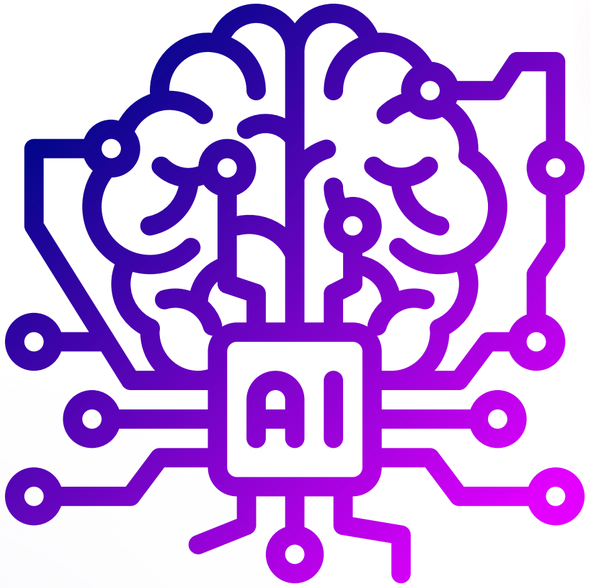
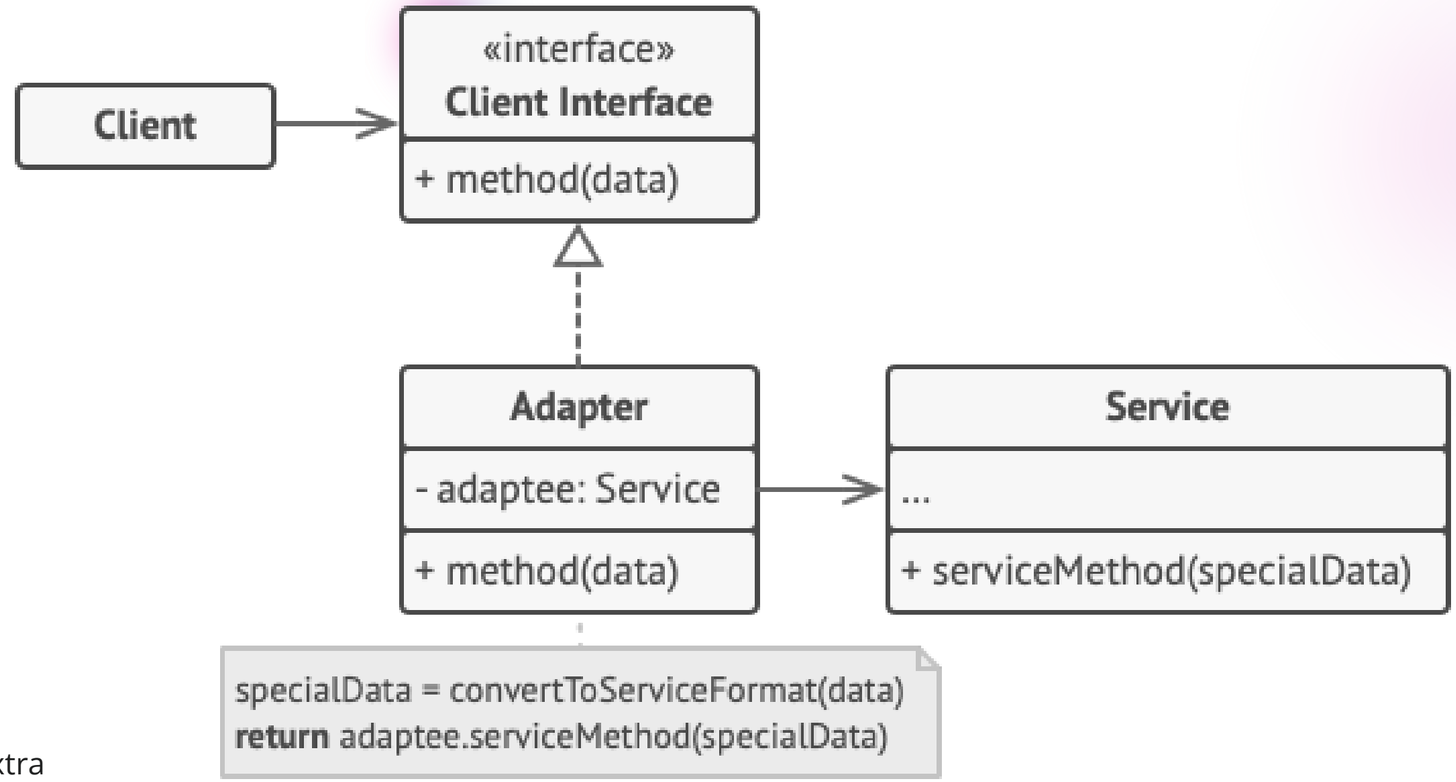



# DESIGN PATTERN

MADE BY REHAM

# STRUCTURAL
# PATTERN

# Adapter Pattern




## When to Use?

- Integrating old code with new systems
- Working with third-party libraries
- Promoting code reusability
- Bridging incompatible interfaces

## Cons

- Increased complexity: Adds an extra layer of code
- Performance overhead: Method calls go through an extra wrapper
- One-way conversion

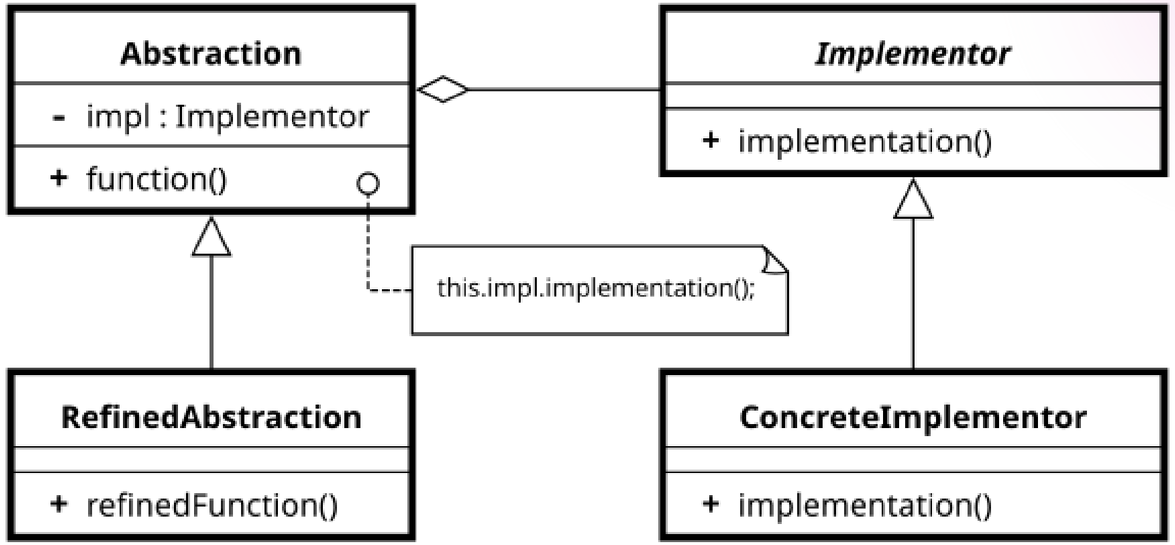
# Bridge Pattern

## When to Use?

- Divide a large class into two separate hierarchies
- Switching from inheritance to composition

## Cons

- Code may become complicated for a highly cohesive class
- Overkill for simple hierarchies: If you only have one dimension of variation (only OS, no UI types), a normal inheritance is enough.



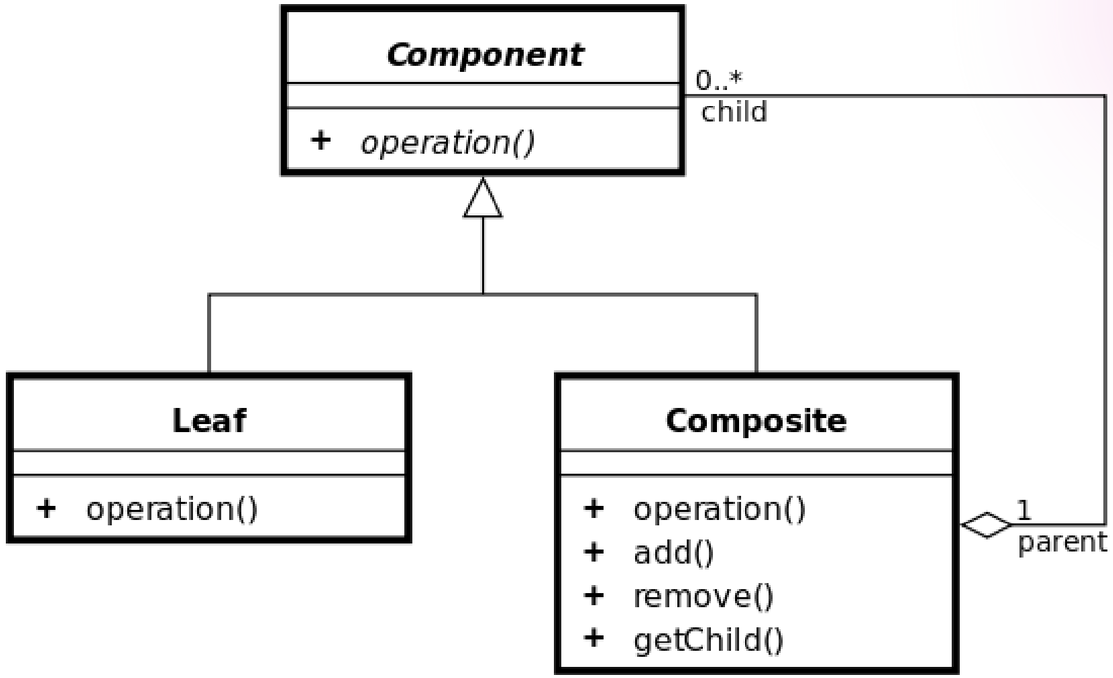
# Composite Pattern

## When to Use?

- When the core can be represented by a tree
- uses polymorphism when iterating on a complex structure
- open close principle (adding new component)

## Cons

- Overgeneralization (having the same interface for different and unrelated objects



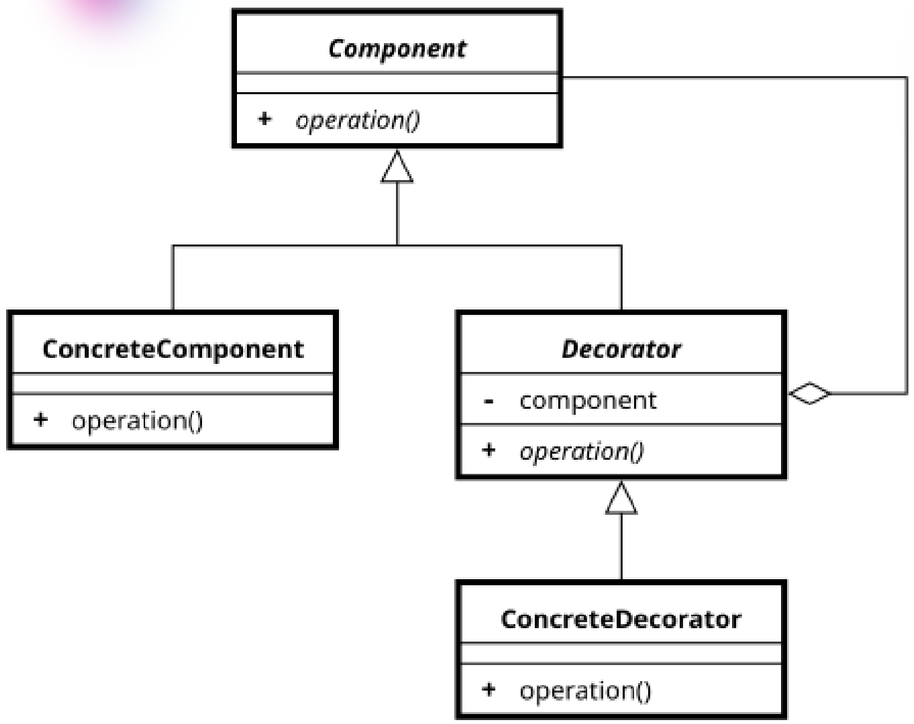
# Decorator Pattern

## When to Use?

- When you want to add responsibilities/behaviors dynamically at runtime without modifying the original class (Open/Closed Principle).
- When you have a large number of feature combinations
- When you want to prefer composition over inheritance (flexibility in extending behavior).

## Cons

- Too many objects:Each decorator creates a new object may lead to runtime overhead.
- Not ideal with too many decorators: If you have a huge number of decorators, the structure can become more complicated than using inheritance.



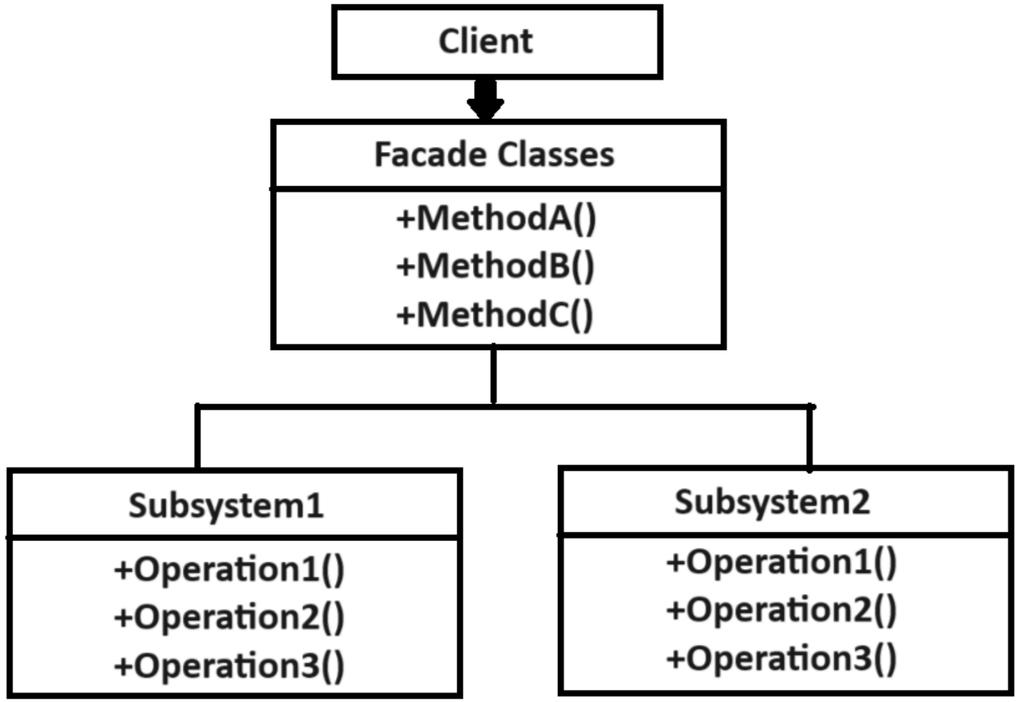
# Facade Pattern

### When to Use?

- When you want to simplify the client interaction with a complex system
- When the system has many subsystems and you want a unified entry point
- When you want to decouple clients from the internal structure (changes does not affect client)

### Cons

- Extra layer: Adds one more abstraction layer (small overhead).



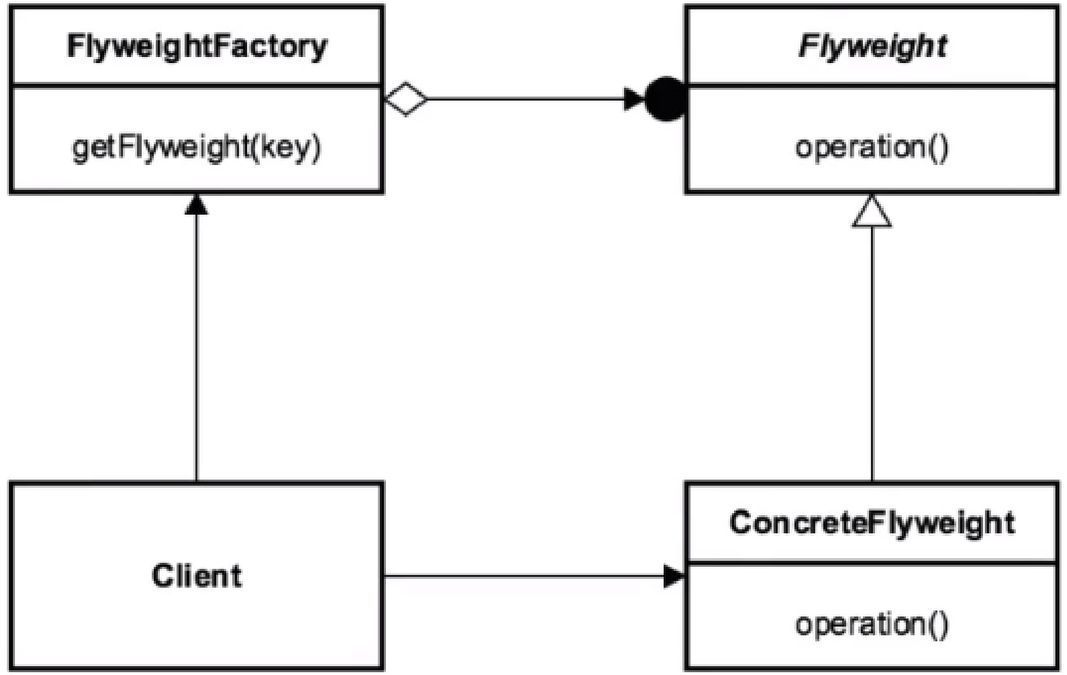
# FlyWeight Pattern

## When to Use?

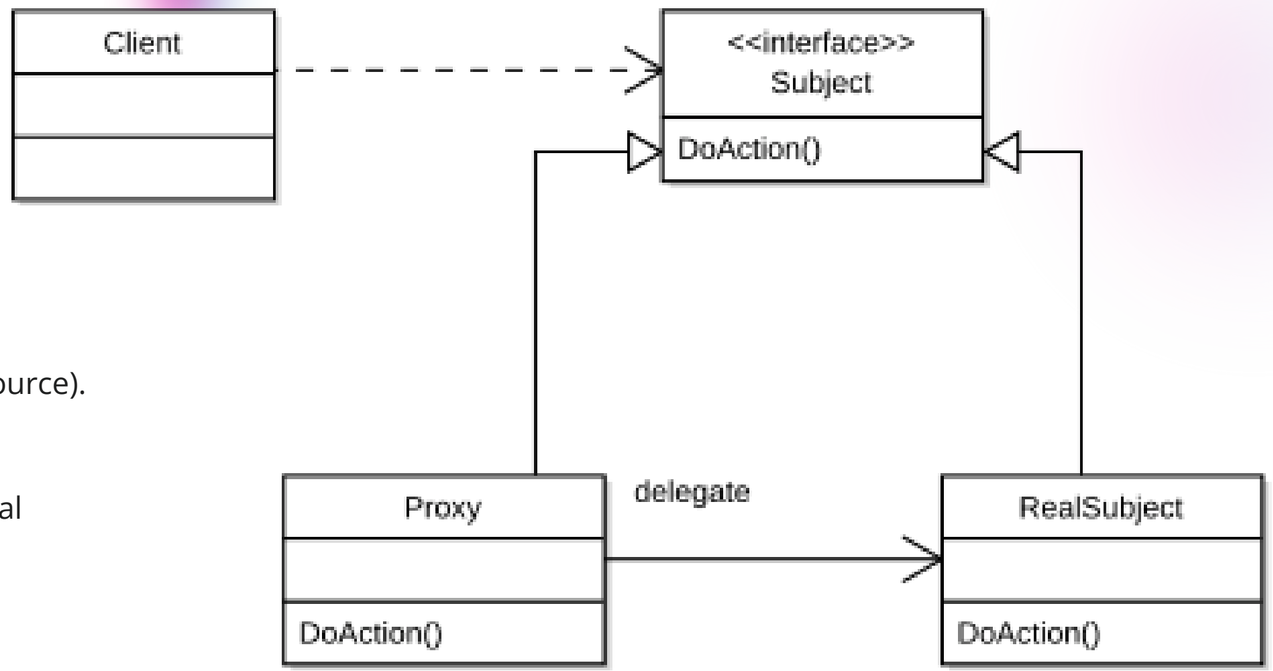
- When you have a large number of objects that share common intrinsic state.
- To reduce memory usage by reusing objects instead of creating duplicates.

## Cons

- Thread-safety issues if many clients access shared flyweights simultaneously.



# Proxy Pattern




## When to Use?

- When you need Access Control (e.g., security check before accessing resource).
- When you want Lazy Loading (create object only when needed).
- When you want to monitor/log requests without changing real object.
- When the real object is remote (e.g., API on server), and proxy acts as local representative.

## Cons

- Adds an extra layer: may slow down performance.
- Need to keep Proxy & Real Object in sync (if interface changes).

# BEHAVIORAL PATTERN

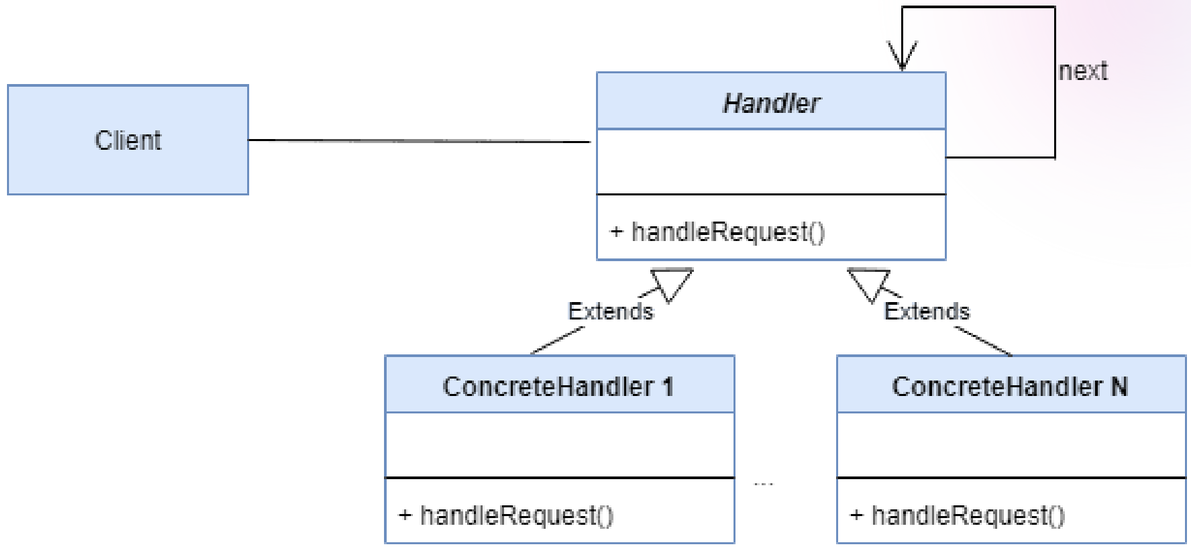
# Chain of Responsibility Pattern

## When to Use?

- Decoupling between sender & receiver
- Flexible Handling
- Avoiding Big if-else: replace with chain + polymorphism for clean code

## Cons

- No Guarantee of Handling
- Debugging is Harder
- Performance Issues



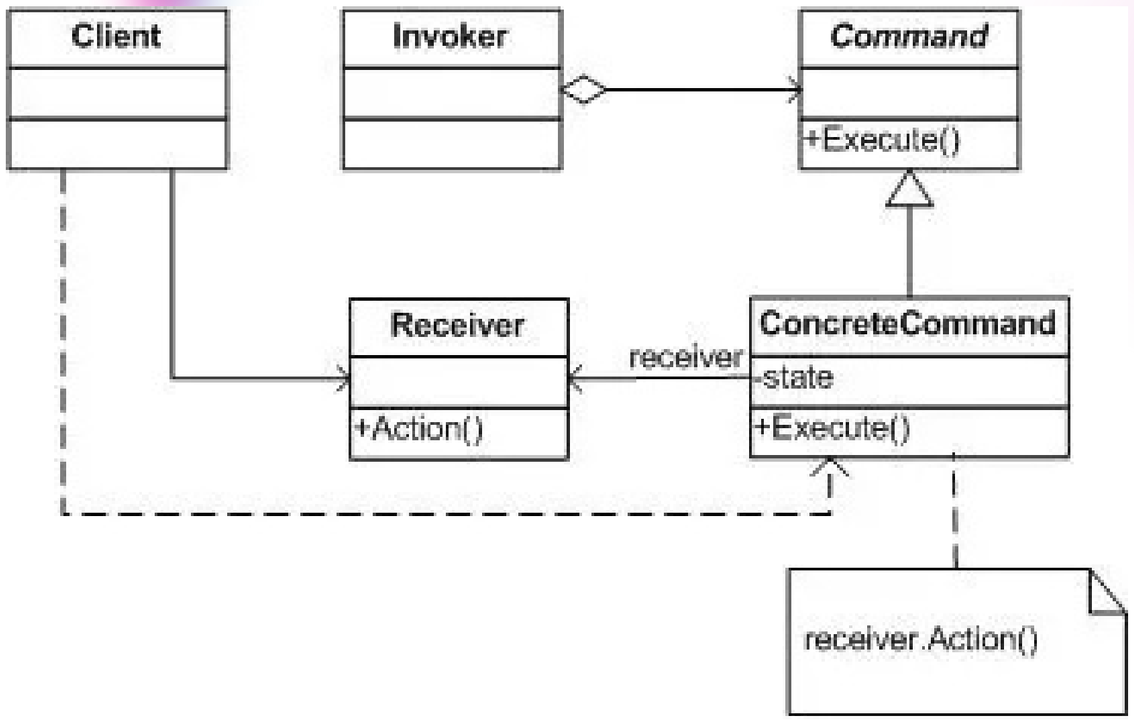
# Command Pattern

**When to Use?**

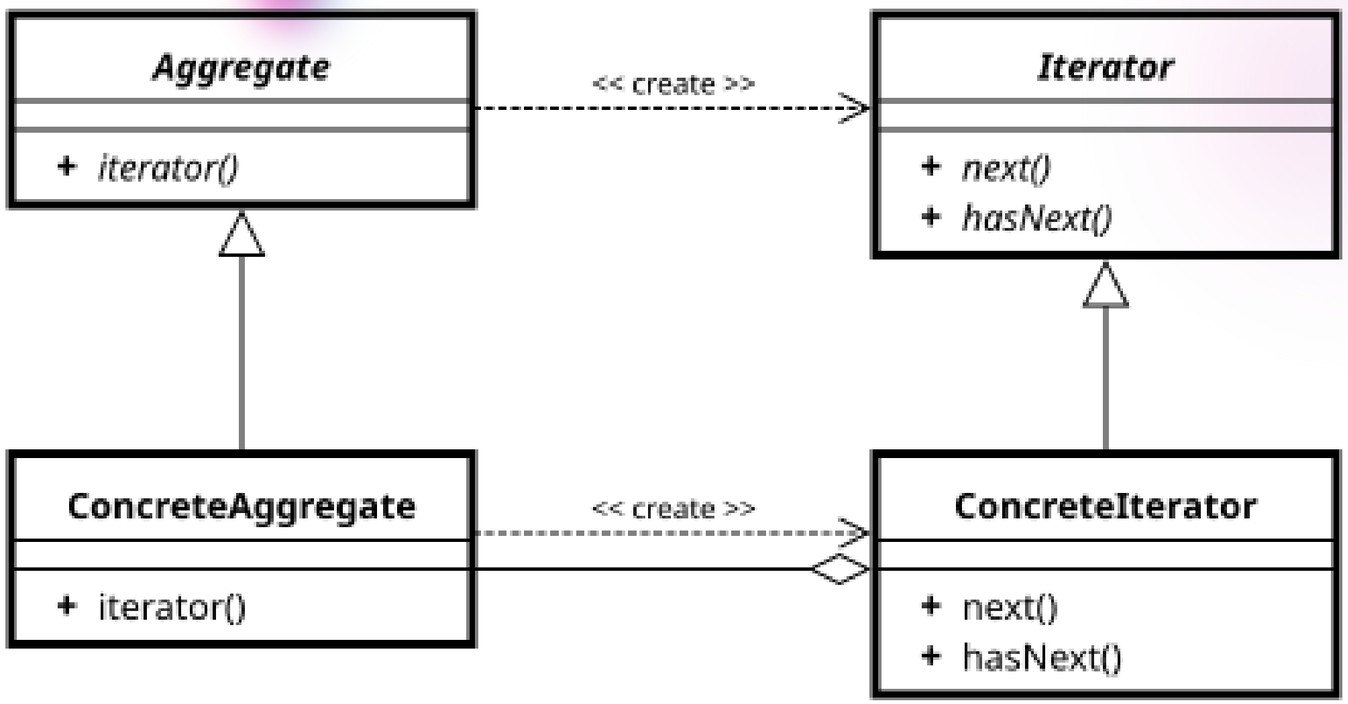
- Undo/Redo Support
- Macro Commands
- Decoupling

**Cons**

- Extra Classes
- Memory Overhead



# Iterator Pattern




## When to Use?

- Uniform Access: You want a standard way to traverse elements without exposing how the collection is implemented (array, list, tree, etc.).
- Multiple Iterations: You need more than one iterator at the same time (e.g., two players browsing the same inventory independently).

## Cons

- Single Direction (in simple versions) : Basic iterators often allow only forward traversal. For backward or random access, you need additional implementation.
- Thread-Safety Issues: If the collection is modified while being iterated (add/remove element), it can lead to errors (like ConcurrentModificationException in Java).

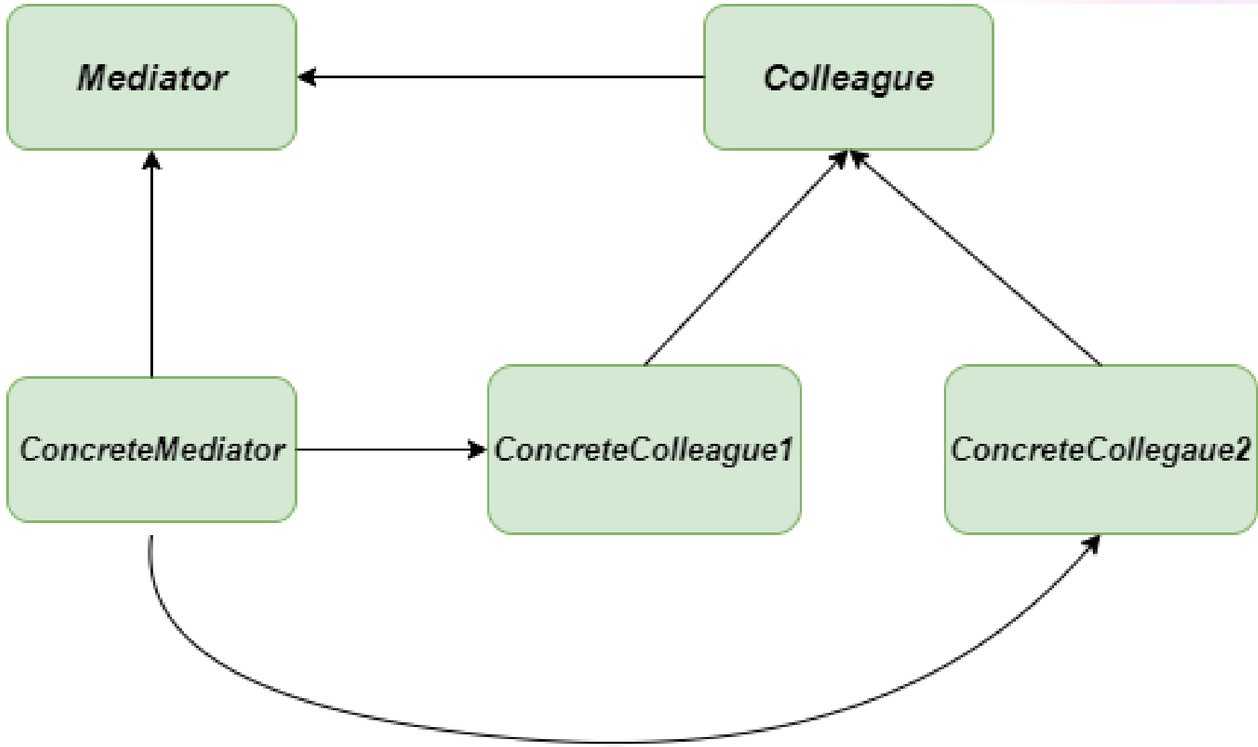
# Mediator Pattern

## When to Use?

- When multiple objects interact in complex ways (e.g., chat system, airplanes, UI components).
- When you want centralized control.

## Cons

- If Mediator fails : whole communication system fails.



# THANK YOU