

# Week Four: Advanced Design Principles

Date: November 6th





# Composition: The Fifth Pillar of OOP

Beyond encapsulation, inheritance, polymorphism, and abstraction, **composition** is often considered a fifth pillar of object-oriented programming.

## Definition:

Composition means building complex objects by **combining simpler ones**. One class contains other classes as part of its state, modeling a “**has-a**” relationship (e.g., a Car *has an* Engine, an Animal *has traits*).

## Benefits:

-  Promotes flexibility and avoids rigid hierarchies
-  Encourages modularity, reuse, and adaptability
-  Preferred over inheritance for long-term scalability
-  Behaviors can be combined like interchangeable building blocks

## Interpretation: Why Composition Matters

Inheritance can be useful, but it often locks systems into rigid structures.

Composition provides flexibility by allowing behaviors to be combined or replaced without disrupting the overall design.

This makes systems easier to adapt when requirements change.

# Composition vs Inheritance

Aspect	Inheritance ( <i>is-a</i> )	Composition ( <i>has-a</i> )
Definition	Subclass derives from parent, reusing properties/methods	Class built by including other objects as fields
Relationship	Hierarchical — “Dog is an Animal”	Structural — “Car has an Engine”
Flexibility	Rigid; parent changes ripple through subclasses	Flexible; parts can be swapped or modified
Coupling	Tight coupling between parent and child	Loose coupling; modules interact via interfaces

## Composition vs Inheritance (cont.)

Aspect	Inheritance ( <i>is-a</i> )	Composition ( <i>has-a</i> )
Reuse	Code reuse via inheritance	Code reuse via delegation to contained objects
Risks	Fragile hierarchies, explosion of subclasses	More maintainable, avoids subclass explosion
Best Use	Clear hierarchical “is-a” relationships	Combining behaviors, avoiding duplication

## Interpretation: Why Favor Composition

Inheritance is appropriate for natural hierarchies, but it can make systems brittle. Composition is more adaptable because modules interact through well-defined boundaries.

This approach reduces duplication and supports long-term maintainability.

## Duplication and Coupling

- Duplication is bad → fix the **design**, not just the code
- **Coupling** → when two modules are tightly bound together, changes in one force changes in the other
- **Solid line = coupling** → direct dependency between modules
- **Dotted line = dependency injection** → reduces circular dependencies and improves adaptability

# Coupling Explained

Coupling is the **degree of dependency between modules or classes**.

- **Tight coupling** → modules rely heavily on each other's internal details
- **Loose coupling** → modules interact only through well-defined interfaces

## Examples of Coupling:

- A UI class directly accessing database queries (tight coupling)
- A payment service hard-coding calls to a specific bank API (tight coupling)
- A logging module used via an interface (loose coupling)
- Dependency injection frameworks (loose coupling)

## Interpretation: Coupling

Tightly coupled modules are fragile because a small change can break multiple parts of the system.

Loose coupling isolates complexity, allowing modules to evolve independently.

Interfaces and abstractions are the key to achieving loose coupling.

## SOLID Principles

- **S: Single Responsibility** → one reason to change
- **O: Open/Closed** → open for extension, closed for modification
- **L: Liskov Substitution** → subclasses must not alter expected behavior
- **I: Interface Segregation** → no client should depend on unused methods
- **D: Dependency Inversion** → high-level modules depend on abstractions, not low-level details

## Interpretation: SOLID


SOLID principles keep code safe to change and easier to extend.

They reduce hidden dependencies, prevent fragile hierarchies, and encourage designs that adapt to new requirements without breaking existing functionality.

## HW4 Example: Animal Shelter API

- **Coupling** → Each module exposes only what is necessary through its API. For example, the Adoptions module interacts with Animals via endpoints, not direct database access. This reduces tight coupling.
- **Single Responsibility** → Each module handles one concern: Animals manage traits, Adoptions track requests, Users manage credentials.
- **Interface Segregation** → Clients only depend on the endpoints they need. A user registering does not need adoption endpoints.
- **Open/Closed** → New endpoints can be added (e.g., search filters) without modifying existing ones.

## HW4 Example: Animal Shelter API Testing

- Testing is performed by running a **bash script of unit tests**
- Unit tests validate each API endpoint (e.g., GET /animals, POST /adoptions)
- Ensures endpoints behave as expected and changes don't break existing functionality
-  Demonstrates iterative design: tests provide quick feedback and protect against regressions