Composition

What if a relationship is so strong that the "part" is meaningless and cannot even exist without the "whole"?

This is the world of Composition.

Composition is the strongest form of association in object-oriented design. It models a “has-a” relationship between a whole and its parts—but unlike aggregation, composition comes with ownership and lifecycle dependency.

When you use composition, you're saying:

“This object is composed of other objects.”
“And if the container goes away, so do its parts.”
1. What is Composition?
Composition is a special type of association that signifies strong ownership between objects. The “whole” class is fully responsible for creating, managing, and destroying the “part” objects. In fact, the parts cannot exist without the whole.

Key Characteristics of Composition:
Represents a strong “has-a” relationship.
The whole owns the part and controls its lifecycle.
When the whole is destroyed, the parts are also destroyed.
The parts are not shared with any other object.
The part has no independent meaning or identity outside the whole.
If the part makes no sense without the whole, use composition.

Real-World Analogy
Imagine a House and its Rooms:

A house has a living room, a kitchen, a bedroom.
These rooms do not exist on their own. They are part of the house.
When the house is demolished, the rooms are gone with it.
You don’t transfer a bedroom from one house to another.
This is a textbook example of composition. The rooms are tightly bound to the house—not just logically, but in lifecycle and ownership as well.

2. Code Example
Let’s model above example using code.

Room class

Java

```
class Room {
    private String name;

    public Room(String name) {
        this.name = name;
    }

    public void describe() {
        System.out.println("This is the " + name);
    }
}
```

Python

```
class Room:
    def __init__(self, name):
        self.name = name

    def describe(self):
        print(f"This is the {self.name}")
```

House class

Java

```
class House {
    private List<Room> rooms;

    public House() {
        rooms = new ArrayList<>();
        rooms.add(new Room("Living Room"));
        rooms.add(new Room("Kitchen"));
        rooms.add(new Room("Bedroom"));
    }

    public void showHouseLayout() {
        for (Room room : rooms) {
            room.describe();
        }
    }
```

}

Python

```
class House:
    def __init__(self):
        self.rooms = []
        self.rooms.append(Room("Living Room"))
        self.rooms.append(Room("Kitchen"))
        self.rooms.append(Room("Bedroom"))

    def show_house_layout(self):
        for room in self.rooms:
            room.describe()
```

Explanation
The House creates, manages, and owns its Room objects.
The Room objects do not exist independently outside the context of the House.
No external class should reuse or manage these Room instances.
If the House is deleted (e.g., garbage collected), the Rooms are destroyed too.
This demonstrates a true composition relationship—where object ownership and lifecycle are tightly coupled.

3. UML Representation

In UML class diagrams, **composition** is represented by a **filled diamond (◆)** at the "whole" end of the relationship.

`House ◆———→ Room`

This means:

- A `House` **owns** multiple `Room`s.
- The `Room`s **do not exist independently**.
- When the `House` dies, the `Room`s die with it.

4. When to Use Composition

Use composition when:

- The **part is not meaningful** without the whole.
- The **whole should control** the lifecycle of its parts.
- The parts are **not reused elsewhere** in the system.
- You want to model a **strong containment** relationship.

Composition is a **preferred alternative to inheritance** when building flexible systems.

"**Favor composition over inheritance**." — GoF Design Principle

Why?

- You can build complex behavior by **composing smaller, reusable parts**.
- It avoids the **tight coupling** and **fragility** of inheritance hierarchies.
- You can **swap out parts dynamically** to modify behavior.

For example:

- A `Vehicle` can **compose** an `Engine` interface.
- Swap between `PetrolEngine`, `ElectricEngine`, or `HybridEngine` at runtime.

This leads to **cleaner, testable, and decoupled code**.

5. Composition vs Aggregation vs Association

```
| Feature        | Association       | Aggregation
| Composition                          |
| -------------- | ----------------- | --------------------------
| ---------------------------------- |
| Ownership      | ❌ None            | ❌ Weak (shared reference)
| ✅ Strong (owns the part)            |
| Lifecycle tie  | ❌ Independent     | ❌ Independent
| ✅ Dependent — part dies with whole  |
| Multiplicity   | Flexible          | Whole can group many parts
| Whole composed of parts              |
| Reusability    | High              | Moderate
| Low — parts not reused               |
| Example        | Student ↔ Course  | Department → Professor
| House → Room
```