Appunti sui Design Pattern

Cosa sono i Design Pattern?

I Design Pattern sono soluzioni ricorrenti a problemi comuni nella progettazione del software. Non sono librerie o codice pronto all'uso, ma **modelli concettuali** che descrivono come organizzare classi e oggetti per risolvere un problema specifico in modo flessibile, riutilizzabile e manutenibile.
Sono stati formalizzati nel libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* (1994) da Gamma, Helm, Johnson e Vlissides, noti come **"Gang of Four" (GoF)**.

Classificazione dei Design Pattern

I pattern GoF si dividono in tre categorie principali, in base allo **scopo**:

1. Pattern Creazionali (Creational Patterns)

Si occupano di **come creare oggetti**, incapsulando la logica di istanziazione per rendere il sistema indipendente da come gli oggetti sono creati, composti e rappresentati.
Obiettivo: rendere flessibile e riutilizzabile il meccanismo di creazione degli oggetti.

2. Pattern Strutturali (Structural Patterns)

Definiscono come **comporre classi e oggetti in strutture più grandi**, mantenendole flessibili ed efficienti.
Obiettivo: semplificare la progettazione identificando modi semplici di realizzare relazioni tra entità.

3. Pattern Comportamentali (Behavioral Patterns)

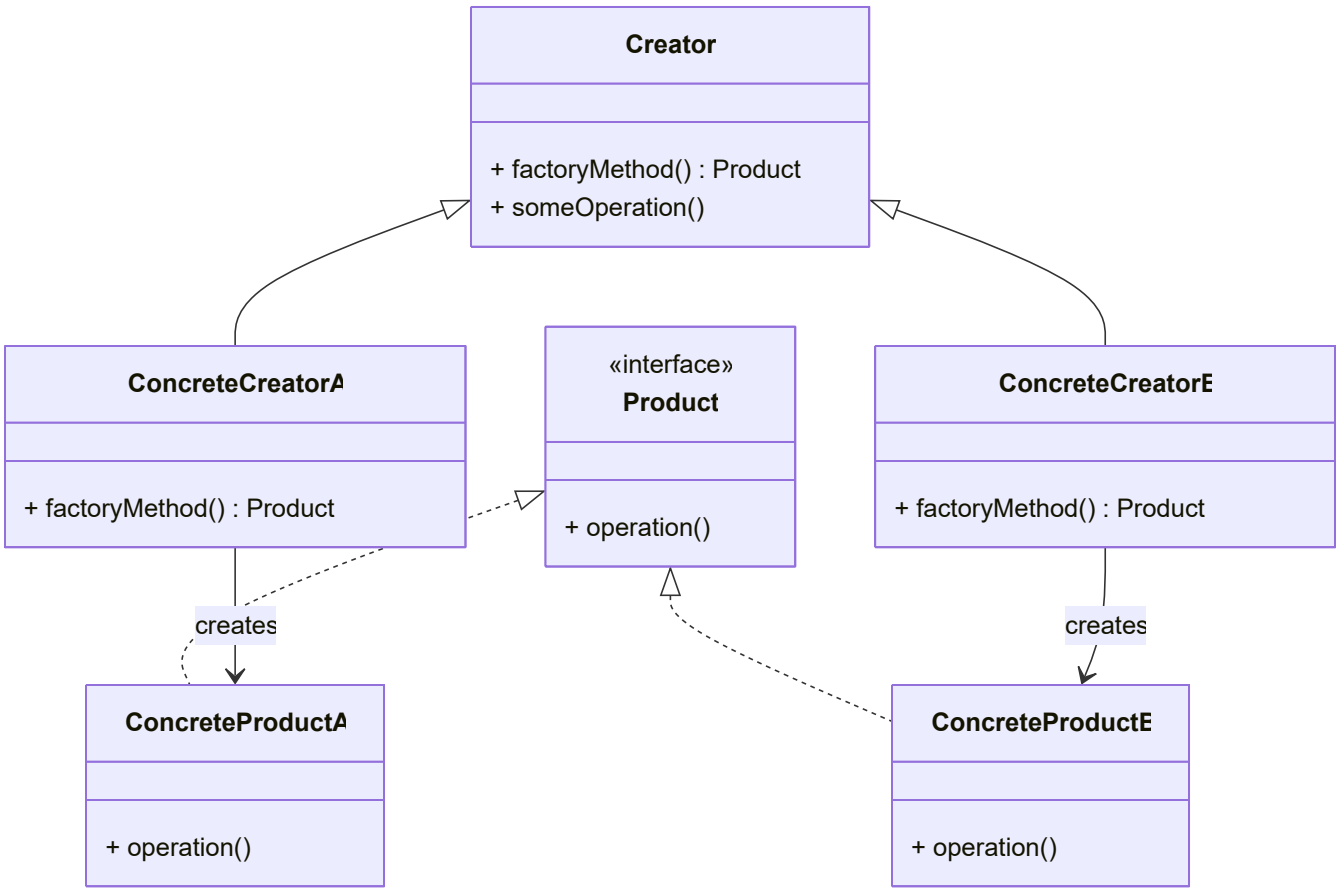
Definiscono il **modo in cui gli oggetti interagiscono e distribuiscono le responsabilità**.
Obiettivo: migliorare la comunicazione tra oggetti, rendendo il comportamento configurabile e estendibile.

Pattern Creazionali

1. Factory Method

Problema:
Quando una classe non può anticipare il tipo di oggetti che deve creare, o quando vuole delegare la creazione alle sue sottoclassi.
Esempio: un'applicazione di logistica che inizialmente gestisce solo camion, ma poi deve supportare anche spedizioni marittime.

Struttura UML:



Vantaggi:

- Incapsula la creazione degli oggetti.
- Rispetta il principio Open/Closed.
- Promuove il basso accoppiamento.

Svantaggi:

- Può introdurre troppe sottoclassi.
- Aumenta la complessità del codice.

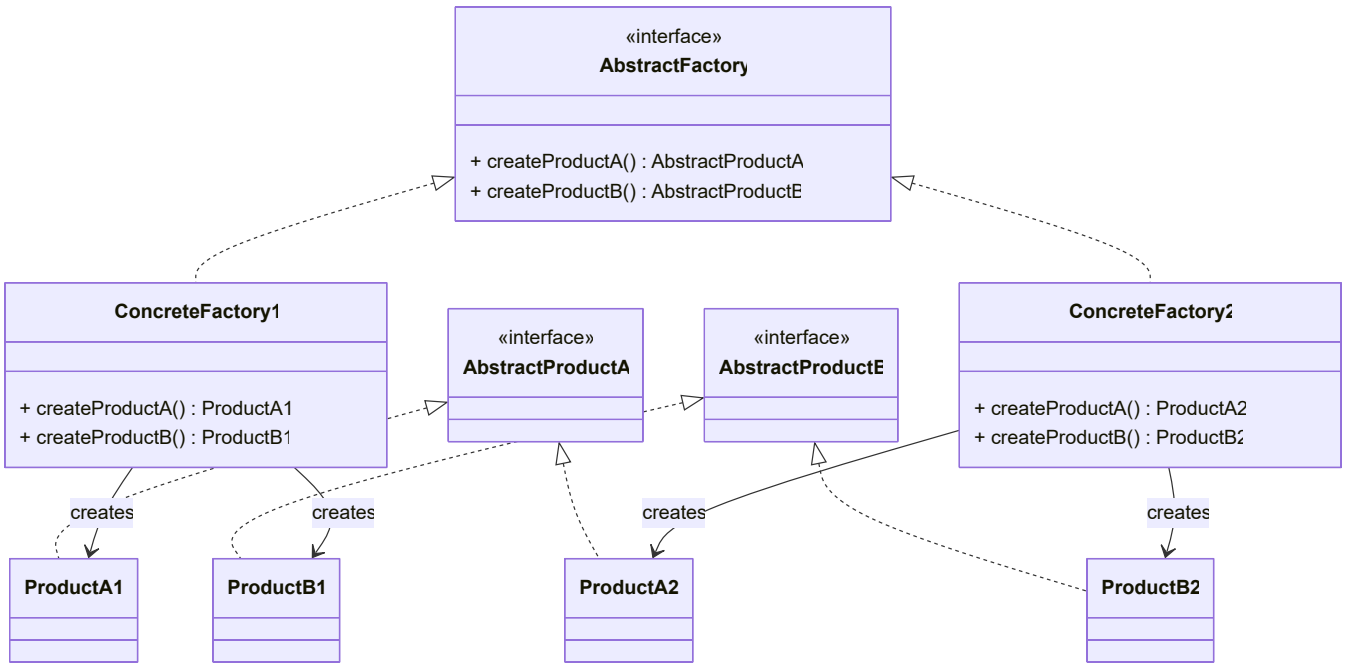
Relazioni:

- Spesso evolve verso Abstract Factory, Builder o Prototype.
- È un caso speciale di Template Method.

2. Abstract Factory

Problema:
Creare famiglie di oggetti correlati senza specificare le loro classi concrete.
Esempio: un sistema di arredamento che deve produrre sedie, divani e tavolini in stili coordinati (moderno, vittoriano).

Struttura UML:



Vantaggi:

- Isola il codice dalle classi concrete.
- Assicura la compatibilità tra prodotti correlati.
- Facilita il cambio di intere famiglie di prodotti.

Svantaggi:

- Aggiunge complessità con molte interfacce e classi.
- Difficile supportare nuovi tipi di prodotti.

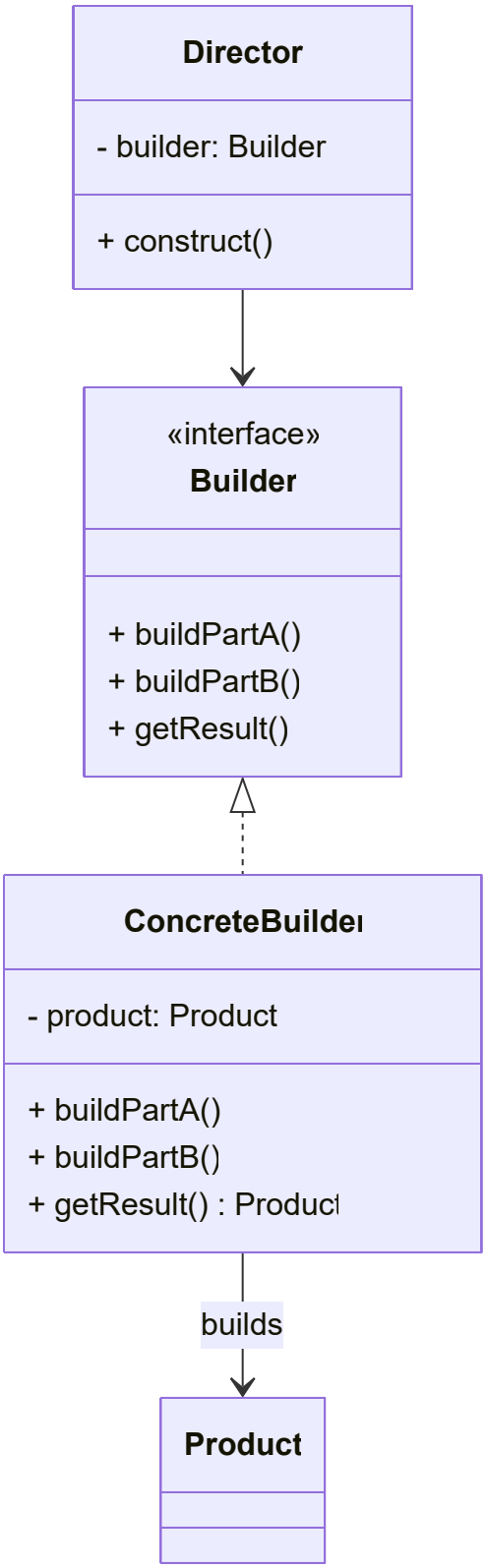
Relazioni:

- Spesso implementato con Factory Method.
- Alternativa a Facade per nascondere la creazione di sottosistemi.
- Può essere usato con Bridge.

3. Builder

Problema:
Creare oggetti complessi passo dopo passo, separando la costruzione dalla rappresentazione.
Esempio: costruire una casa con diverse opzioni (mura, porte, finestre, piscina).

Struttura UML:



Vantaggi:

- Permette di costruire oggetti passo passo.
- Riutilizza il codice di costruzione.
- Isola il codice di costruzione dalla business logic.

Svantaggi:

- Aumenta il numero di classi.
- Richiede un Director per gestire la costruzione.

Relazioni:

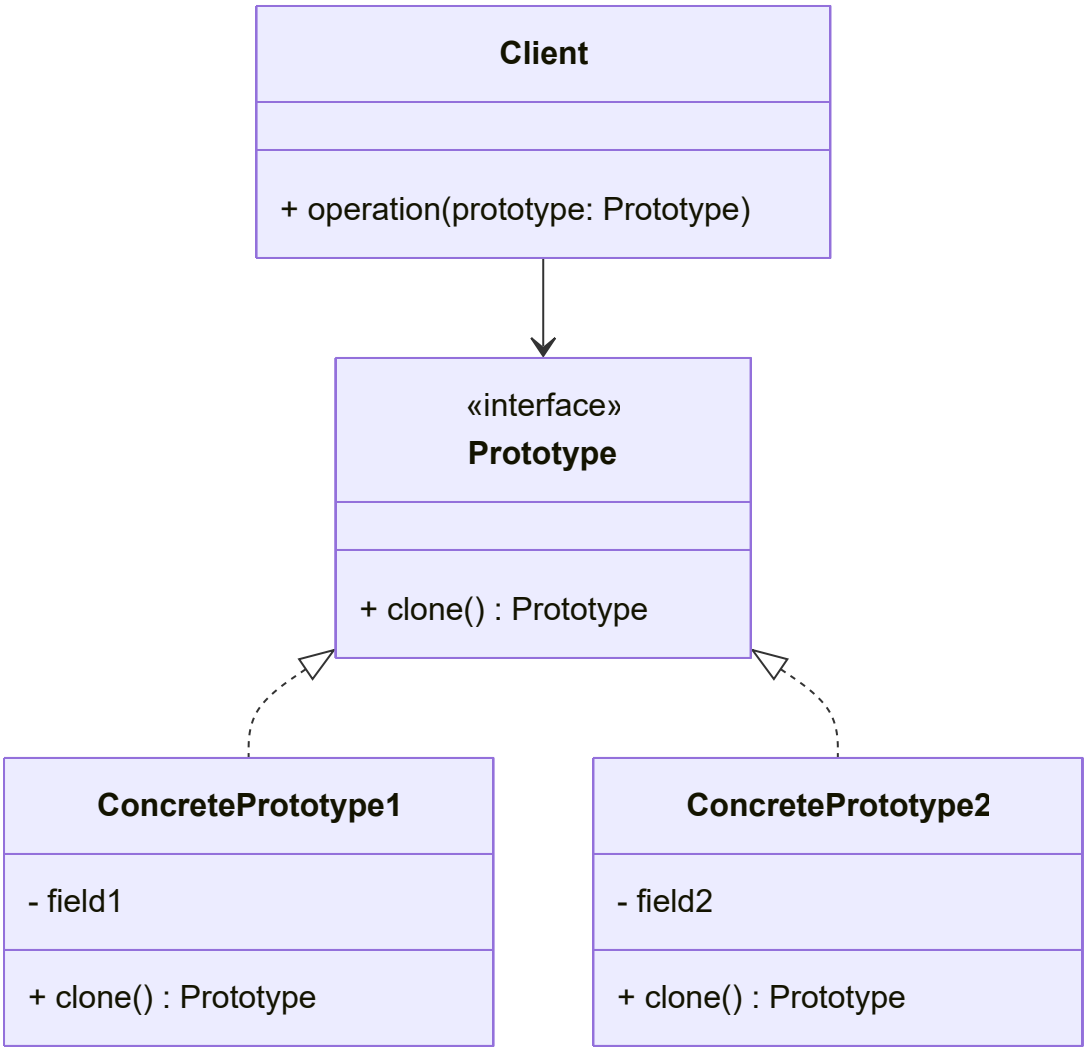
- Può essere combinato con Bridge.
- Abstract Factory restituisce prodotti subito, Builder permette passi aggiuntivi.
- Utile per costruire alberi Composite.

---

4. Prototype

Problema:
Creare copie di oggetti esistenti senza rendere il codice dipendente dalle loro classi.
Esempio: clonare forme geometriche (cerchi, rettangoli) senza conoscere i dettagli concreti.

Struttura UML:



Vantaggi:

- Clona oggetti senza accoppiamento alle classi concrete.
- Elimina codice di inizializzazione ripetuto.
- Alternativa all’ereditarietà per configurazioni complesse.

Svantaggi:

- Clonare oggetti con riferimenti circolari può essere complesso.
- Difficile clonare oggetti con stati interni complessi.

Relazioni:

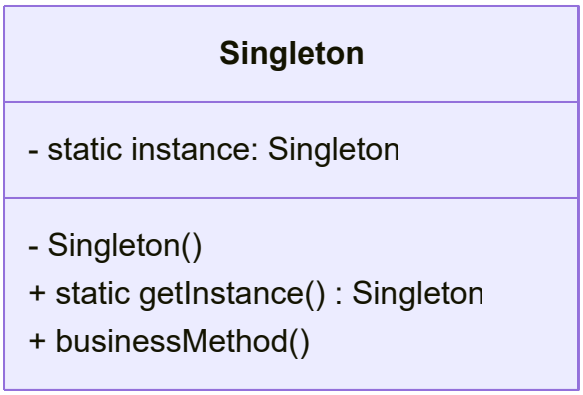
- Spesso usato con Abstract Factory e Builder.
- Alternativa a Factory Method quando l’inizializzazione è costosa.
- Utile con Composite e Decorator per clonare strutture complesse.

---

5. Singleton

Problema:
Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale a essa.
Esempio: una connessione al database condivisa in tutta l'applicazione.

Struttura UML:



Vantaggi:

- Controllo sull'unica istanza.
- Accesso globale.
- Inizializzazione lazy.

Svantaggi:

- Viola il Single Responsibility Principle.
- Difficile testare.
- Può mascherare un design scadente.

Relazioni:

- Può essere usato con Abstract Factory, Builder, Prototype.
- Facade spesso diventa Singleton.
- Flyweight simile ma con più istanze immutabili.

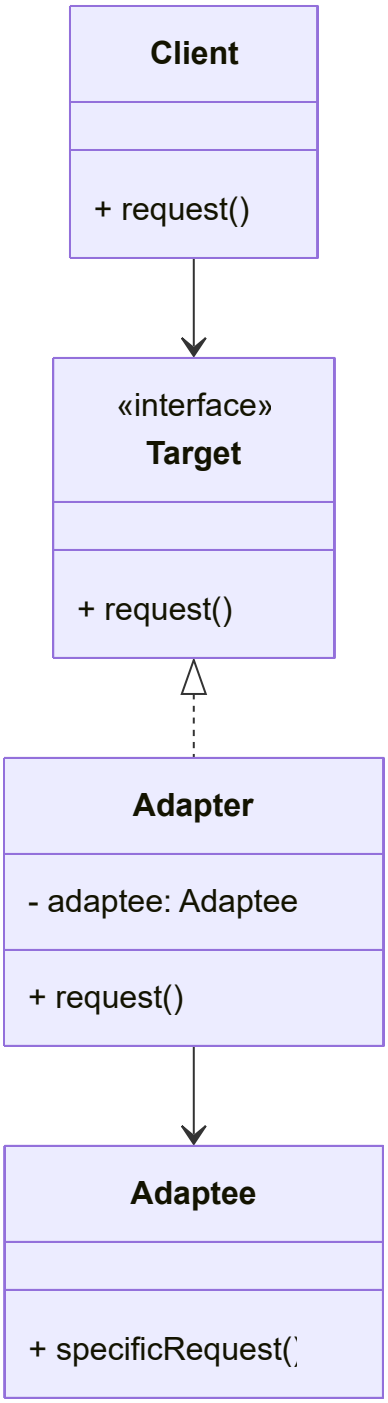
Pattern Strutturali

1. Adapter

Problema:
Far collaborare classi con interfacce incompatibili.

Esempio: adattare un’API che restituisce XML a un sistema che si aspetta JSON.

Struttura UML (Object Adapter):



Vantaggi:

- Rende compatibili interfacce diverse.
- Rispetta Open/Closed.

Svantaggi:

- Aumenta la complessità con nuove classi.

- A volte è meglio modificare l’interfaccia originale.

Relazioni:

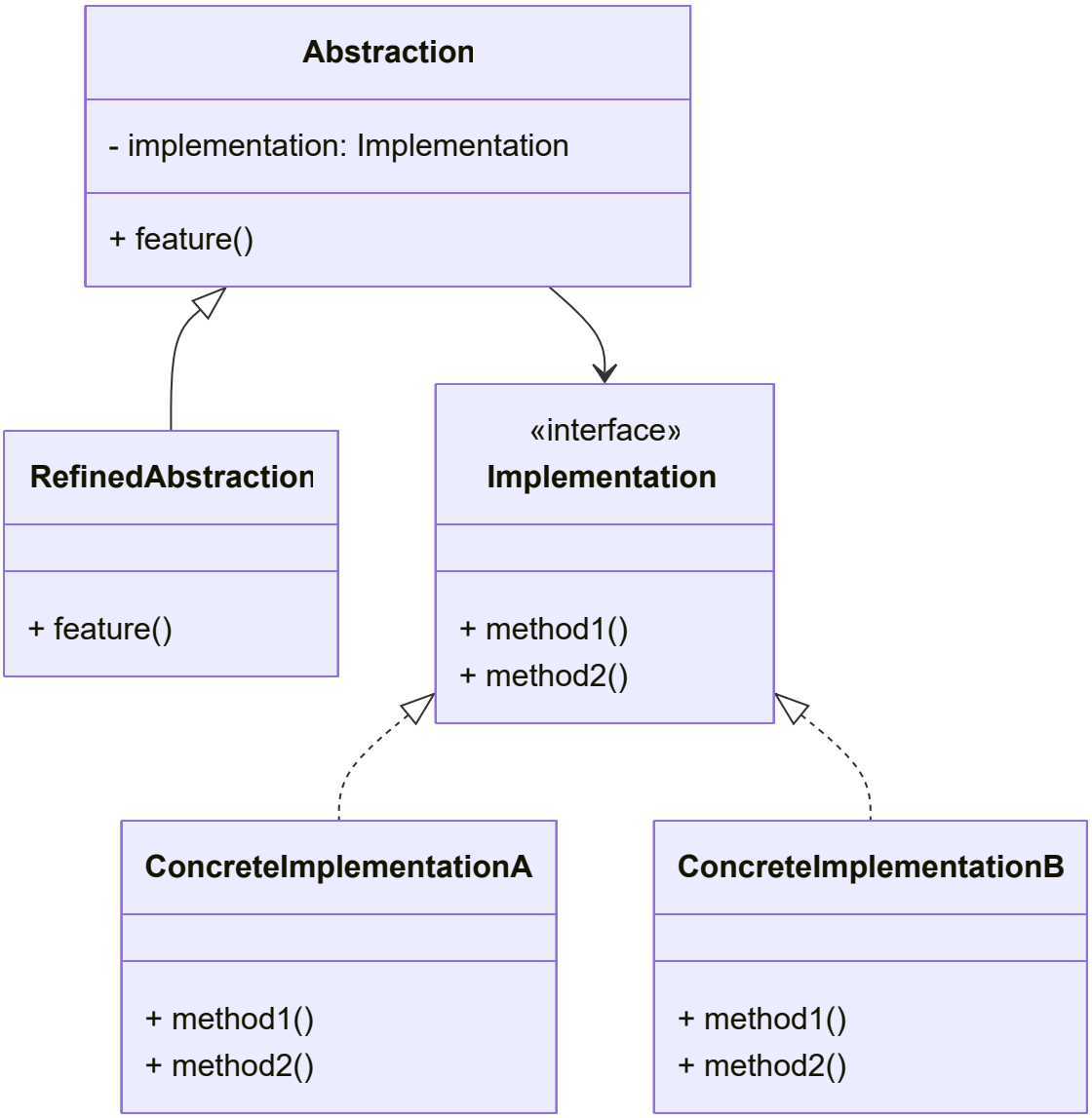
- Bridge è progettato a priori, Adapter per integrare codice esistente.
- Decoratore estende il comportamento, Adapter cambia l’interfaccia.
- Simile a Facade ma Adapter lavora con un solo oggetto.

---

2. Bridge

Problema:
Separare un’astrazione dalla sua implementazione per poterle cambiare indipendentemente.
Esempio: disaccoppiare una GUI (astrazione) dalle API del sistema operativo (implementazione).

Struttura UML:



Vantaggi:

- Permette di cambiare implementazioni a runtime.
- Rispetta Open/Closed e Single Responsibility.

Svantaggi:

- Può complicare eccessivamente il codice.

Relazioni:

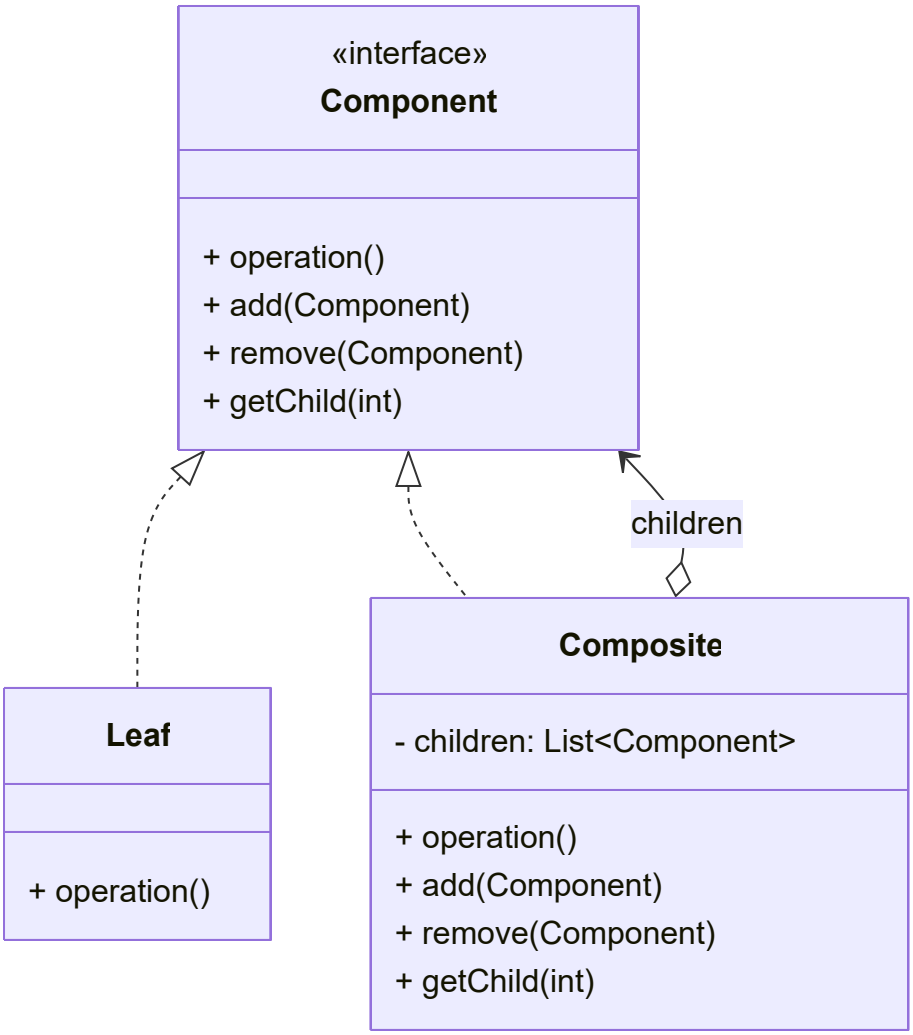
- Spesso usato con Abstract Factory.
- Può essere confuso con Strategy o Adapter.

---

3. Composite

Problema:
Comporre oggetti in strutture ad albero e trattare singoli oggetti e composizioni in modo uniforme.
Esempio: un sistema di grafica che gestisce sia forme semplici (cerchi) che gruppi di forme.

Struttura UML:



Vantaggi:

- Tratta uniformemente oggetti semplici e complessi.
- Facilita l'aggiunta di nuovi tipi di componenti.

Svantaggi:

- Può rendere il design troppo generale.
- Difficile limitare i tipi di componenti in una composizione.

Relazioni:

- Spesso usato con Iterator e Visitor.

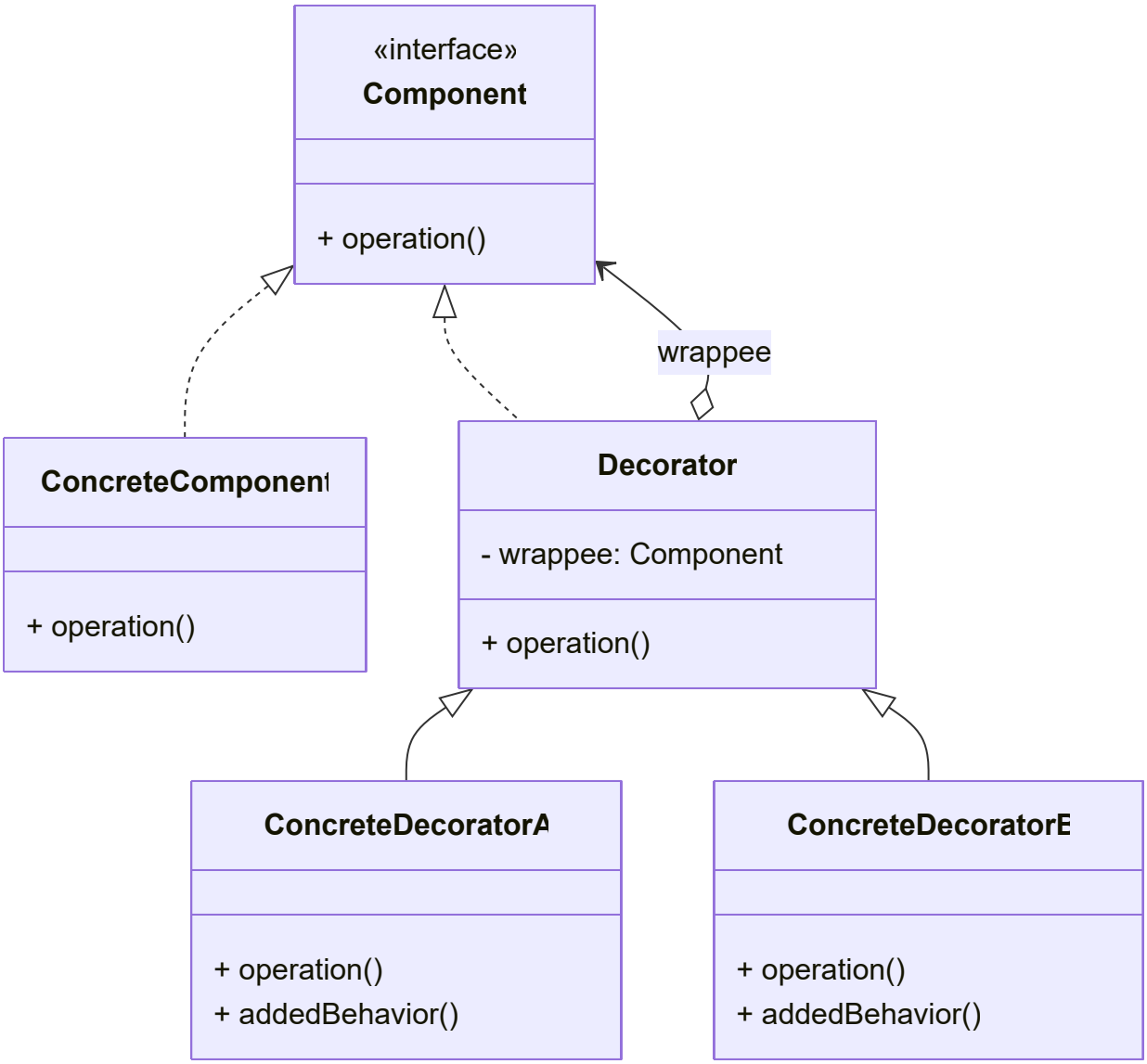
- Builder può costruire alberi Composite.
- Decorator può estendere singoli componenti nel Composite.

---

4. Decorator

Problema:
Aggiungere responsabilità a oggetti in modo dinamico e flessibile, senza usare l'ereditarietà.
Esempio: aggiungere cifratura e compressione a un flusso di dati.

Struttura UML:



Vantaggi:

- Aggiunge comportamenti senza sottoclassare.
- Combina comportamenti a runtime.

- Rispetta Single Responsibility.

Svantaggi:

- Difficile rimuovere decoratori specifici.
- L’ordine dei decoratori può influenzare il risultato.

Relazioni:

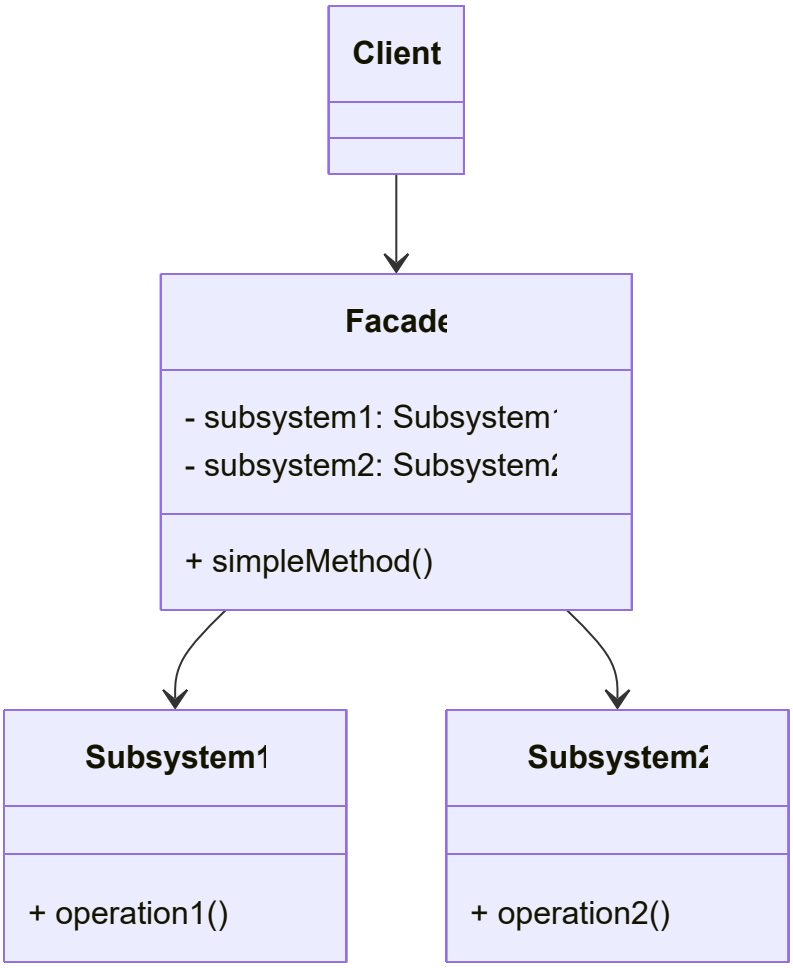
- Adapter cambia l’interfaccia, Decorator la estende.
- Composite simile nella struttura ma diverso nell’intento.
- Strategy cambia l’algoritmo, Decorator aggiunge funzionalità.

---

5. Facade

Problema:
Fornire un’interfaccia semplice a un sottosistema complesso.
Esempio: una classe che semplifica l’uso di una libreria di conversione video.

Struttura UML:



Vantaggi:

- Isola la complessità.
- Rende il sottosistema più facile da usare.

Svantaggi:

- Può diventare un “god object”.
- Nasconde funzionalità avanzate.

Relazioni:

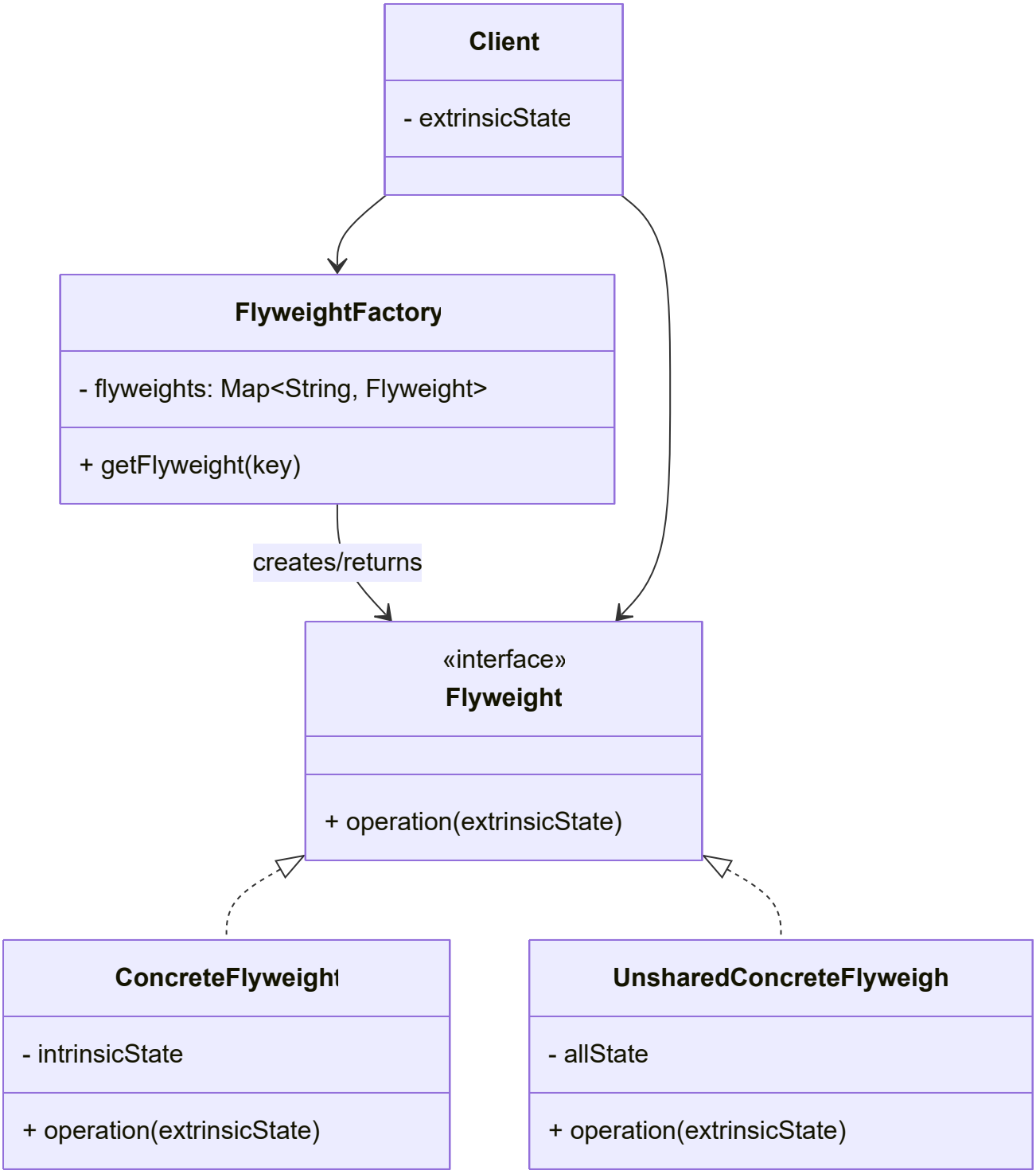
- Abstract Factory può nascondere la creazione di oggetti come Facade.
- Mediator simile ma per coordinare oggetti.
- Spesso implementato come Singleton.

---

6. Flyweight

Problema:
Ridurre l’uso di memoria condividendo parti comuni di stato tra molti oggetti.
Esempio: rendere milioni di particelle in un gioco condividendo testo e colore.

Struttura UML:



Vantaggi:

- Riduce drasticamente l’uso di RAM.
- Condivide efficacemente stati comuni.

Svantaggi:

- Aumenta la complessità.
- Può sacrificare CPU per risparmiare memoria.

Relazioni:

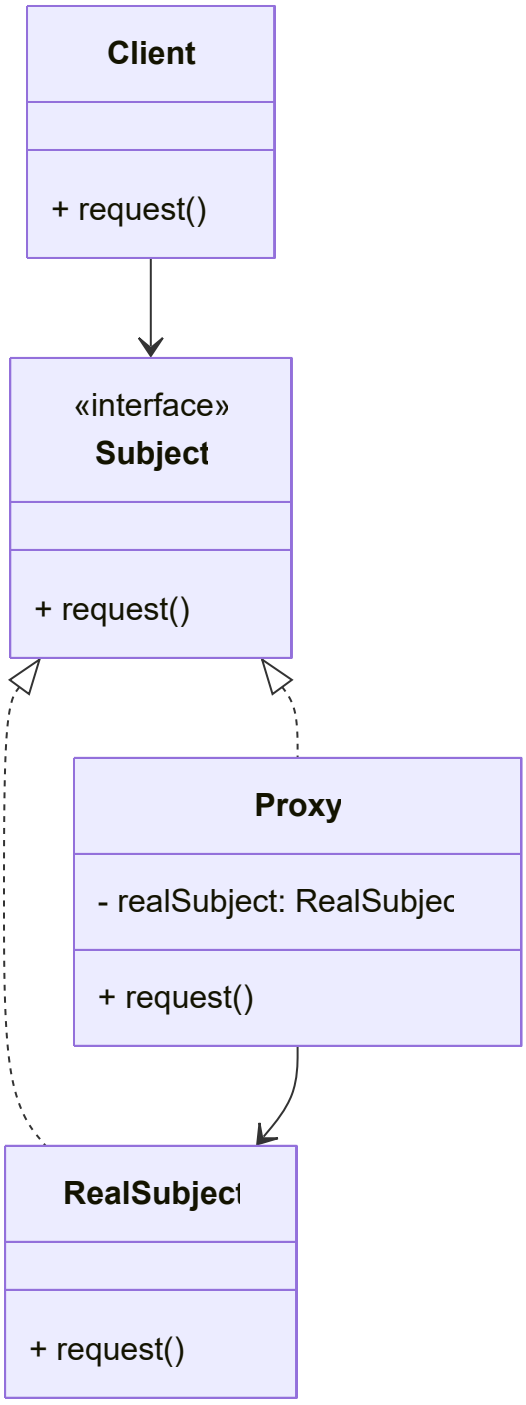
- Spesso usato con Composite per foglie condivise.
- Singleton ha una sola istanza, Flyweight molte.
- Facade mostra un oggetto che rappresenta un sottosistema, Flyweight molti oggetti piccoli.

---

7. Proxy

Problema:
Controllare l’accesso a un oggetto, aggiungendo logica prima o dopo la chiamata.
Esempio: caching, logging, lazy initialization di oggetti pesanti.

Struttura UML:



Vantaggi:

- Controlla l'accesso all'oggetto reale.
- Gestisce il ciclo di vita dell'oggetto.
- Può lavorare anche se l'oggetto reale non è disponibile.

Svantaggi:

- Può introdurre latenza.
- Aumenta la complessità.

Relazioni:

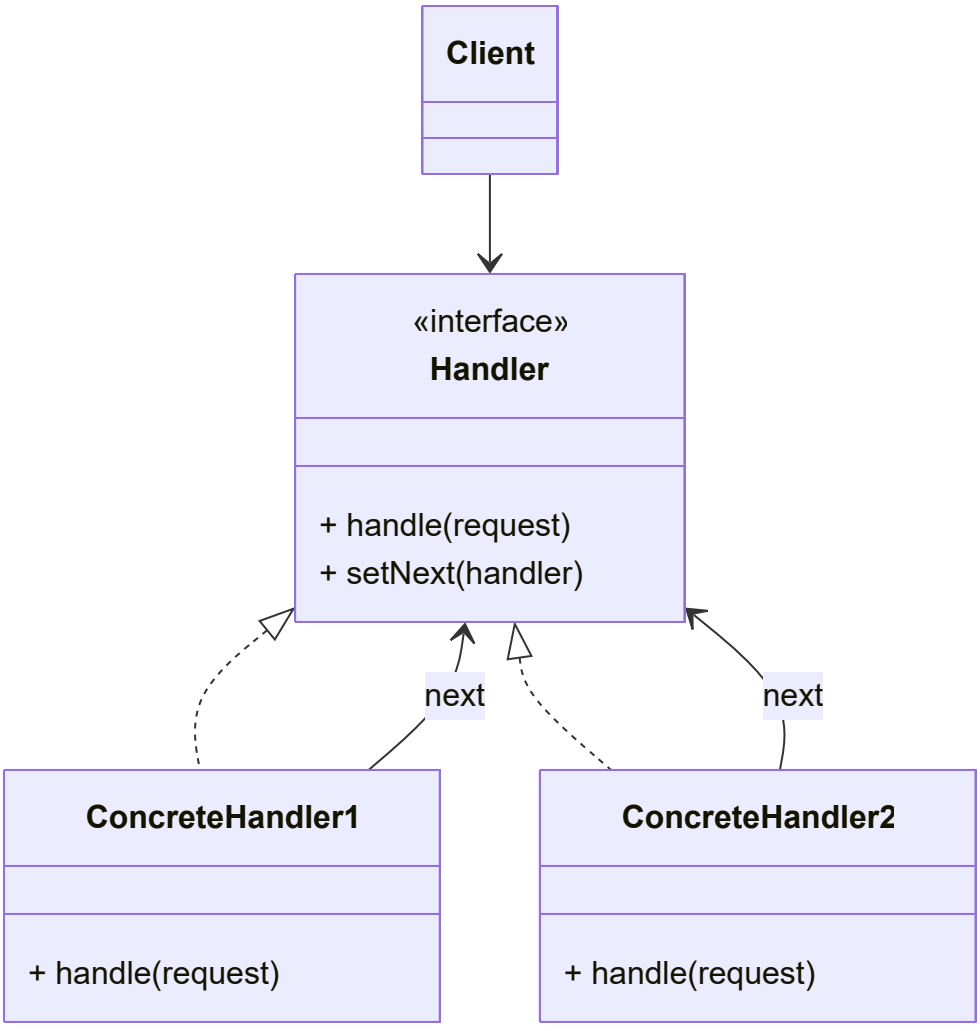
- Decorator aggiunge comportamento, Proxy controlla l'accesso.
- Adapter cambia interfaccia, Proxy la mantiene.
- Facade può essere simile ma per interi sottosistemi.

Pattern Comportamentali

1. Chain of Responsibility

Problema:
Passare una richiesta lungo una catena di handler fino a che uno non la gestisce.
Esempio: sistema di autenticazione e validazione di una richiesta web.

Struttura UML:



Vantaggi:

- Disaccoppia sender e receiver.
- Permette di aggiungere handler dinamicamente.
- Rispetta Single Responsibility.

Svantaggi:

- Le richieste possono rimanere senza risposta.
- Difficile debug per la catena dinamica.

Relazioni:

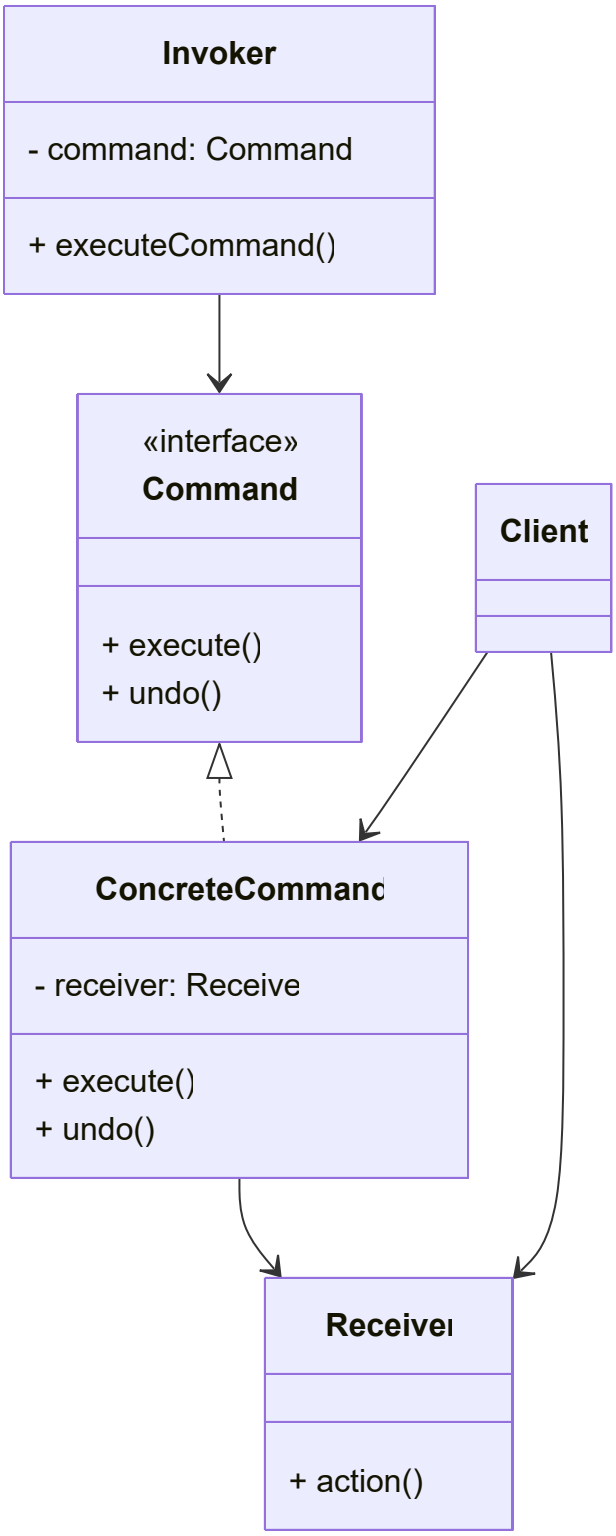
- Spesso usato con Composite.
- Simile a Decorator nella struttura ma diverso nell'intento.
- Può essere implementato con Command.

---

2. Command

Problema:
Incapsulare una richiesta come oggetto, permettendo di parametrizzare, mettere in coda, loggare o annullare operazioni.
Esempio: sistema di undo/redo in un editor di testo.

Struttura UML:



Vantaggi:

- Disaccoppia l'invoker dal receiver.
- Supporta undo/redo, logging, coda.
- Rispetta Open/Closed.

Svantaggi:

- Aumenta il numero di classi.
- Può diventare complesso.

Relazioni:

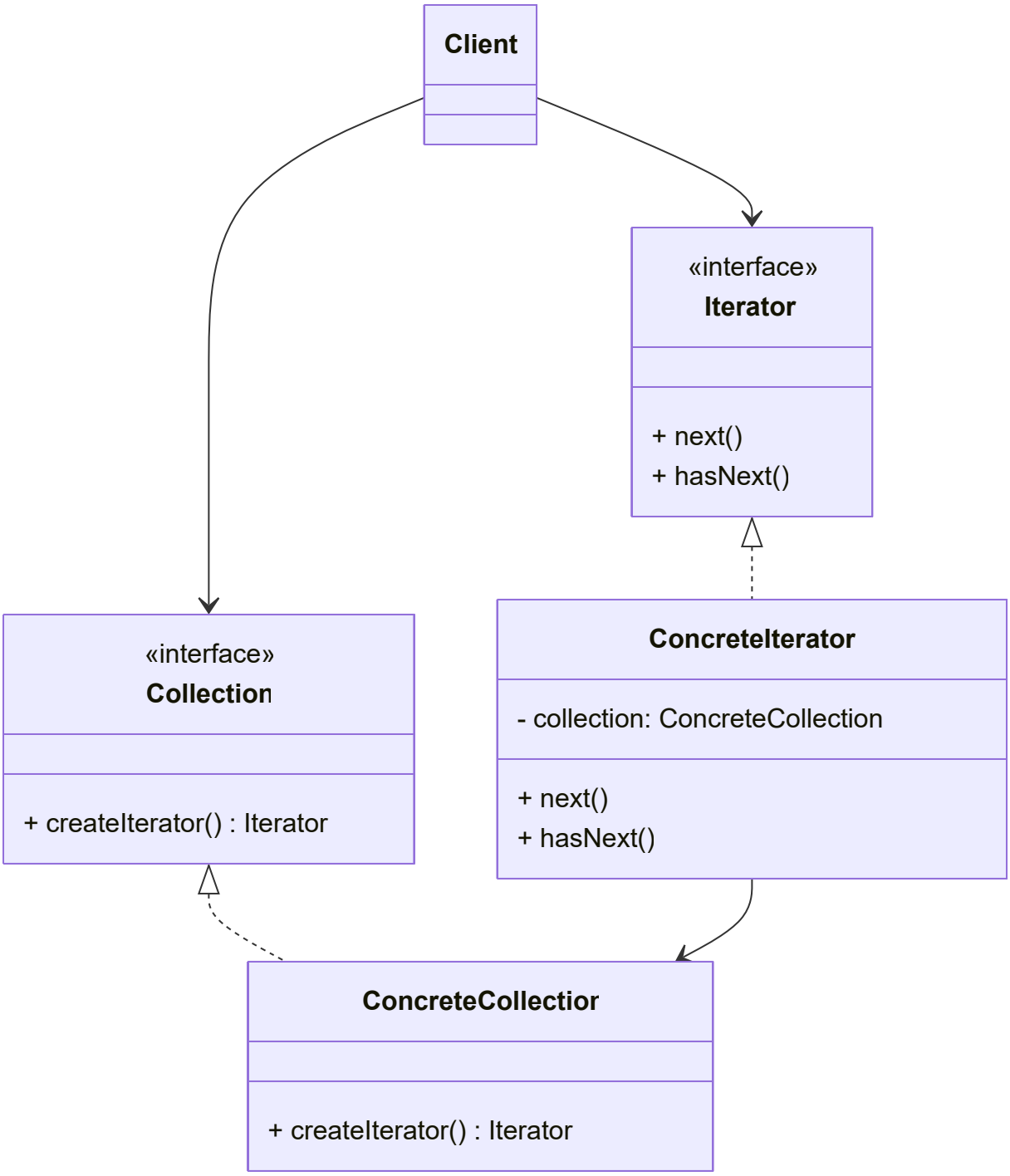
- Può essere usato con Memento per salvare lo stato.
- Strategy cambia algoritmi, Command incapsula richieste.
- Prototype può clonare comandi per la storia.

---

3. Iterator

Problema:
Permettere di attraversare una collezione senza esporne la struttura interna.
Esempio: iterare su una lista, un albero o un grafo in modo uniforme.

Struttura UML:



Vantaggi:

- Astrae la logica di traversamento.
- Supporta più iteratori sulla stessa collezione.
- Rispetta Single Responsibility.

Svantaggi:

- Può essere overkill per collezioni semplici.
- Alcuni linguaggi lo forniscono nativamente.

Relazioni:

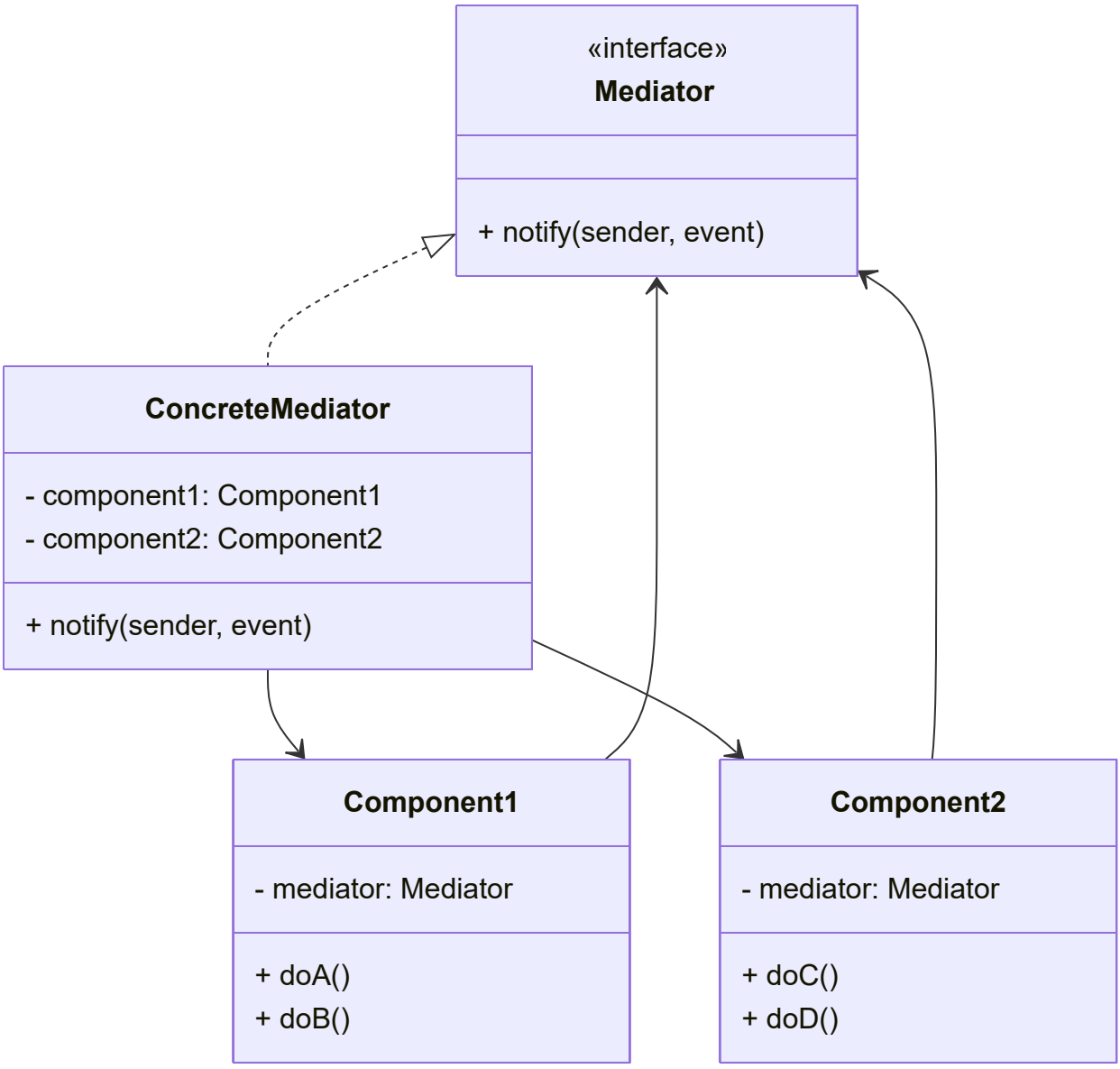
- Spesso usato con Composite.
- Può essere implementato con Factory Method nella collezione.

---

4. Mediator

Problema:
Ridurre le dipendenze caotiche tra oggetti, centralizzando la comunicazione in un mediatore.
Esempio: gestire l'interazione tra componenti GUI (bottoni, campi, liste).

Struttura UML:



Vantaggi:

- Riduce l’accoppiamento tra componenti.
- Centralizza il controllo.
- Facilita l’estensione.

Svantaggi:

- Può diventare un collo di bottiglia.
- Il mediatore può diventare troppo complesso.

Relazioni:

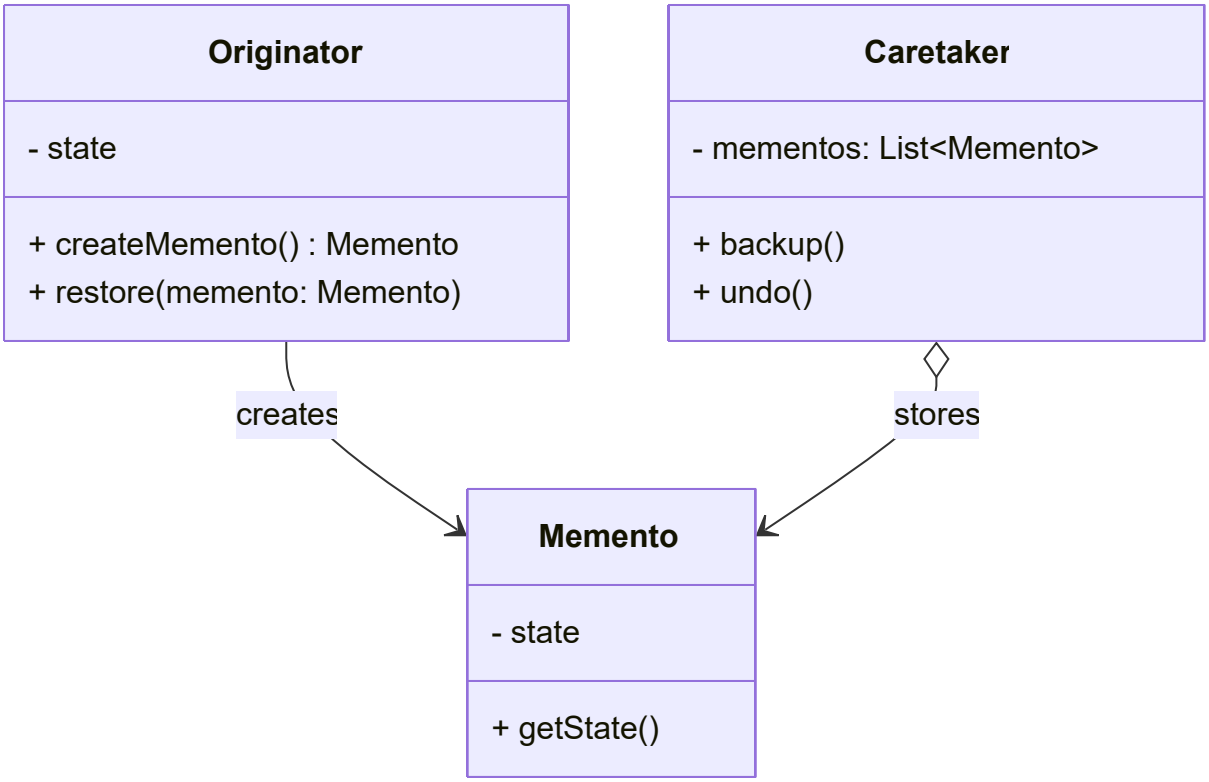
- Observer distribuisce eventi, Mediator li centralizza.
- Facade semplifica l’interfaccia, Mediator coordina oggetti.

---

5. Memento

Problema:
Salvare e ripristinare lo stato di un oggetto senza violarne l’incapsulamento.
Esempio: sistema di undo in un editor di testo.

Struttura UML:



Vantaggi:

- Preserva l’incapsulamento.
- Facilita il rollback dello stato.

Svantaggi:

- Può consumare molta memoria se lo stato è grande.
- Difficile gestire memento di oggetti complessi.

Relazioni:

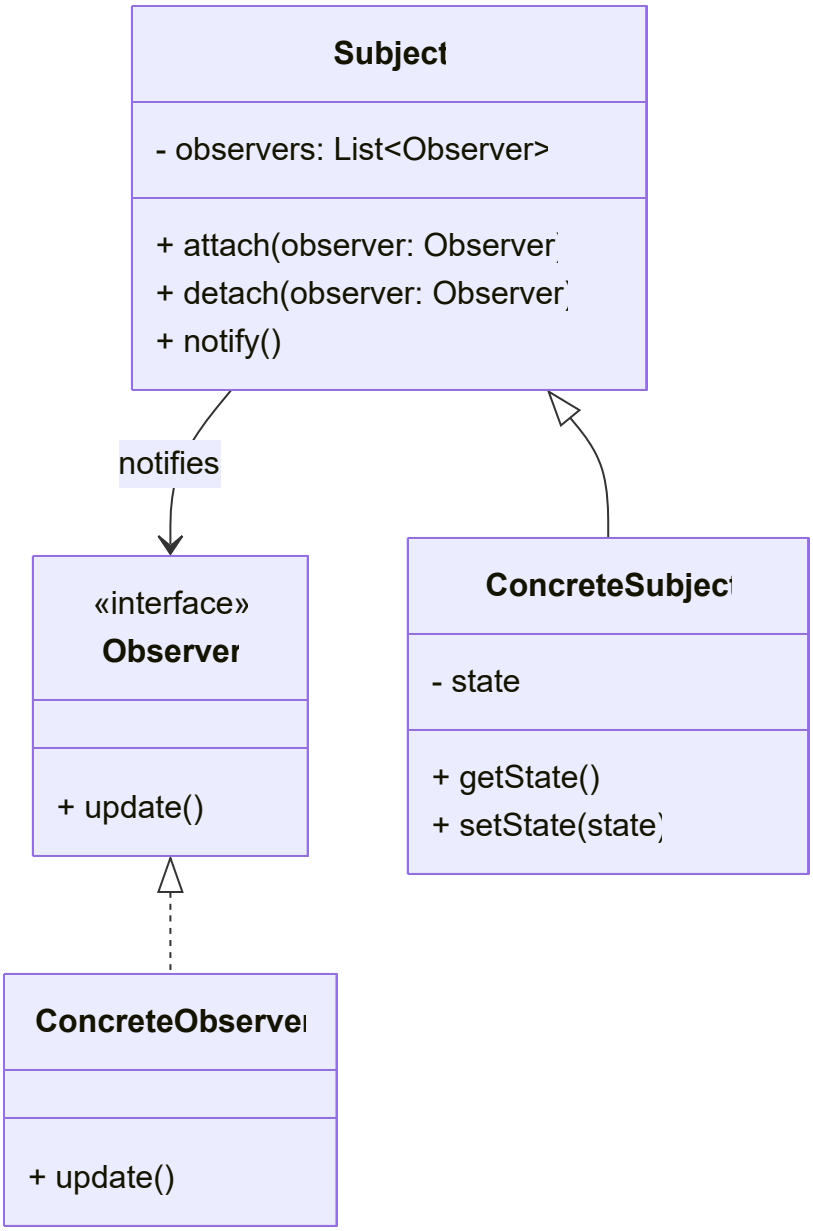
- Spesso usato con Command per undo/redo.
- Prototype può essere un’alternativa più semplice.

---

6. Observer

Problema:
Definire un meccanismo di notifica per oggetti dipendenti quando cambia lo stato di un soggetto.
Esempio: sistema di notifiche in un’app (email, SMS, push).

Struttura UML:



Vantaggi:

- Disaccoppia soggetto e osservatori.
- Supporta broadcast di eventi.
- Facile aggiungere nuovi osservatori.

Svantaggi:

- Notifiche non garantite (se un observer fallisce).
- Può causare performance issues con molti observer.

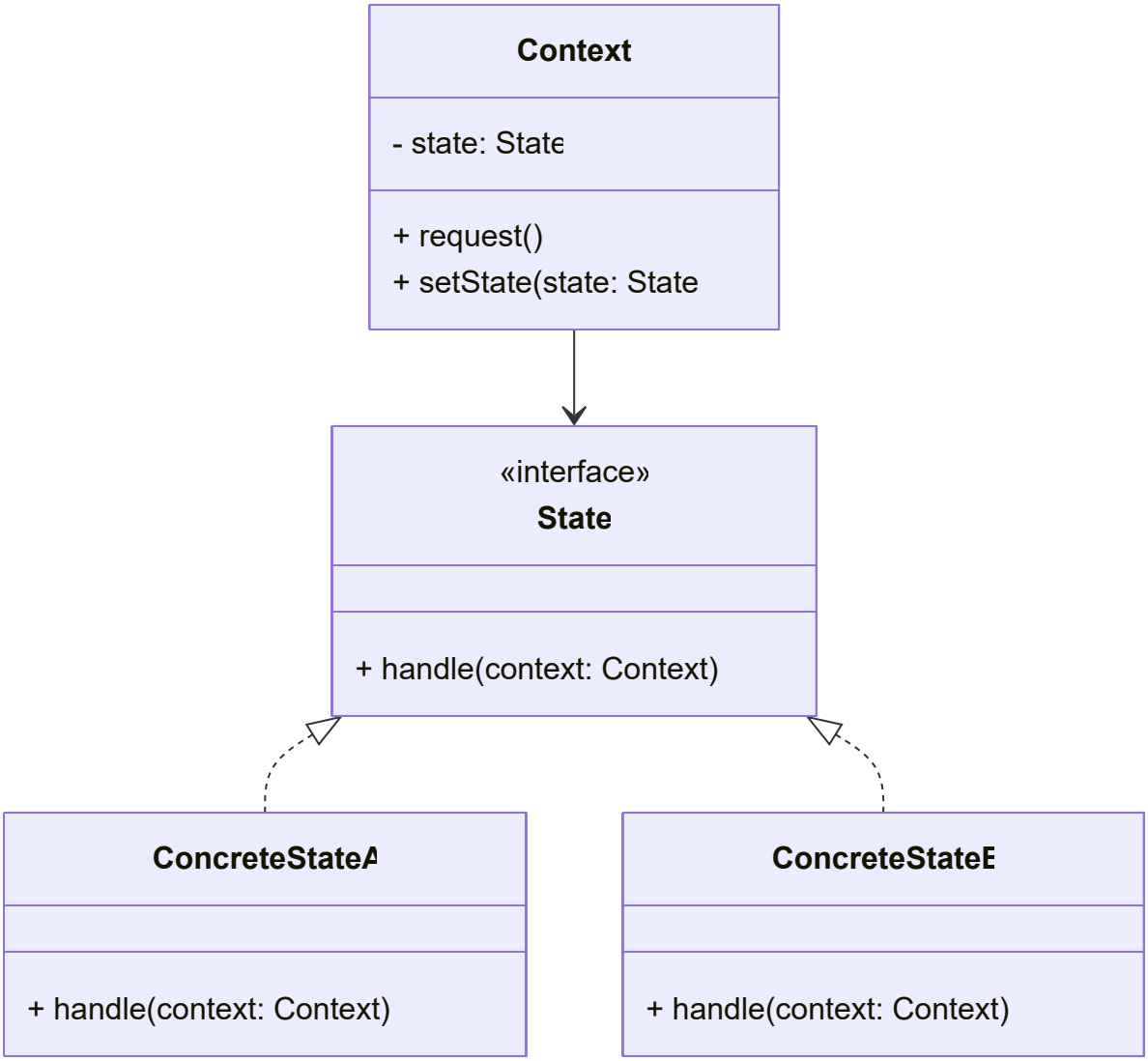
Relazioni:

- Mediator centralizza, Observer distribuisce.
- Spesso usato con MVC.

7. State

Problema:
Permettere a un oggetto di cambiare comportamento al cambiare del suo stato interno.
Esempio: un player audio con stati “play”, “pause”, “stop”.

Struttura UML:



Vantaggi:

- Organizza il codice relativo agli stati.
- Evita grandi statement condizionali.
- Rispetta Open/Closed.

Svantaggi:

- Può introdurre troppe classi per stati semplici.

- Transizioni di stato possono essere complesse.

Relazioni:

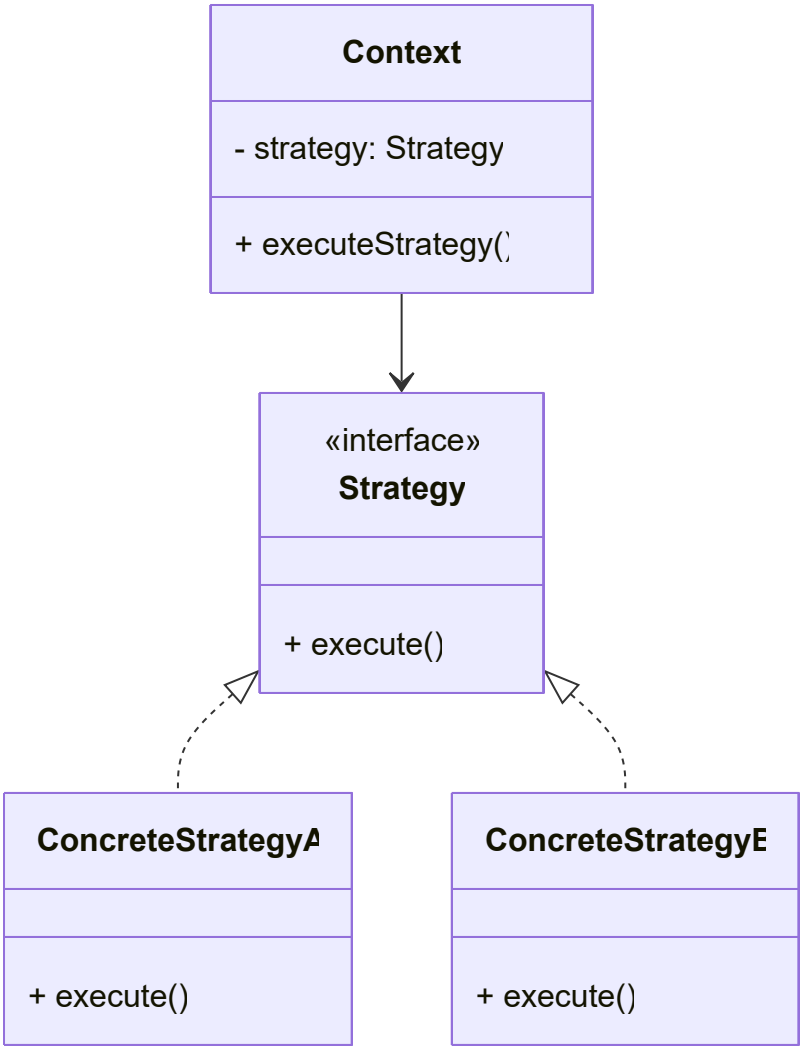
- Simile a Strategy ma cambia automaticamente con lo stato.
- Può essere implementato con Flyweight se stati sono condivisi.

---

8. Strategy

Problema:
Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili.
Esempio: algoritmi di ordinamento (bubble sort, quick sort) selezionabili a runtime.

Struttura UML:



Vantaggi:

- Rende gli algoritmi intercambiabili.

- Elimina grandi blocchi condizionali.
- Rispetta Open/Closed.

Svantaggi:

- Aumenta il numero di classi.
- Il client deve conoscere le strategie.

Relazioni:

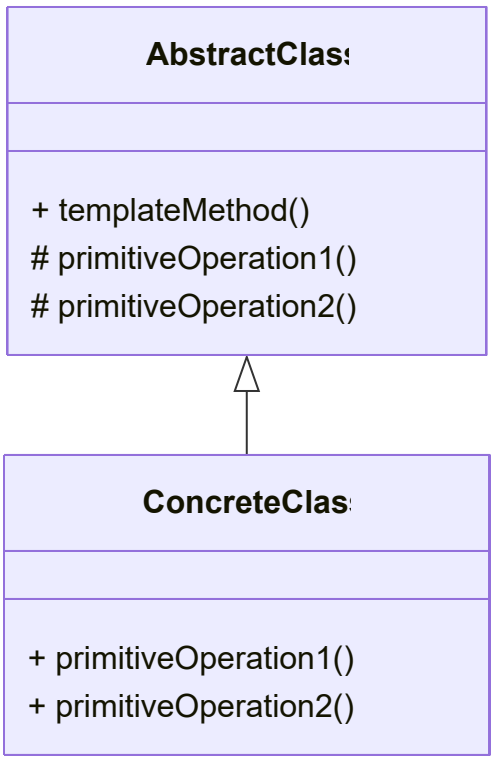
- Simile a State ma con cambio esplicito.
- Command incapsula richieste, Strategy incapsula algoritmi.

---

9. Template Method

Problema:
Definire lo scheletro di un algoritmo in una superclasse, lasciando alle sottoclassi la definizione di alcuni passi.
Esempio: processo di preparazione di una bevanda (the, caffè) con passi comuni.

Struttura UML:



Vantaggi:

- Riusa codice comune.
- Controlla il flusso dell’algoritmo.
- Rispetta Open/Closed.

Svantaggi:

- Può limitare le sottoclassi.
- Difficile comprendere il flusso.

Relazioni:

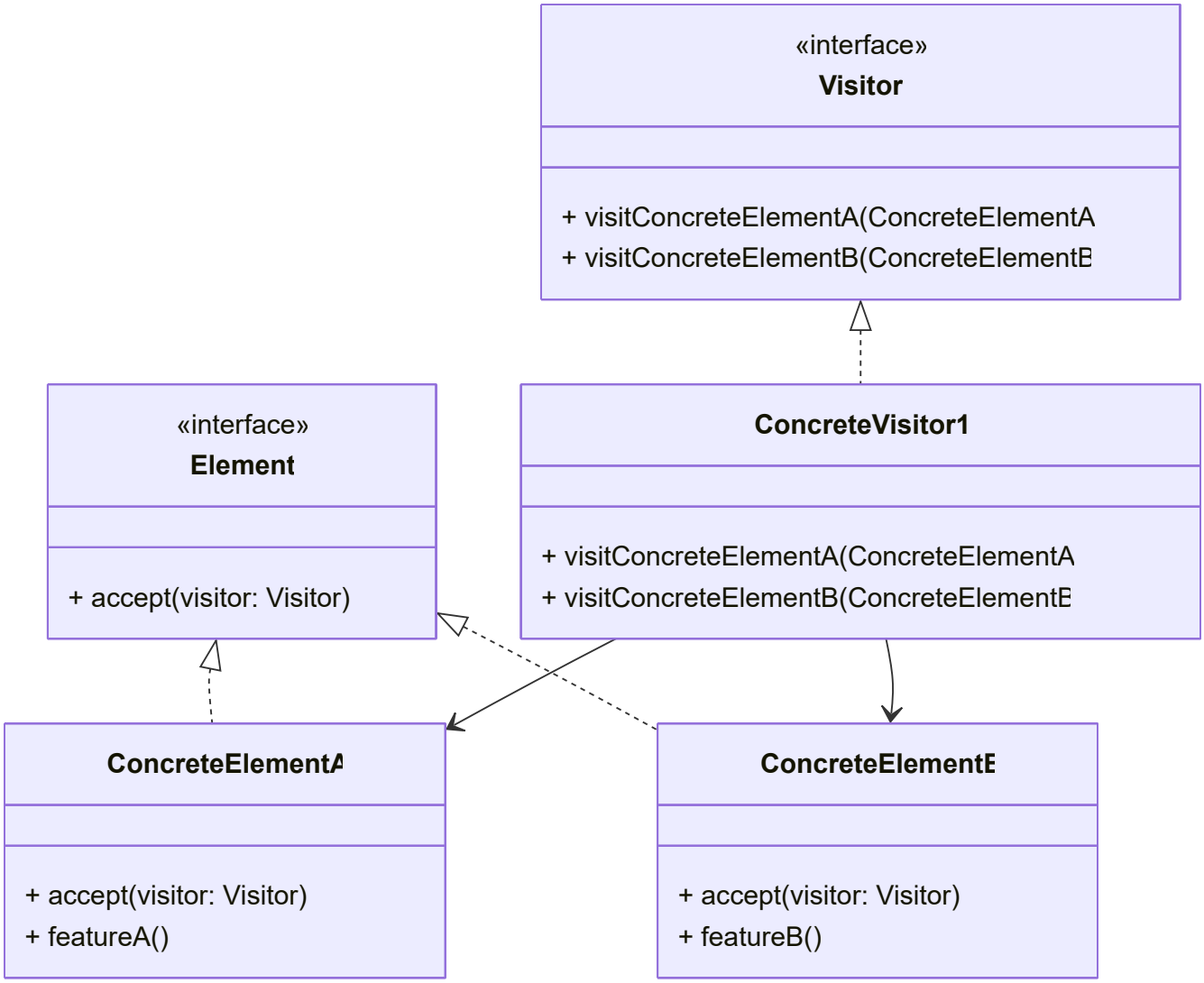
- Factory Method è un caso speciale di Template Method.
- Strategy usa delega, Template Method usa eredità.

---

10. Visitor

Problema:
Separare algoritmi dagli oggetti su cui operano, permettendo di aggiungere nuove operazioni senza modificare le classi.
Esempio: calcolare il costo totale di un carrello della spesa con diversi tipi di prodotti.

Struttura UML:



Vantaggi:

- Aggiunge operazioni senza modificare classi esistenti.
- Riunisce operazioni correlate.
- Rispetta Open/Closed.

Svantaggi:

- Difficile aggiungere nuovi tipi di elementi.
- Può violare l'incapsulamento.

Relazioni:

- Può essere usato con Iterator per attraversare strutture complesse.
- Simile a Command ma per operazioni su più oggetti.
