Informatica e Computazione
Design Pattern

Armando Tacchella

Corso di Laurea in Ingegneria Informatica
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

Sommario

Introduzione ai Design Pattern

"*Each **pattern** describes a **problem** which occurs **over and over again** in our environment, and then describes the **core of the solution** to that problem, in such a way that you can **use this solution a million times over**, **without ever doing it** the same way twice*"

Cristopher Alexander (architetto), 1977

"*Design Patterns capture **solutions** that have developed and **evolved over time**. Hence they aren't the designs people tend to generate **initially**. They reflect **untold redesign and recoding** as developers have struggled for **greater reuse and flexibility** in their software. Design patterns capture these solutions in a **succint and easily applied** form.*"

Gamma, Helm, Johnson, Vlissides (*The Gang of Four*), 1994

Elementi dei Design Pattern nel software

- **Nome**: un appellativo sintetico con cui riferirsi al problema e alla corrispondente soluzione.
- **Problema**: la descrizione del problema e del contesto per il quale il pattern è applicabile.
- **Soluzione**: la descrizione degli elementi che costituiscono il progetto, le loro relazioni, responsabilità e collaborazioni.
- **Conseguenze**: i risultati e i compromessi ottenuti applicando il pattern, utili al fine di valutare soluzioni alternative.

Tipologie di Pattern

- **Creazionali**: introducono astrazioni utili nel processo di istanziazione delle classi (ad es. FACTORY METHOD).
- **Strutturali**: gestiscono la composizione di classi in strutture di dimensioni maggiori (ad es. COMPOSITE).
- **Comportamentali**: gestiscono aspetti algoritmici e le responsabilità condivise tra diverse classi (ad es. ITERATOR).

Pattern Creazionali

Forniscono un'**astrazione del processo di istanziazione** degli oggetti e aiutano a rendere un sistema **indipendente dalle modalità di creazione**, composizione e rappresentazione degli oggetti utilizzati.

- Incapsulano la conoscenza delle classi concrete utilizzate nel sistema.
- Nascondono le modalità di creazione e composizione delle classi del sistema.

Abstract Factory: Scheda riassuntiva

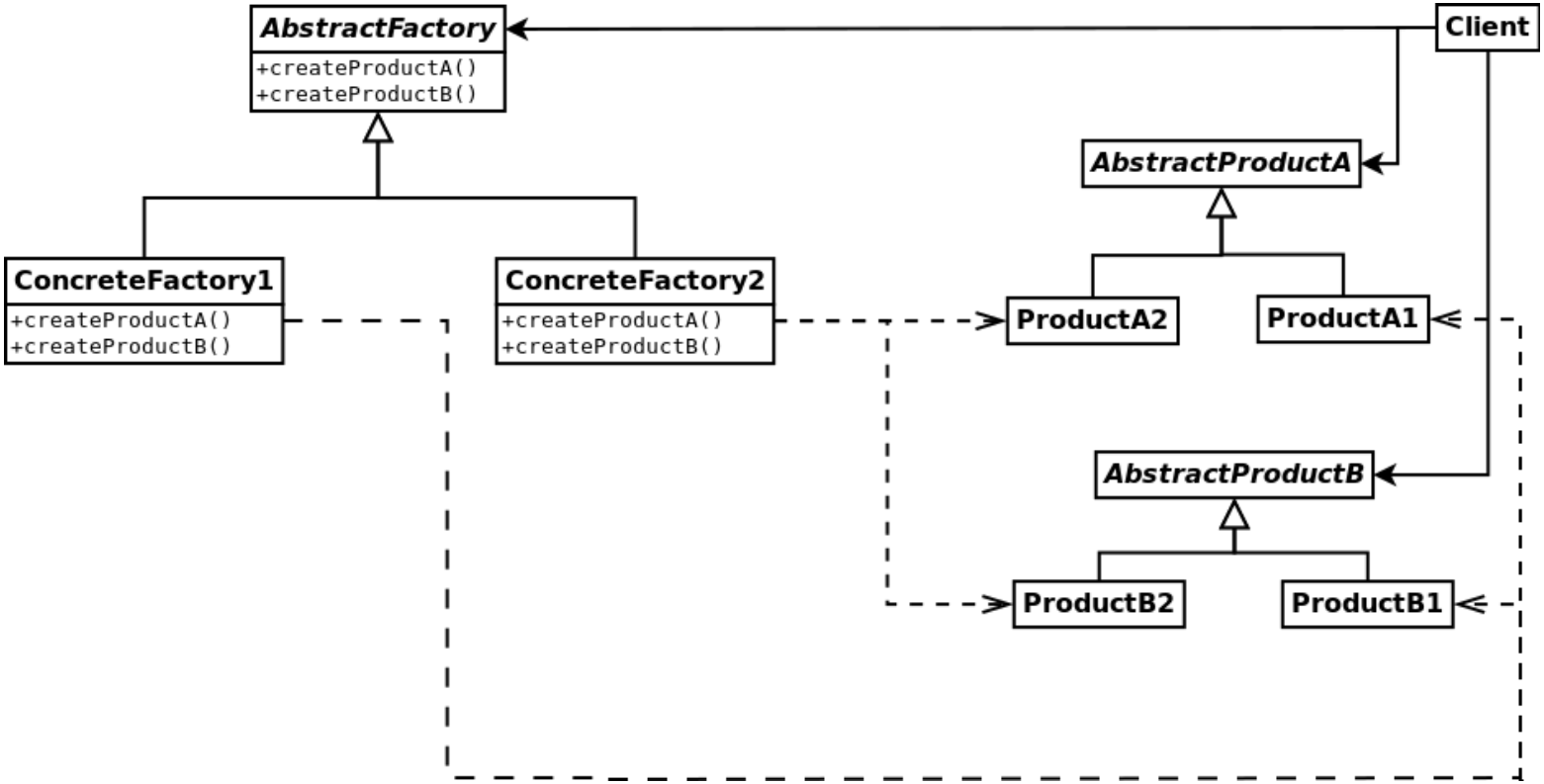
Scopo

Fornire un'**interfaccia** per la creazione di famiglie di **oggetti correlati o dipendenti** senza specificare quali siano le loro classi **concrete**.

Applicabilità

- Un sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti.
- Esistono famiglie di oggetti correlati, progettati per essere utlizzati insieme e occorre garantire il rispetto di questo vincolo.
- Un sistema deve essere configurato scegliendo fra più famiglie di prodotti.

Abstract Factory: Struttura



Abstract Factory: Partecipanti

- **AbstractFactory** dichiara un'interfaccia per le operazioni di creazione di oggetti prodotto astratti.
- **ConcreteFactory** implementa le operazioni di creazione degli oggetti prodotto concreti.
- **AbstractProduct** dichiara un'interfacccia per una tipologia di oggetti prodotto.
- **ConcreteProduct** definisce un oggetto prodotto che dovrà essere creato dalla factory concreta e implementa `AbstractProduct`.
- **Client** Utilizza soltanto le interfacce dichiarate dalle classi `AbstractFactory` e `AbstractProduct`.

Builder: Scheda riassuntiva

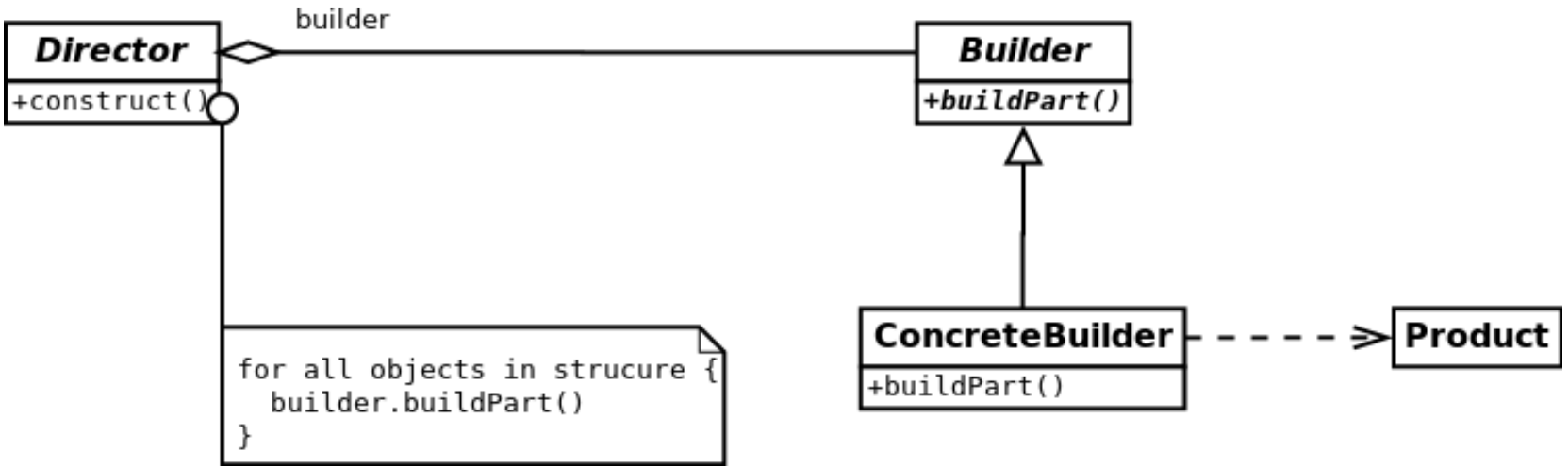
Scopo

Separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa essere utilizzato per creare rappresentazioni diverse.

Applicabilità

- L'algoritmo per la creazione di un oggetto complesso deve essre mantenuto separato e indipendente dalle parti di cui l'oggetto è composto e dal modo con cui queste sono assemblate.
- Il processo di costruzione deve consentire rappresentazioni differenti per l'oggetto da costruire.

Builder: Struttura



Builder: Partecipanti

- **Builder** specifica un'interfaccia astratta per la creazione di parti di un oggetto prodotto.
- **ConcreteBuilder**
  - Costruisce e assembla le parti del prodotto attraverso l'implementazione dell'interfaccia `Builder`.
  - Definisce e tiene traccia delle rappresentazioni di `Product` create.
  - Fornisce un'interfaccia per ottenere l'oggetto che rappresenta il risultato dell'attività di costruzione.
- **Director** Costruisce un oggetto utilizzando l'interfaccia Builder.
- **Product**
  - Rappresenta l'oggetto complesso in costruzione. `ConcreteBuilder` costruisce la rappresentazione interna di `Product` e definisce il processo attraverso il quale il prodotto viene assemblato.
  - Comprende le classi che definiscono le singole parti che costituiscono il prodotto e le interfacce per assemblare le parti al fine di ottenere il risultato finale.

Factory Method: Scheda riassuntiva

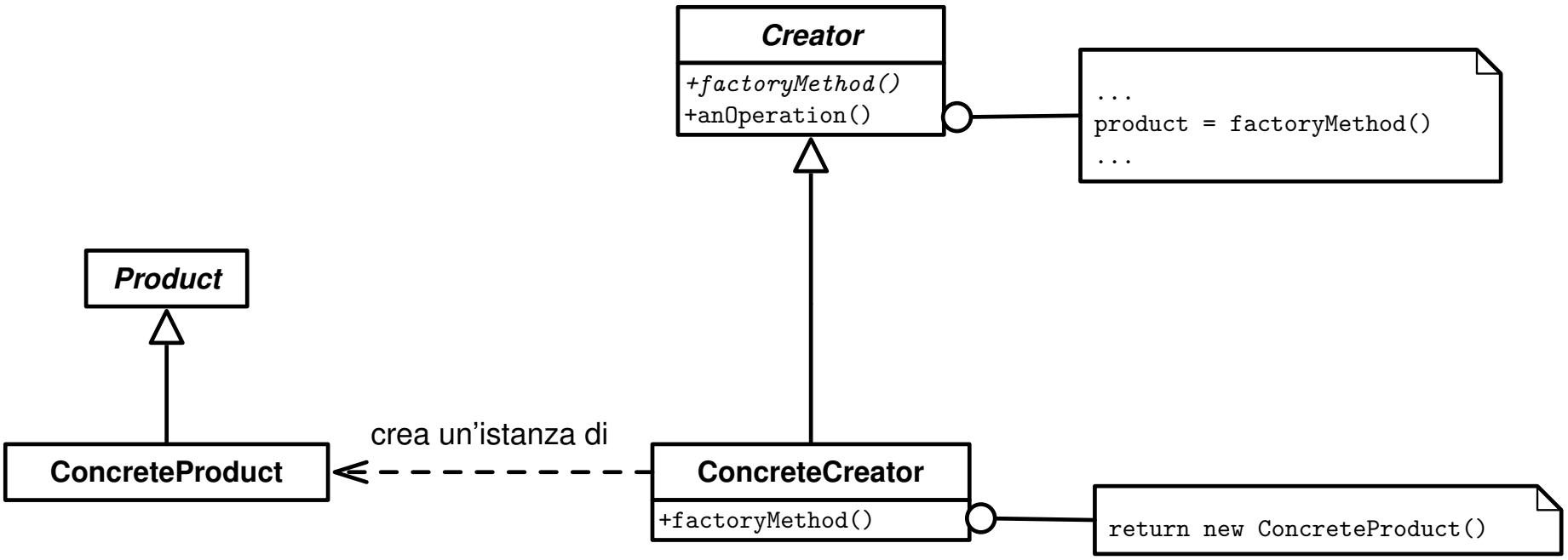
Scopo

Definisce un'**interfaccia per la creazione di un oggetto**, lasciando alle sottoclassi la decisione sulla classe che deve essere istanziata. Il pattern Factory Method consente di **deferire l'istanziazione** di una classe alle sottoclassi.

Applicabilità

- Una classe non è in grado di sapere **in anticipo** le classi degli oggetti che deve creare.
- Una classe vuole che le sue sottoclassi scelgano gli oggetti da creare.
- Le classi **delegano la responsabilità** a una o più classi di supporto e si vuole localizzare in un punto ben preciso la conoscenza di quale o quali classi di supporto vengano delegate.

Factory Method: Struttura



Factory Method: Partecipanti

- **Product** definisce l'interfaccia degli oggetti creati dal metodo factory.
- **ConcreteProduct** implementa l'interfaccia `Product`.
- **Creator** dichiara il metodo factory, che restituisce un oggetto di tipo `Product`; può anche definire un'implementazione standard di `factoryMethod` per restituire un oggetto `ConcreteProduct`.
- **ConcreteCreator** ridefinisce `factoryMethod` in modo da restituire un'istanza di uno specifico `ConcreteProduct`.

Prototype: Scheda riassuntiva

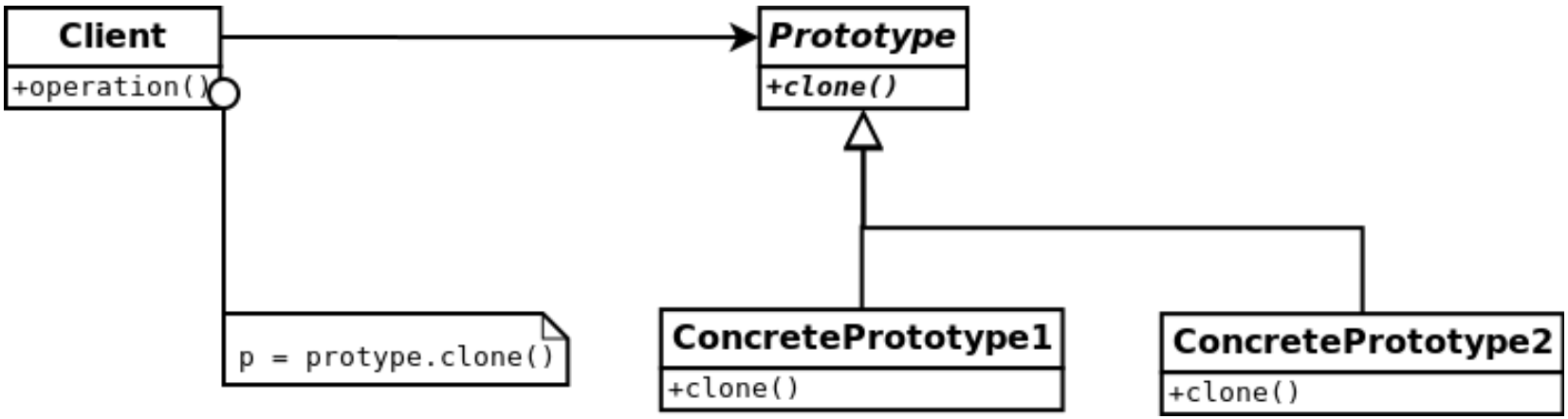
Scopo

Specificare la tipologia di oggetti da creare utilizzando un'istanza prototipo e creare nuovi oggetti copiando questo prototipo.

Applicabilità

- Le classi istanziate sono note solo in fase di esecuzione.
- Evitare di costruire una gerararchia di classi factory parallela alla gerarchia dei prodotti.
- Le istanze di una classe possono avere un numero ridotto di combinazioni di stati. In questo caso potrebbe essere conveniente installare un corrispondente numero di prototipi da clonare, piuttosto che istanziare le classi manualmente, ogni volta con uno stato particolare.

Prototype: Struttura



Prototype: Partecipanti

- **Prototype** dichiara un'interfaccia per consentire la propria clonazione.
- **ConcretePrototype** implementa l'operazione di clonazione.
- **Client** Crea un nuovo oggetto chiedendo ad un suo prototipo di clonarsi.

Singleton: Scheda riassuntiva

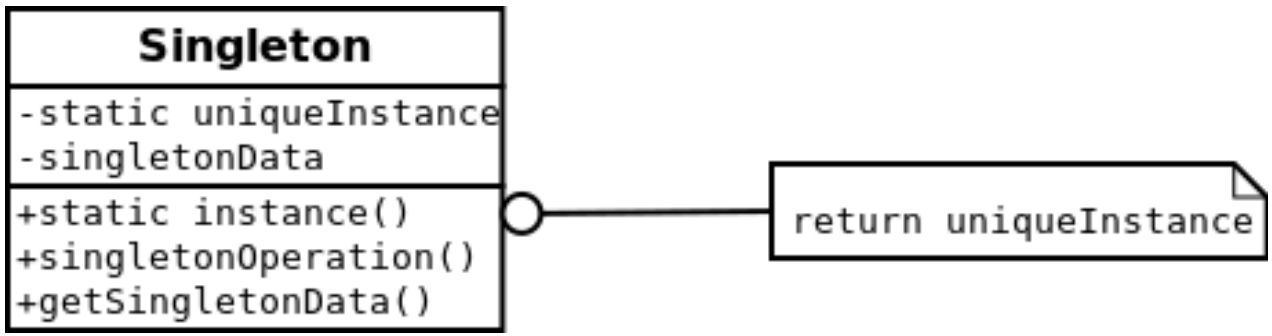
Scopo

Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale a tale istanza.

Applicabilità

- Deve esistere esattamente un'istanza di una classe e tale istanza deve essere resa accessibile ai client attraverso un punto di accesso noto a tutti gli utilizzatori.
- L'unica istanza deve poter essere estesa attraverso la definizione di sottoclassi e i client devono essere in grado di utilizzare le istanze estese senza dover modificare il proprio codice.

Singleton: Struttura



Singleton: Partecipanti

- **Singleton** definisce un'operazione `instance` che consente ai client di accedere all'unica istanza esistente della classe.

Pattern Strutturali

Riguardano le modalità con cui le classi e gli oggetti si **combinano** per formare strutture di maggiori dimensioni.

- Utilizzano l'ereditarietà per **comporre interfacce e implementazioni**.
- Compongono oggetti al fine di realizzare **nuove funzionalità**.

Adapter: Scheda riassuntiva

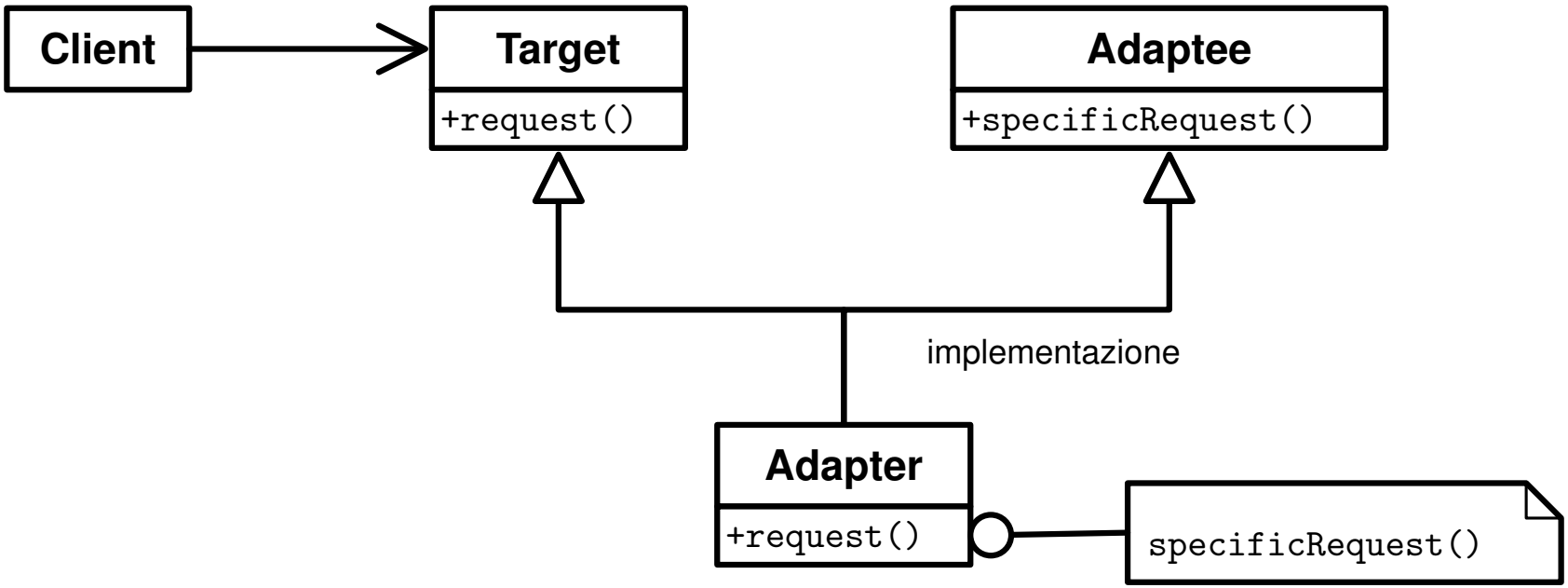
Scopo

Convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client. Adapter consente a classi diverse di operare insieme quando ciò non sarebbe altrimenti possibile a causa di **interfacce incompatibili**.

Applicabilità

- Si vuole usare una classe esistente ma la sua interfaccia **non è compatibile** con quella desiderata.
- Si vuole creare una classe **riusabile** in grado di cooperare con classi non correlate o impreviste, cioè con classi che non necessariamente hanno interfacce compatibili.
- Si devono utilizzare diverse sottoclassi esistenti ma non è pratico **adattare la loro interfaccia** creando una sottoclasse per ciascuna di esse.

Adapter: Struttura



Nota: questo pattern è specificato con una **ereditarietà multipla**. In Java, è necessario introdurre una **implementazione di interfaccia**.

Adapter: Partecipanti

- **Target** definisce l'interfaccia specifica del dominio utilizzata dal client.
- **Client** collabora con oggetti compatibili con l'interfaccia `Target`.
- **Adaptee** Individua un'interfaccia esistente che deve essere adattata.
- **Adapter** Adatta l'interfaccia di `Adaptee` all'interfaccia `Target`.

Bridge: Scheda riassuntiva

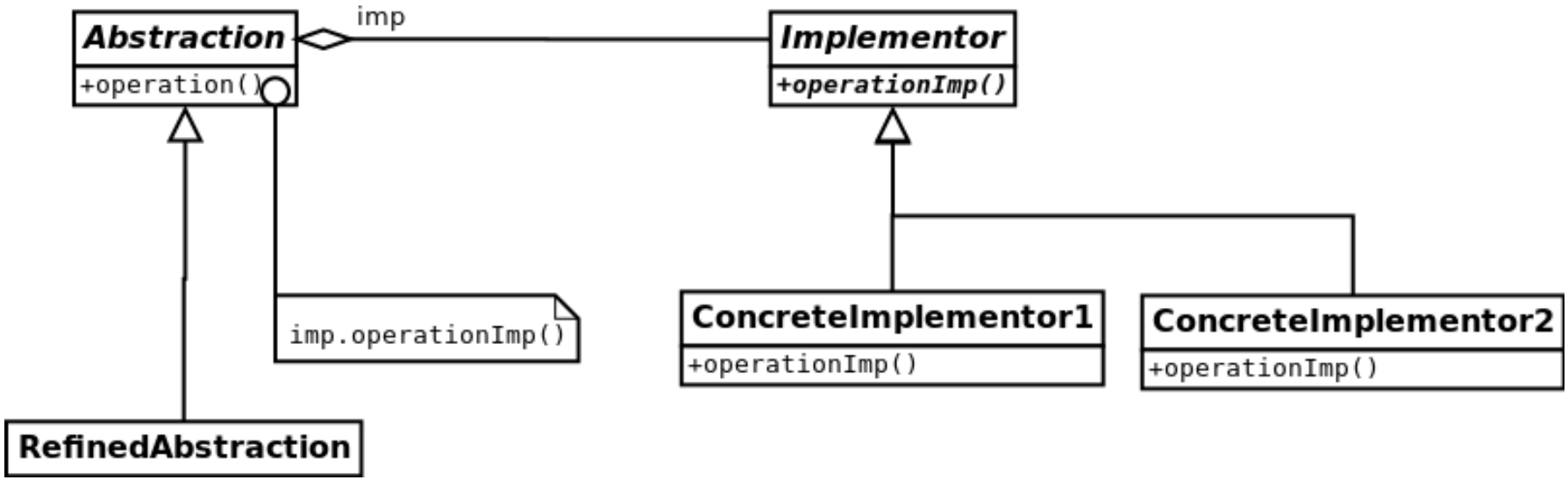
Scopo

Disaccoppia un'astrazione dalla sua implementazione in modo che le due possano variare indipendentemente.

Applicabilità

- Si vuole evitare un legamen permanente tra un'astrazione e la sua implementazione.
- Si vuole avere la possibilità di estendere sia le astrazioni sia le implementazioni introducendo sottoclassi.
- I cambiamenti dell'implementazione di un'astrazione non devono avere impatto sui client, ossia non si vuole ricompilare il codice dei client a seguito di cambiamenti nell'implementazione.
- Si vuole condividere una stessa implementazione fra più oggetti.

Bridge: Struttura



Bridge: Partecipanti

- **Abstraction** definisce l'interfaccia dell'astrazione e mantiene un riferimento ad un oggetto di tipo `Implementor`.
- **RefinedAbstraction** estende l'interfaccia definita da `Abstraction`.
- **Implementor** Definisce l'interfaccia per le classi che implementano l'astrazione. Questa interfaccia non deve corrispondere esattamente all'interfaccia di `Abstraction`.
- **ConcreteImplementor** Fornisce l'interfaccia `Implementor` e definisce la sua realizzazione concreta.

Composite: Scheda riassuntiva

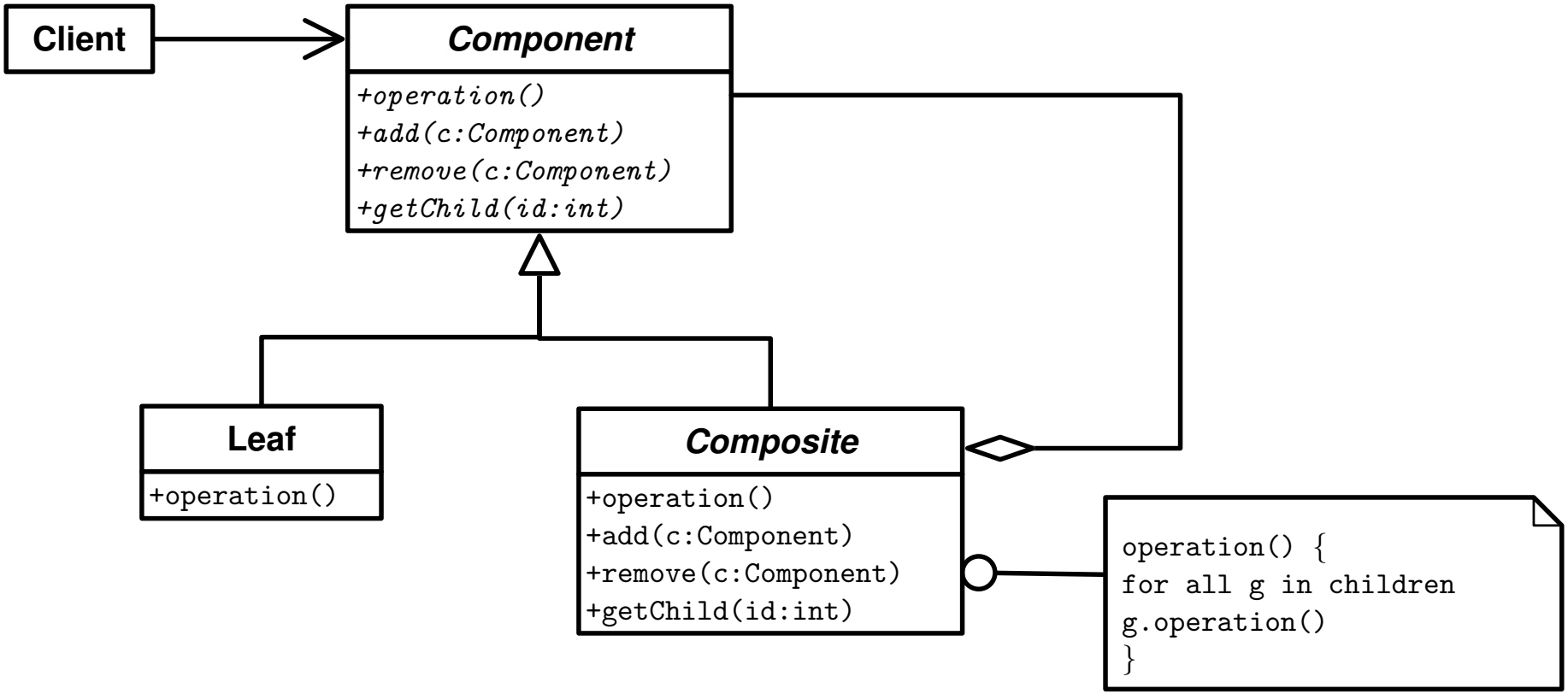
Scopo

Comporre oggetti in **strutture ad albero** per rappresentare gerarchie **tutto-parte**; Composite consente ai client di gestire oggetti individuali e composizioni di oggetti **in modo uniforme**.

Applicabilità

- Si desidera rappresentare una **gerarchia tutto-parte**.
- Si desidera che i client possano ignorare la differenza tra **composizioni** di oggetti e oggetti **individuali**
- I client devono poter trattare gerarchie e oggetti individuali **in modo uniforme**.

Composite: Struttura



Composite: Partecipanti

- **Component** dichiara l'interfaccia degli oggetti nella composizione, implementa i meccanismi comuni dell'interfaccia, dichiara un'interfaccia per gestire le sue parti.
- **Leaf** rappresenta gli oggetti elementari nella composizione, definisce i meccanismi per gli oggetti elementari.
- **Composite** definisce i meccanismi per i componenti composti da più parti, aggrega le parti e implementa le operazioni legate alle parti definite nell'interfaccia di `Component`.
- **Client** Manipola gli oggetti composti utilizzando l'interfaccia definita in `Component`.

Decorator: Scheda riassuntiva

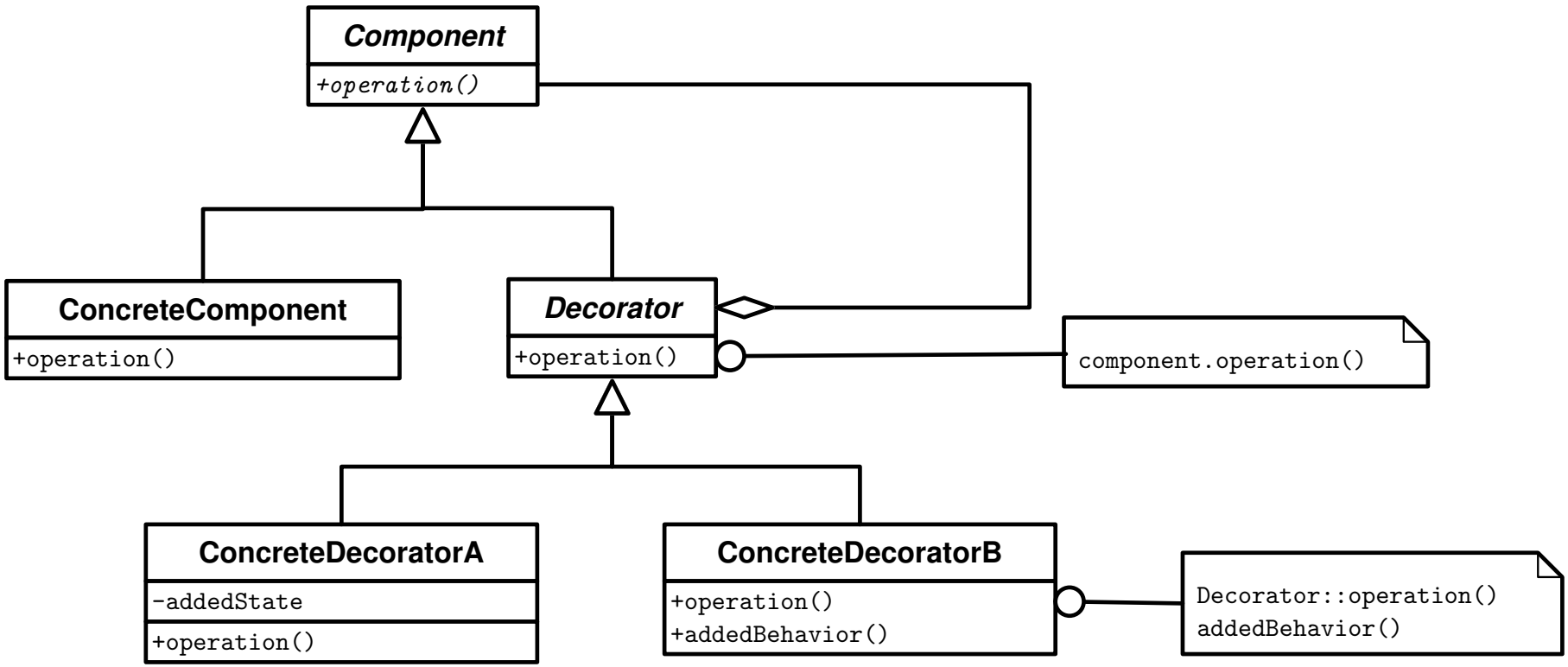
Scopo

Aggiungere responsabilità ad un oggetto **in modo dinamico**, in modo da fornire un'alternativa flessibile all'ereditarietà

Applicabilità

- Si desidera **aggiungere responsabilità** ad oggetti specifici in modo dinamico e trasparente, **senza influenzare** altri oggetti.
- Si vuole avere la possibilità di **rimuovere le responsabilità aggiunte** in un secondo tempo.
- L'estensione tramite classi derivate **non è praticabile** (ad es. troppe combinazioni di estensioni indipendenti).

Decorator: Struttura



Decorator: Partecipanti

- **Component** definisce l'interfaccia per gli oggetti a cui è possibile agigungere responsabilità dinamicamente.
- **ConcreteComponent** definisce l'oggetto concreto a cui aggiungere responsabilità.
- **Decorator** gestisce un riferimento all'oggetto **Component** e definisce un'interfaccia ad esso conforme.
- **ConcreteDecorator** realizza le responsabilità aggiuntive da aggiungere al componente.

Pattern Comportamentali

Descrivono non solo **schemi di relazioni** tra oggetti, ma anche **modalità di comunicazione** tra oggetti; queste modalità sono proprie di flussi di controllo **non elementari**, e i pattern comportamentali consentono di focalizzare l'analisi sull'interconnessione tra oggetti piuttosto che sulle complessità del controllo

- Utilizzo dell'ereditarietà per **distribuire comportamenti** tra diverse classi.
- Utilizzo della composizione per **creare cooperazione** tra oggetti diversi.

Command: Scheda riassuntiva

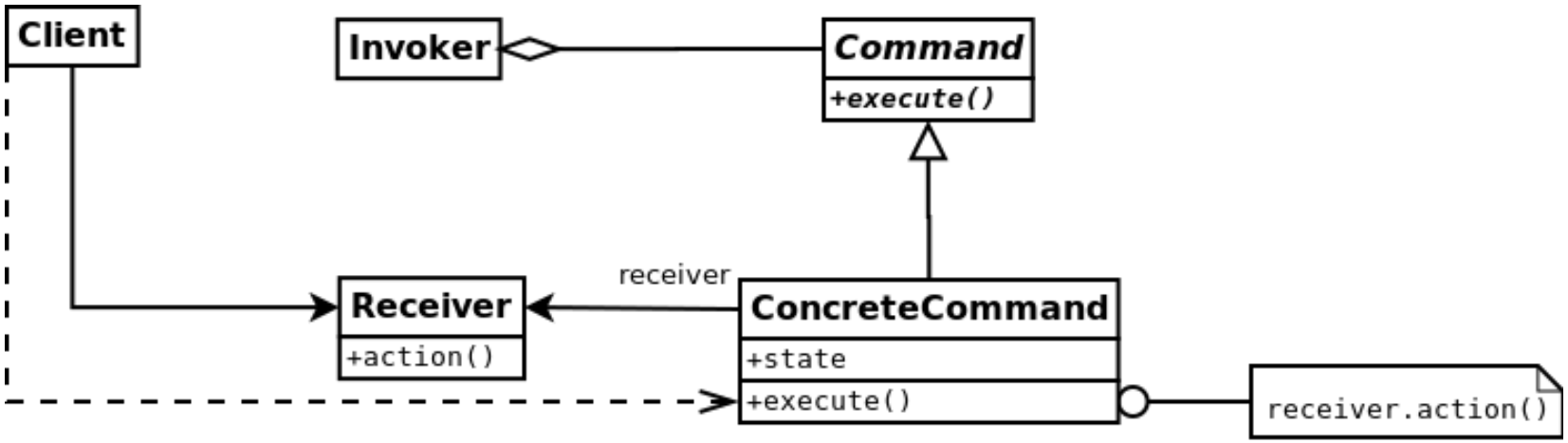
Scopo

Incapsulare una **richiesta** in un **oggetto** per gestire le **richieste come dati** e supportare una varietà di operazioni.

Applicabilità

- Parametrizzare oggetti sulla base di un'azione da compiere.
- Specificare, accodare ed eseguire richieste in tempi diversi.
- Supportare operazioni di "undo".

Command: Struttura



Command: Partecipanti

- **Command** dichiara un'interfaccia per eseguire un'operazione.
- **ConcreteCommand** implementa il comando.
- **Client** crea un `ConcreteCommand` e ne imposta il `Receiver`
- **Invoker** richiede al comando di eseguire l'azione.
- **Receiver** riceve il comando e svolge effettivamente l'azione.

Interpreter: Scheda riassuntiva

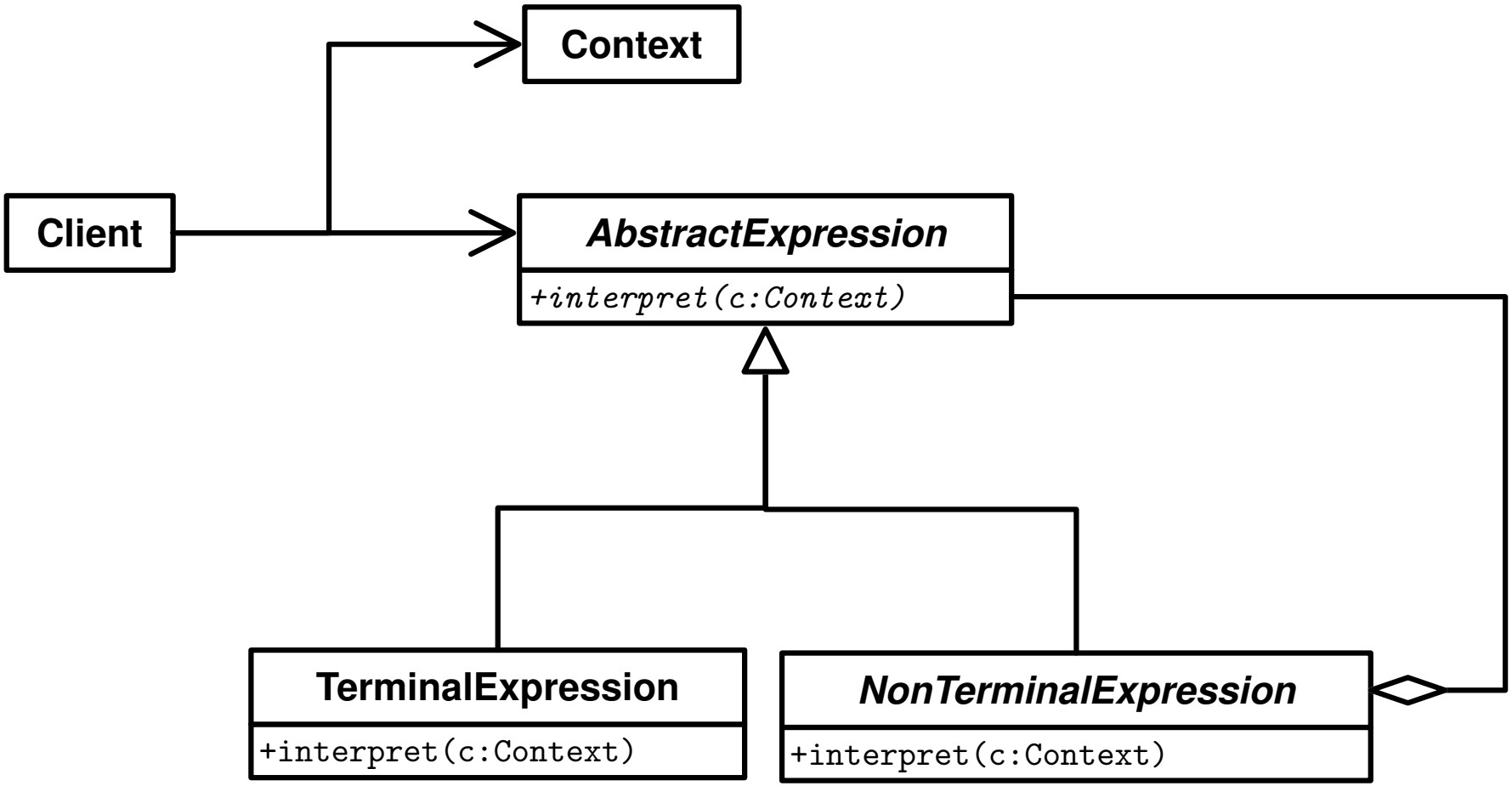
Scopo

Dato un **linguaggio formale**, definire una rappresentazione per la sua **grammatica** insieme ad un **interprete** che utilizza tale rappresentazione per **valutare** frasi del linguaggio.

Applicabilità

- La grammatica ha una **struttura semplice**; per grammatiche complesse, la gerarchia delle classi diventa difficile da gestire ed è meglio utilizzare **generatori automatici di parser**.
- L'efficienza **non è** un aspetto principale, in quanto vi possono essere casi particolari in cui è possibile ottenere valutazioni più efficienti con metodi specifici.

Interpreter: Struttura



Interpreter: Partecipanti

- **AbstractExpression** dichiara un'operazione astratta di interpretazione che è comune a tutti i nodi dell'albero sintattico.
- **TerminalExpression** implementa un'operazione di valutazione relativa ai **simboli terminali** della grammatica; ne è richiesta un'istanza diversa per ogni simbolo terminale della grammatica.
- **NonterminalExpression** implementa un'operazione di valutazione relativa ai **simboli nonterminali** della grammatica; di norma, è richiesta una classe diversa per ogni regola della grammatica.
- **Context** contiene informazioni globali relative all'interprete.
- **Client** costruisce l'albero sintattico e invoca l'operazione di interpretazione.

Iterator: Scheda riassuntiva

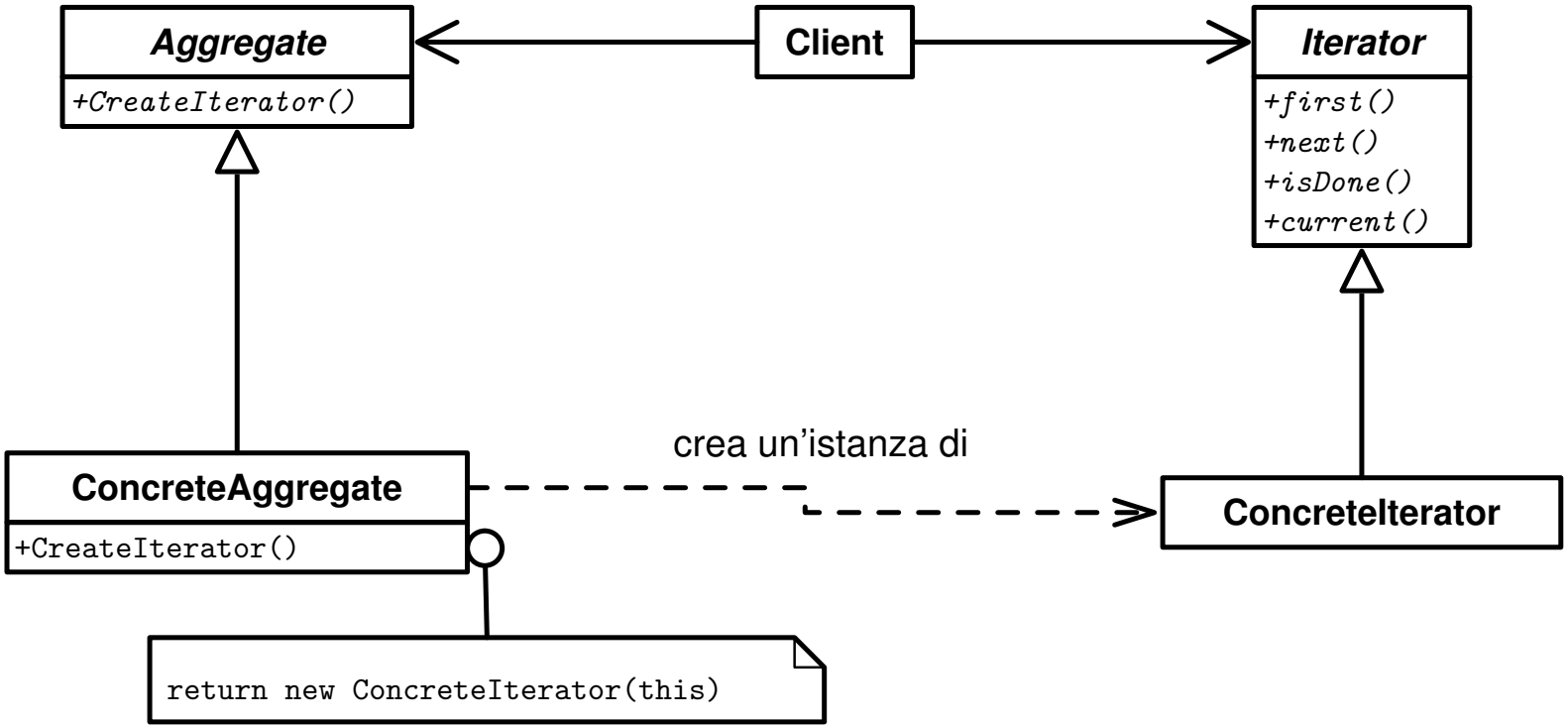
Scopo

Fornire un modo per accedere **sequenzialmente** agli elementi di un contenitore senza esporne la rappresentazione sottostante.

Applicabilità

- Necessità di **mantenere nascosta** la struttura interna del contenitore, consentendone l'iterazione dall'esterno.
- Necessità di eseguire iterazioni multiple.
- Possibilità di supportare l'iterazione in modo consistente su contenitori di tipo diverso (**iterazione polimorfa**).

Iterator: Struttura



Iterator: Partecipanti

- **Iterator** definisce l'interfaccia per accedere agli elementi in sequenza.
- **ConcreteIterator** implementa l'interfaccia dell'iteratore e tiene traccia della posizione durante la scansione del contenitore.
- **Aggregate** il contenitore che definisce un'interfaccia per la creazione di un iteratore.
- **ConcreteAggregate** implementa la creazione dell'iteratore e restituisce un'istanza di **ConcreteIterator**.

Observer: Scheda riassuntiva

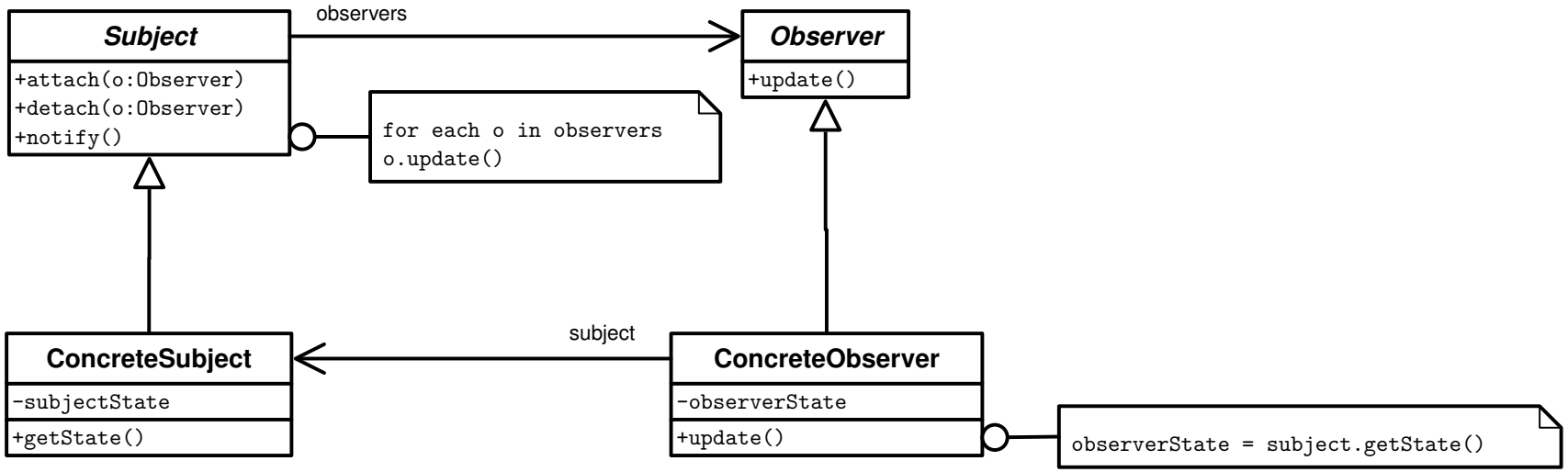
Scopo

Definire una **dipendenza uno-a-molti** tra oggetti in modo tale che, quando un oggetto cambia il suo **stato**, tutti quelli ad esso collegati vengono notificati e aggiornati **automaticamente**.

Applicabilità

- Un'astrazione ha due aspetti, uno dipendente dall'altro. Incapsulare questi aspetti in oggetti separati consente di variarli e riutilizzarli **indipendentemente**.
- Quando un cambiamento allo stato di un oggetto richiede il cambiamento allo stato di altri, e non si conosce in anticipo **quanti oggetti** debbano essere modificati.
- Quando un oggetto deve essere in grado di notificare altri oggetti **senza conoscere** come questi ultimi sono realizzati.

Observer: Struttura



Observer: Partecipanti

- **Observer** Definisce un'interfaccia per l'aggiornamento degli oggetti che devono essere notificati dei cambiamenti di stato del `Subject`.
- **Subject** Tiene traccia dei propri `Observer` e fornisce un'interfaccia per (s)collegare oggetti di questo tipo.
- **ConcreteObserver** Mantiene un riferimento all'oggetto osservato e contiene informazioni che sono mantenute coerenti con il suo stato.
- **ConcreteSubject** Invia notifiche agli oggetti osservatori quando il suo stato cambia.

State: Scheda riassuntiva

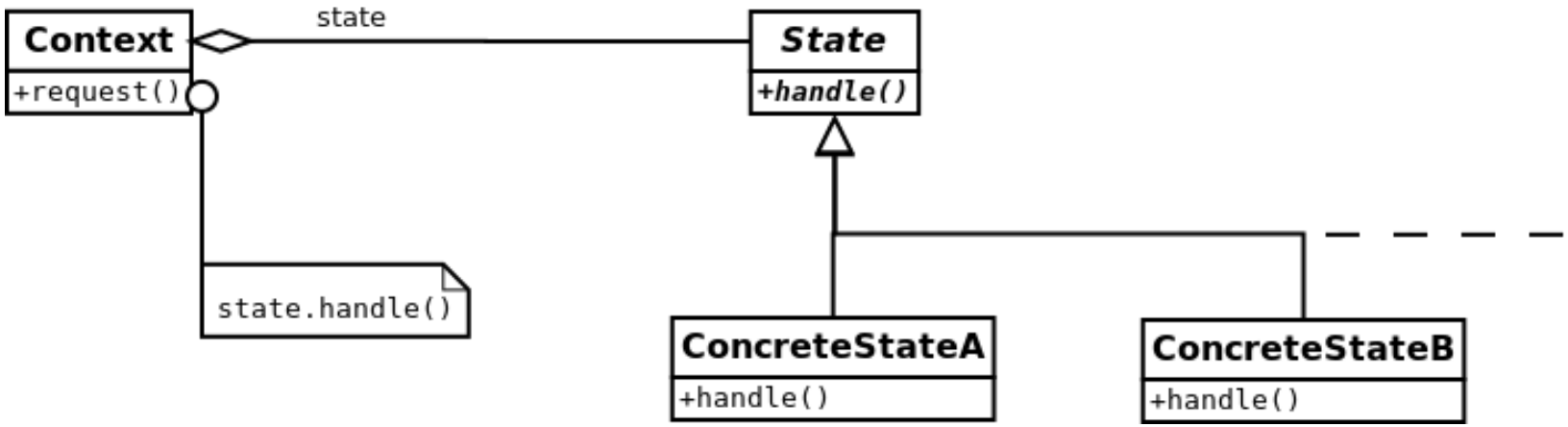
Scopo

Consentire ad un oggetto di **alterare** il suo comportamento quando lo stato interno cambia.

Applicabilità

- Il **comportamento** di un oggetto dipende dal suo **stato**.
- Le operazioni da svolgere consistono di **istruzioni condizionali** molto grandi che dipendono dallo stato dell'oggetto; lo stato è rappresentato da una o più **costanti enumerative**.

State: Struttura



State: Partecipanti

- **Context** Definisce l'interfaccia verso i client e mantiene un'istanza di `ConcreteState` che definisce lo stato corrente.
- **State** Definisce l'interfaccia per incapsulare il comportamento associato con uno stato particolare del contesto.
- **ConcreteState** Sottoclassi che definiscono il comportamento associato ad un particolare stato del contesto.

Strategy: Scheda riassuntiva

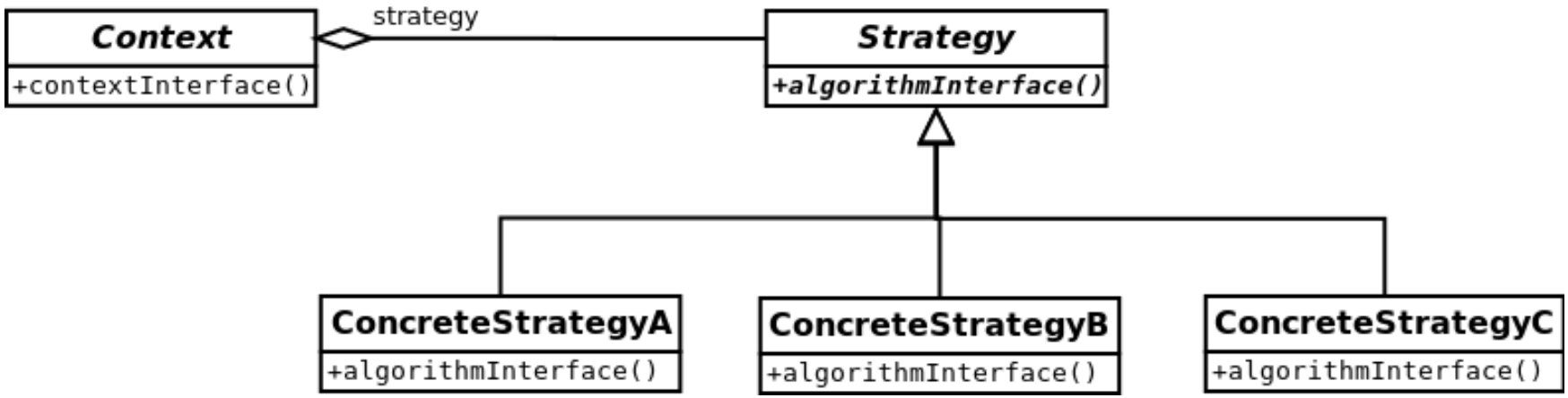
Scopo

Definire una **famiglia di algoritmi**, incapsulare ognuno di essi e renderli **intercambiabili**. L'obiettivo è quello di utilizzare diverse strategie in modo **trasparente**.

Applicabilità

- Vi sono diverse classi in relazione tra di loro che differiscono **solo** per il comportamento.
- Sono necessarie diverse **varianti** di un algoritmo che realizzano strategie efficaci in contesti diversi.
- Un algoritmo utilizza strutture dati di cui gli utilizzatori non devono essere a conoscenza.
- Una classe definisce molti comportamenti che possono essere utilmente isolati come strategie a sè stanti.

Strategy


Context
+contextInterface()
strategy
Strategy
+algorithmInterface()
ConcreteStrategyA
+algorithmInterface()
ConcreteStrategyB
+algorithmInterface()
ConcreteStrategyC
+algorithmInterface()

Strategy: Partecipanti

- **Strategy** Dichiara un'**interfaccia comune** a tutti gli algoritmi supportati. `Context` usa questa interfaccia per chiamare gli algoritmi definiti negli oggetti `ConcreteStrategy`.
- **ConcreteStrategy** Realizza gli algoritmi definiti dall'inteerfaccia `Strategy`.
- **Context** È configurato con un oggetto `ConcreteStrategy`, mantiene un riferimento a `Strategy` e può definire una **interfaccia** per consentire ad oggetti `Strategy` l'accesso ai dati.

Template Method: Scheda riassuntiva

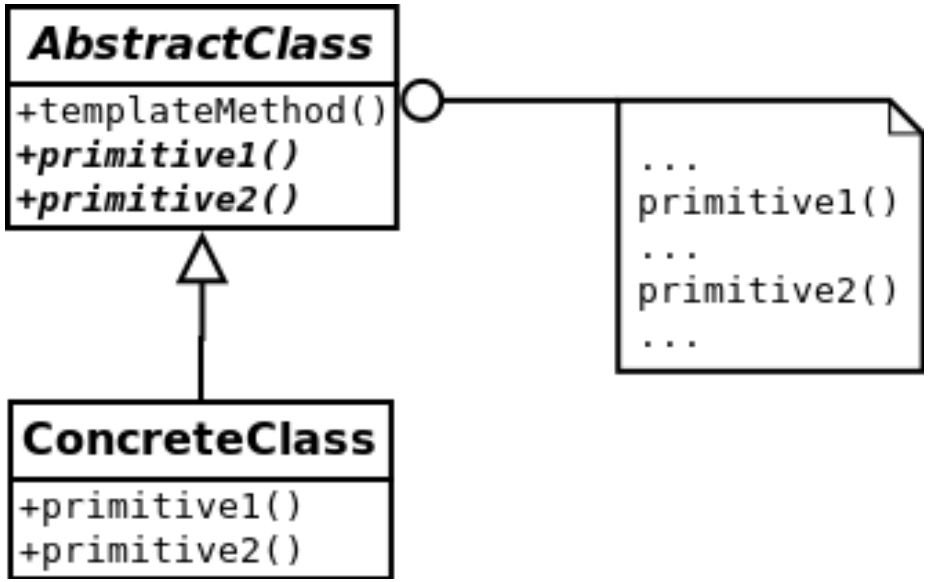
Scopo

Definire lo **scheletro** dell'algoritmo di una operazione, lasciando i particolari alle sottoclassi. Le sottoclassi possono **ridefinire** alcuni passi dell'algoritmo senza cambiarne la struttura.

Applicabilità

- Realizzare le parti **invarianti** di un algoritmo in modo stabile e lasciare alle sottoclassi le parti **varianti**.
- **Fattorizzare** codice per diminuire la duplicazione.
- Controllare le modalità di estensione delle sottoclassi, che possono intervenire **solo** su specifici "hook".

Template Method



Template Method: Partecipanti

- **AbstractClass** Definisce operazioni **primitive astratte** che verranno ridefinite dalle sottoclassi e realizza lo scheletro di un algoritmo.
- **ConcreteClass** Implementa gli spefici passi dell'algoritmo richiesti da `AbstractClass`.

Visitor: Scheda riassuntiva

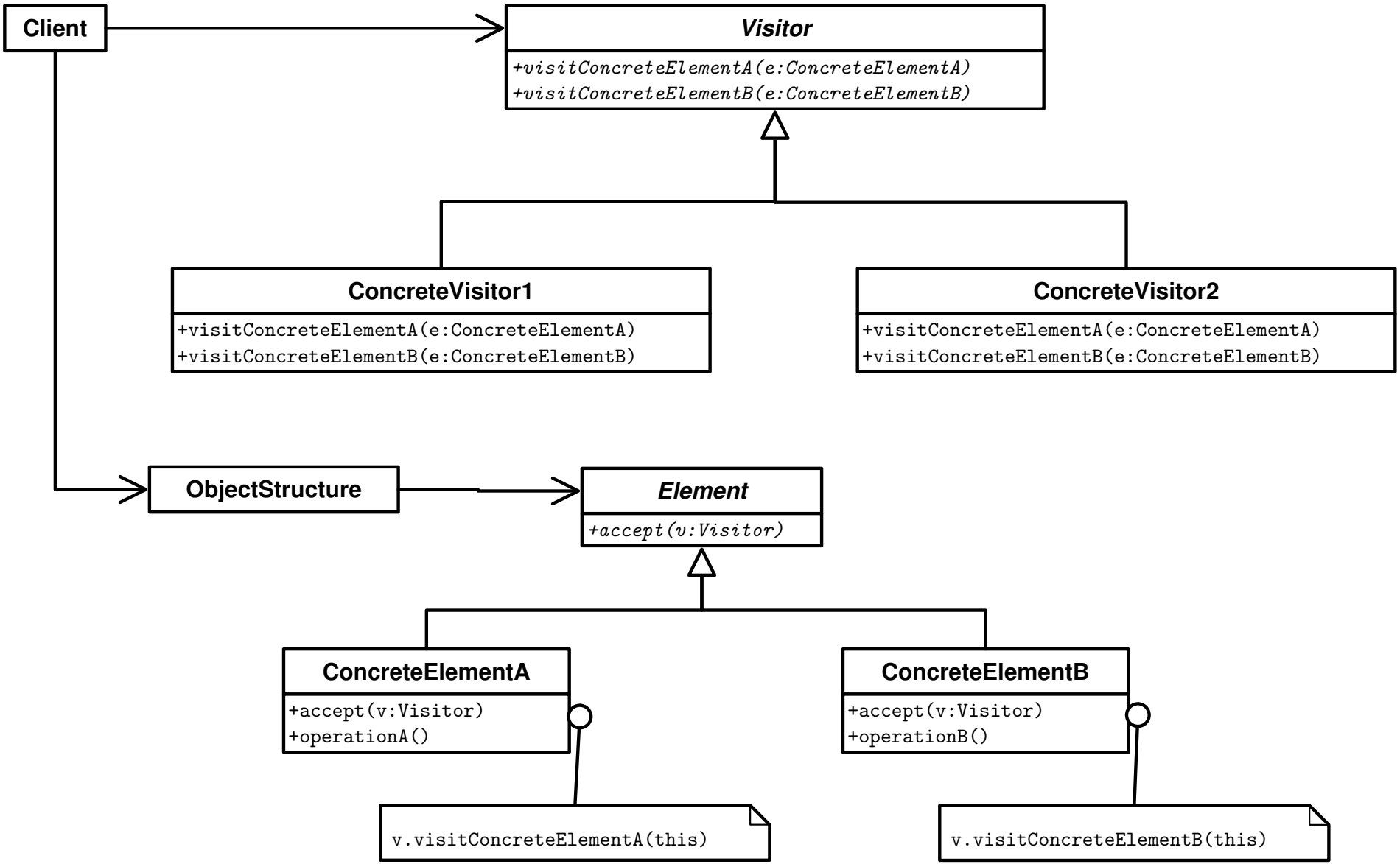
Scopo

Rappresentare **un'operazione** da eseguire sugli elementi di una struttura, mantenendo la possibilità di definire **una nuova operazione** senza modificare le classi degli elementi su cui opera.

Applicabilità

- Un oggetto è composto da **diverse classi** di oggetti con **diverse interfacce**, e si devono implementare operazioni che dipendono dalle **classi concrete**.
- Operazioni **distinte e scorrelate** devono essere eseguite su oggetti di una struttura senza "inquinare" le relative definizioni di classi con queste operazioni.
- Le classi che definiscono la struttura non sono soggette a **frequenti modifiche**.

Visitor: Struttura



Visitor: Partecipanti

- **Visitor** dichiara un'operazione di visita per ogni classe concreta nella struttura da visitare.
- **ConcreteVisitor** implementa ogni operazione dichiarata da `Visitor`.
- **Element** definisce un'operazione di accettazione del visitor.
- **ConcreteElement** implementa l'interfaccia definita da **Element**.
- **ObjectStructure** può enumerare i suoi elementi e fornire un'interfaccia di alto livello per consentire al visitor di esplorarne il contenuto.