represents a ”whole-part” relationship where the part cannot exist without the whole.
- If the whole is destroyed, the parts are also destroyed.

Composition

```cpp
class Room {
    public:
        string roomName;  // Name of the room
        Room(string name) : roomName(name) {}
        void describeRoom() const {
            cout << "This is the " << roomName << endl;
        }
};
class House {
    private:
        Room livingRoom;    // Composition: House has a living room
        Room bedroom;       // Composition: House has a bedroom
    public:
        string houseName;  // Name of the house
        House(string hName, string livingRoomName, string bedroomName)
        : houseName(hName), livingRoom(livingRoomName), bedroom(bedroomName) {}

        void describeHouse() const {
            cout << "House: " << houseName << endl;
            livingRoom.describeRoom();
            bedroom.describeRoom();
        }
};
```

Composition

```cpp
int main() {
    // Create a House object with specific room names
    House myHouse("My Sweet Home", "Living Room", "Master Bedroom");
    // Describe the house and its rooms
    myHouse.describeHouse();
    return 0;
}

```