10 - GoF

I design pattern **GoF** (***Gang-of-Four***) sono *schemi di progettazione avanzata* per la programmazione orientata agli oggetti.

> Questi pattern sono stati descritti nel libro "*Design Patterns*" pubblicato nel 1994 da *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides*, noti appunto come la "*Gang of Four*".

L'idea di pattern ebbe origine con i pattern architettonici di costruzione di *Christopher Alexander*. Nel software, i pattern emersero negli anni ottanta grazie a *Ken Beck*, famoso anche per **Extreme Programming**, che riconobbe il lavoro di Alexander e sviluppò l'idea con Ward Cunningham.

Ciascun design pattern GoF descrive una **soluzione progettuale comune a un problema di progettazione ricorrente**.

Sono classificati in base al loro scopo in tre categorie principali:

1. **Creazionali**:
   Risolvono problematiche inerenti all'instanziazione degli oggetti.
   Includono:
   - **Abstract Factory**,
   - **Builder**,
   - **Factory Method**,
   - **Lazy Initialization**,
   - **Prototype Pattern**,
   - **Singleton**,
   - **Double-check Locking**.
2. **Strutturali**:
   Risolvono problematiche inerenti alla struttura delle classi e degli oggetti.
   Includono:
   - **Adapter**,
   - **Bridge**,
   - **Composite**,
   - **Decorator**,
   - **Facade**,
   - **Flyweight**,
   - **Proxy**.
3. **Comportamentali**:
   Forniscono soluzioni alle più comuni tipologie di interazione tra gli oggetti.

Includono:

- **Chain of Responsibility**,
- **Command**,
- **Event Listener**,
- **Hierarchical Visitor**,
- **Interpreter**,
- **Iterator**,
- **Mediator**,
- **Memento**,
- **Observer**,
- **State**,
- **Strategy**,
- **Template Method**,
- **Visitor**.

Principio chiave

Favorire la composizione rispetto all'ereditarietà tra classi.
Questo aiuta a mantenere le classi incapsulate e coese, e la delegazione può rendere la composizione potente quanto l'ereditarietà.

⚠ Attenzione:

L'ereditarietà di classi definisce un oggetto in termini di un altro e permette un riuso "white-box", dove la sottoclasse può accedere ai dettagli implementativi della superclasse.
Tuttavia, è definita staticamente e non può cambiare a tempo di esecuzione, e una modifica alla superclasse può avere ripercussioni indesiderate sulla sottoclasse, non rispettando l'incapsulamento.

✓ La composizione di oggetti:

ottiene funzionalità assemblando o componendo oggetti per creare funzionalità più complesse, con un riuso "black-box" dove i dettagli interni non sono noti.
Se una classe usa un'altra classe, questa può essere referenziata tramite un'interfaccia, permettendo a runtime di utilizzare qualsiasi altra classe che implementi tale interfaccia, rispettando l'incapsulamento. Solo una modifica all'interfaccia comporterebbe ripercussioni.

L'ereditarietà può essere usata per **polimorfismo** (sottoclassi scambiabili, nascondendo il loro tipo effettivo) o **specializzazione** (sottoclassi guadagnano elementi e proprietà rispetto alla classe base).

> **I pattern GoF suggeriscono di diffidare della specializzazione, utilizzando quasi tutti l'ereditarietà per creare polimorfismo**.

In breve, il riuso è meglio ottenerlo attraverso il meccanismo di delega piuttosto che attraverso il meccanismo di ereditarietà, almeno per quanto riguarda l'utilizzo della specializzazione.

Pattern Creazionali

Abstract Factory

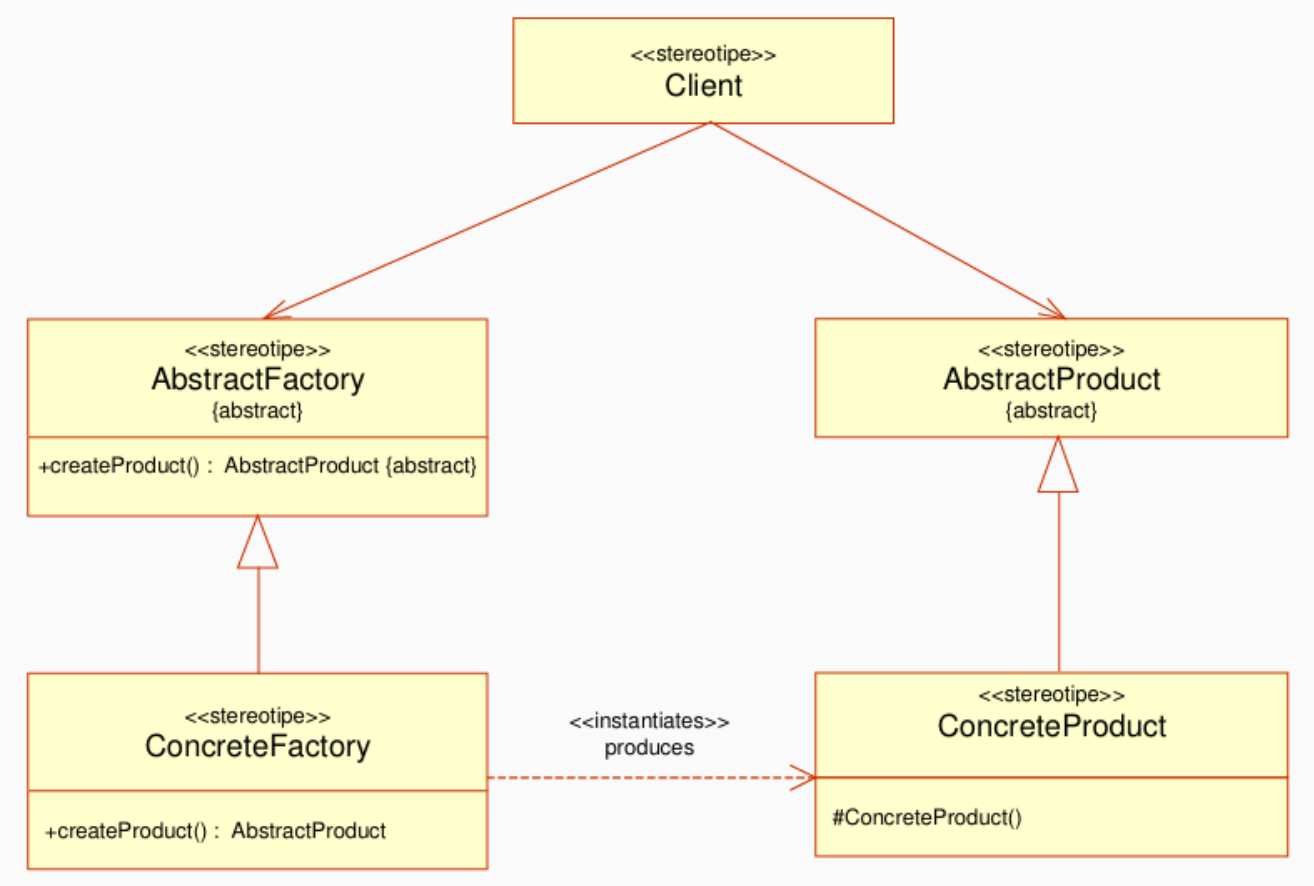
Nome:	Abstract Factory
Problema:	Come creare famiglie di classi correlate che implementano un'interfaccia comune?
Soluzione:	Definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.

Questo pattern presenta un'interfaccia per la creazione di famiglie di prodotti, in modo che il cliente che li utilizza non conosca le loro classi concrete.

Questo assicura che il cliente crei solo prodotti vincolati tra loro e consente l'uso di diverse famiglie di prodotti da parte dello stesso cliente.

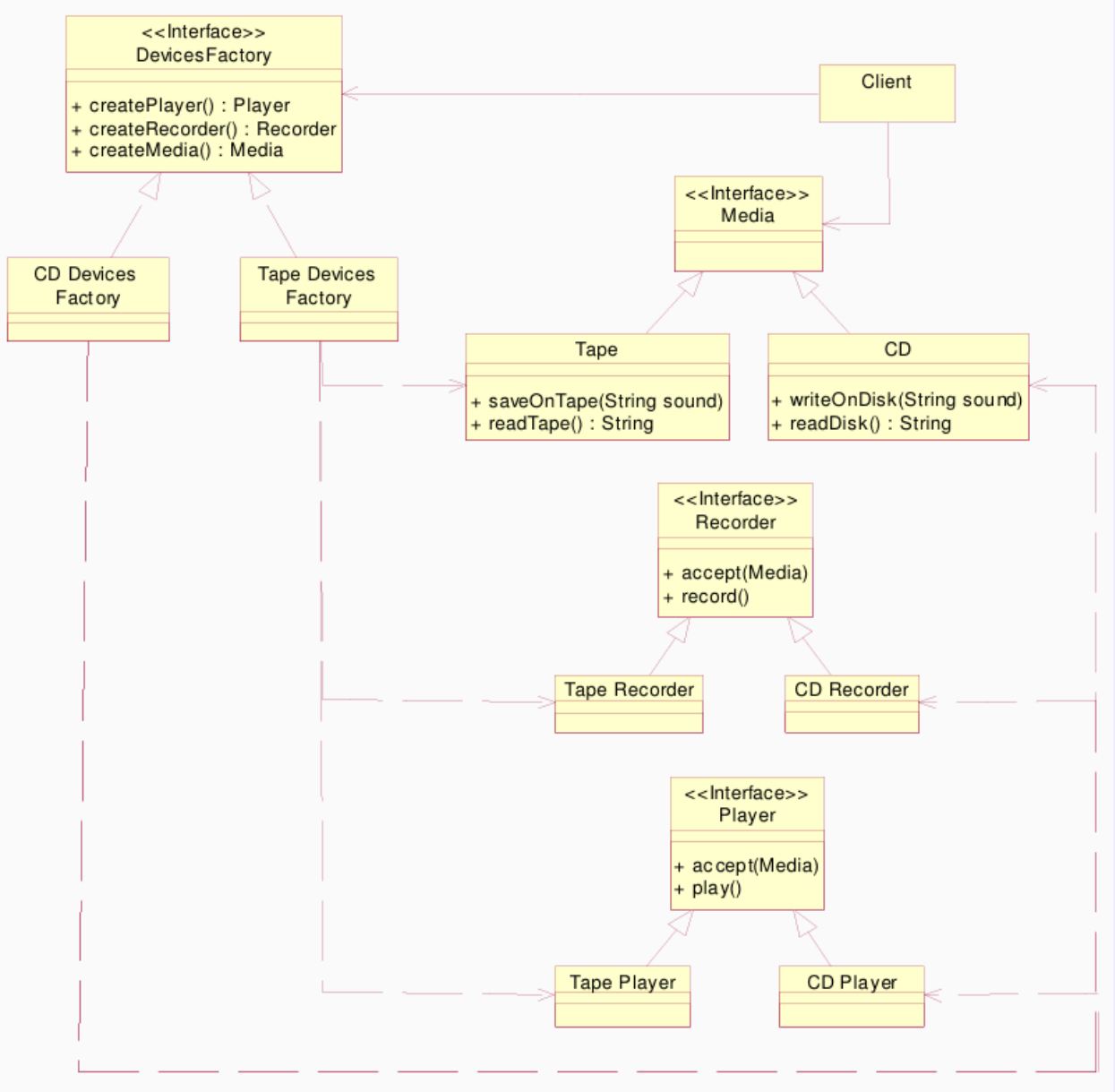
Una variante comune prevede di accedere a una classe astratta factory usando il pattern Singleton.

È usato nelle librerie Java per la creazione di elementi GUI per diversi sistemi operativi.



Esempio

Immaginiamo un negozio Hi-Fi che dimostra sistemi con due famiglie di prodotti: basati su nastro (tape) e su compact disc (CD).
Ogni famiglia include il supporto (tape/CD), un registratore (recorder) e un riproduttore (player).
Il problema è creare queste famiglie senza vincolare il codice del cliente alle specifiche famiglie.



La soluzione consiste nel creare interfacce per ogni tipo di prodotto (`Media` , `Player` , `Recorder`), prodotti concreti che le implementano (`Tape` , `TapeRecorder` , `TapePlayer` , `CD` , `CDRecorder` , `CDPlayer`), e classi factory (`TapeDevicesFactory` , `CDDevicesFactory`) che implementano un'interfaccia comune (`DevicesFactory`).

Il cliente interagisce solo con l'interfaccia `DevicesFactory` per creare prodotti, senza conoscere le classi concrete.

ⓘ Info

Nelle implementazioni Java, `AbstractFactory` e `AbstractProduct` sono spesso codificati come interfacce piuttosto che classi astratte.

Singleton

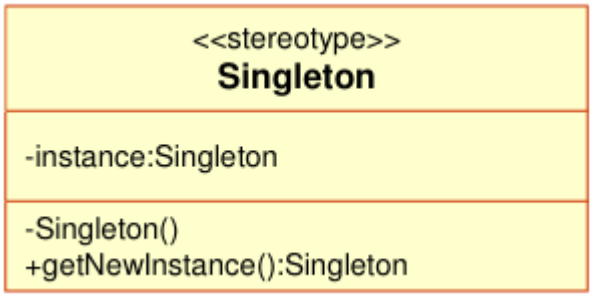
Nome: Singleton

Problema: È consentita (o richiesta) esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

Soluzione: Difinisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton.

Il pattern Singleton definisce una classe di cui è possibile istanziare un unico oggetto. Diverse richieste di istanziazione comportano la restituzione di un **riferimento allo stesso oggetto**.

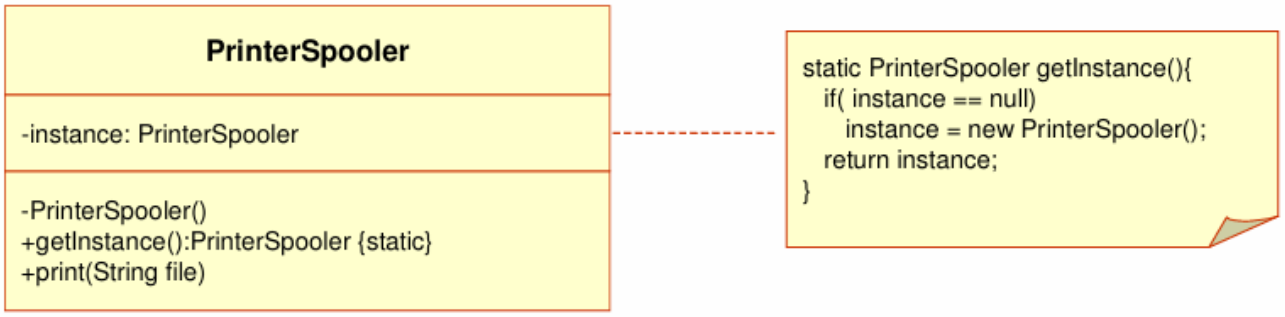
In UML, un singleton è indicato con un "1" nella sezione del nome, in alto a destra.



Struttura:

- Una classe `Singleton` con un costruttore privato
- Un metodo statico pubblico `getNewInstance()` che restituisce l'unica istanza.

Implementazioni in Java:



Singleton come classe statica:

Non è un vero Singleton, si lavora con la classe statica e i suoi metodi statici.
Ha il costruttore privato per impedire l'istanziazione.

```
public static class PrinterSpooler {

    private PrinterSpooler() { }

    public static void print (String msg) {
        System.out.println( msg );
```

```
    }
}
```

Un limite è che non può implementare interfacce e richiede la conoscenza completa per la sua creazione al momento del caricamento della classe.

Singleton creato da un metodo statico:

Una classe con un metodo statico (`getInstance()`) che restituisce l'istanza.

```
public class PrinterSpooler {
    private static PrinterSpooler instance;

    private PrinterSpooler() { }

    public static PrinterSpooler getInstance() {
        if (instance==null) {
            instance = new PrinterSpooler();
        }
        return instance;
    }

    public void print (String msg) {
        System.out.println( msg );
    }
}
```

L'oggetto viene istanziato solo la prima volta, e le successive restituisce un riferimento allo stesso oggetto (inizializzazione pigra).

✓ **Preferibile rispetto alla classe statica:**

- i metodi d'istanza consentono la ridefinizione nelle sottoclassi,
- la maggior parte dei meccanismi di comunicazione remota supporta solo l'accesso a metodi d'istanza,
- una classe non è sempre un singleton in tutti i contesti applicativi.

L'inizializzazione pigra è preferibile a quella golosa (quando la classe è caricata) perché evita la creazione dell'oggetto se non viene mai usato.

Singleton multi-thread:

Una versione della soluzione precedente che sincronizza il metodo `getInstance()` per prevenire la creazione di istanze multiple in ambienti multi-thread.

```java
public class PrinterSpooler {
    private static PrinterSpooler instance;

    private PrinterSpooler() { }

    public static synchronized PrinterSpooler getInstance() {
        if (instance==null) {
            instance = new PrinterSpooler();
        }
        return instance;
    }

    public void print (String msg) {
        System.out.println( msg );
    }
}
```

La soluzione presentata può essere considerata inefficiente perché ogni volta che si fa una invocazione al metodo `getInstance()` deve essere acquisito un lock.
Una strategia nota ma scorretta in Java è il "*double-checked locking*", poiché criteri di ottimizzazione della JVM possono comunque portare a istanze multiple.

⚠ **Si deve prestare attenzione a:**

- presenza di Singleton in virtual machine multiple,
- Singleton caricati contemporaneamente da diversi class loader,
- Singleton distrutti dal garbage collector e ricaricati,
- presenza di istanze multiple come sottoclassi,
- copia di Singleton come risultato di un doppio processo di deserializzazione.

Pattern Strutturali

Adapter

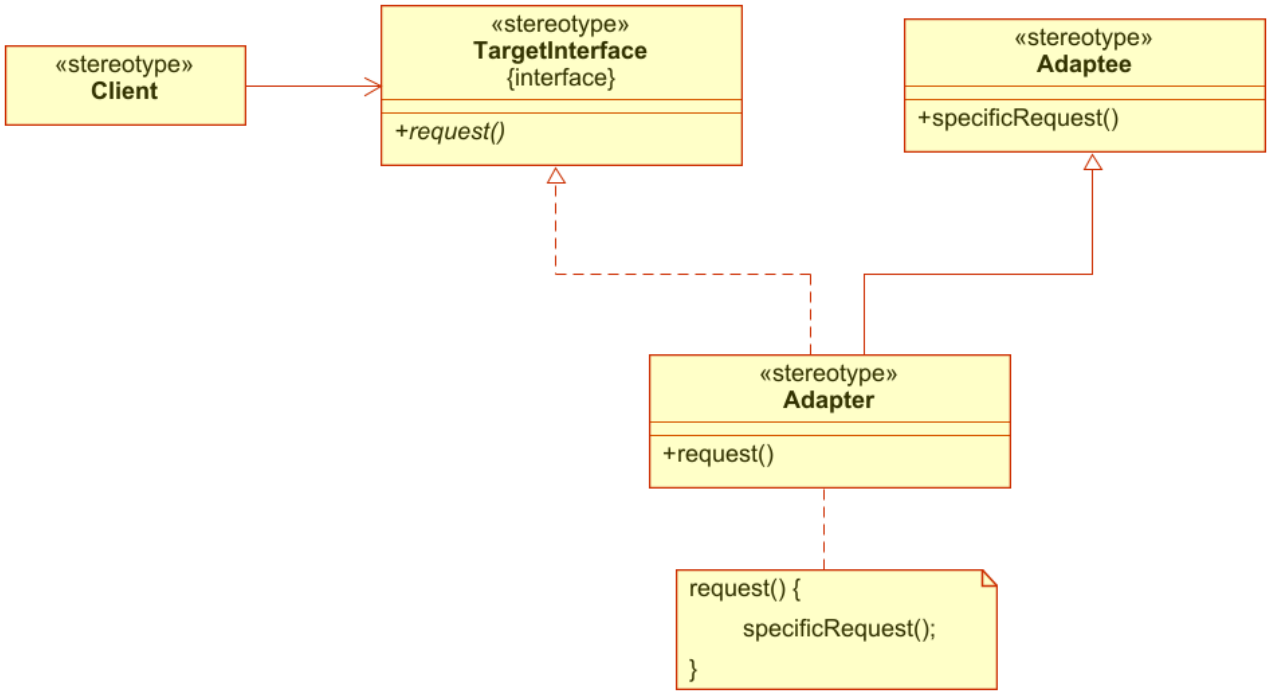
Nome:	Adapter
Problema:	Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?
Soluzione:	Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio.

Si usa quando un oggetto client vuole fruire di servizi offerti da un oggetto server, ma le loro interfacce sono incompatibili.
L'adattatore riceve richieste nel formato del client, le trasforma nel formato del server, le

invia al server, e poi trasforma le risposte del server nel formato del client prima di restituirle. È anche utile quando più oggetti server offrono servizi simili ma con interfacce diverse.

L'Adapter pattern offre due soluzioni possibili:

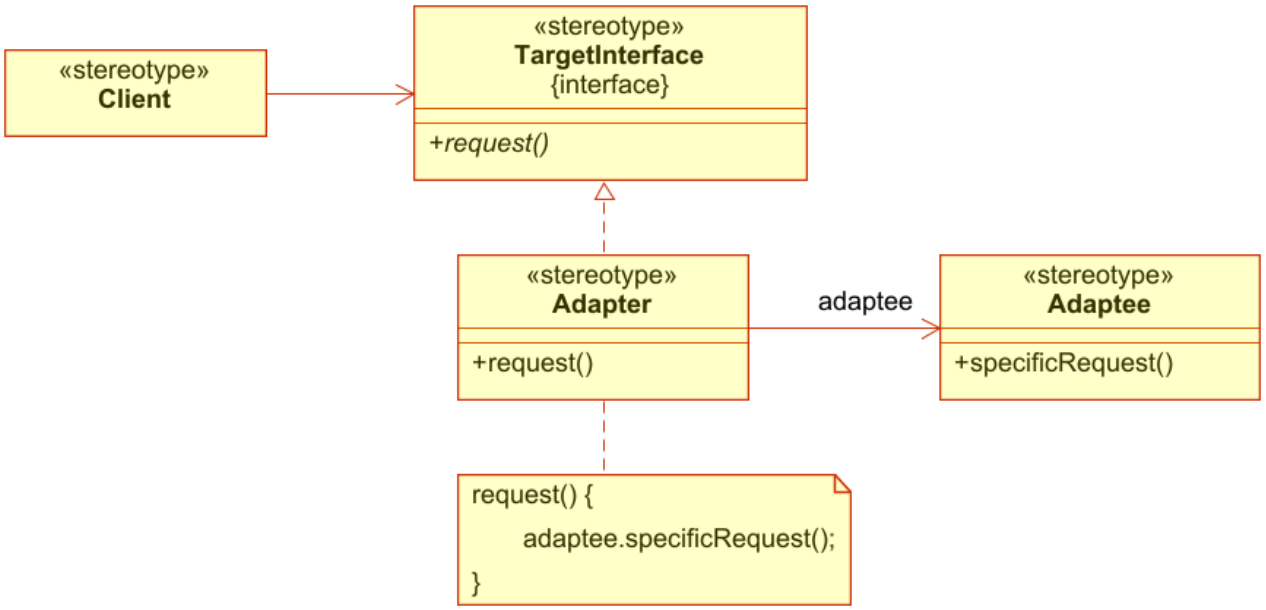
Class Adapter:



La classe esistente (`Adaptee`) viene estesa in una sottoclasse (`Adapter`) che implementa l'**interfaccia** desiderata (`TargetInterface`).

I metodi della sottoclasse mappano le loro operazioni alle richieste ai metodi e attributi della classe di base.
Questo è possibile solo se l'Adaptee non è una `final class` .

Object Adapter:



Si crea una nuova classe (`Adapter`) che implementa l'interfaccia richiesta (`TargetInterface`) e che possiede al suo interno un'istanza della classe da riutilizzare (`Adaptee`).
Le operazioni della nuova classe invocano i metodi dell'oggetto interno.

Composite

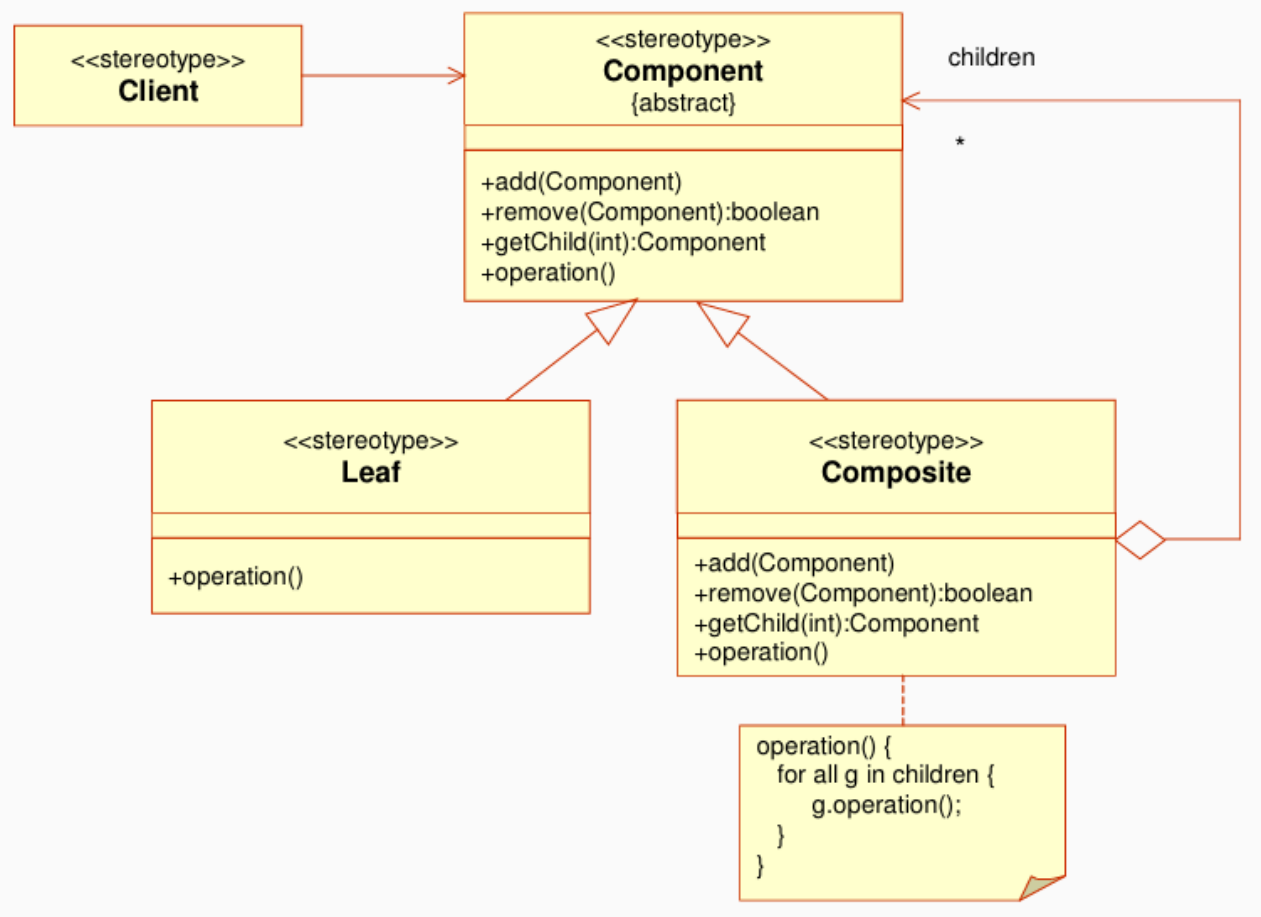
Nome:	Composite
Problema:	Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?
Soluzione:	Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

Consente la costruzione di gerarchie di oggetti "tutto-parte".
È utile quando si vogliono rappresentare queste gerarchie e ignorare le differenze tra oggetti singoli e oggetti composti.

È anche noto come struttura ad albero, composizione ricorsiva o struttura induttiva, dove foglie e nodi hanno la stessa funzionalità.

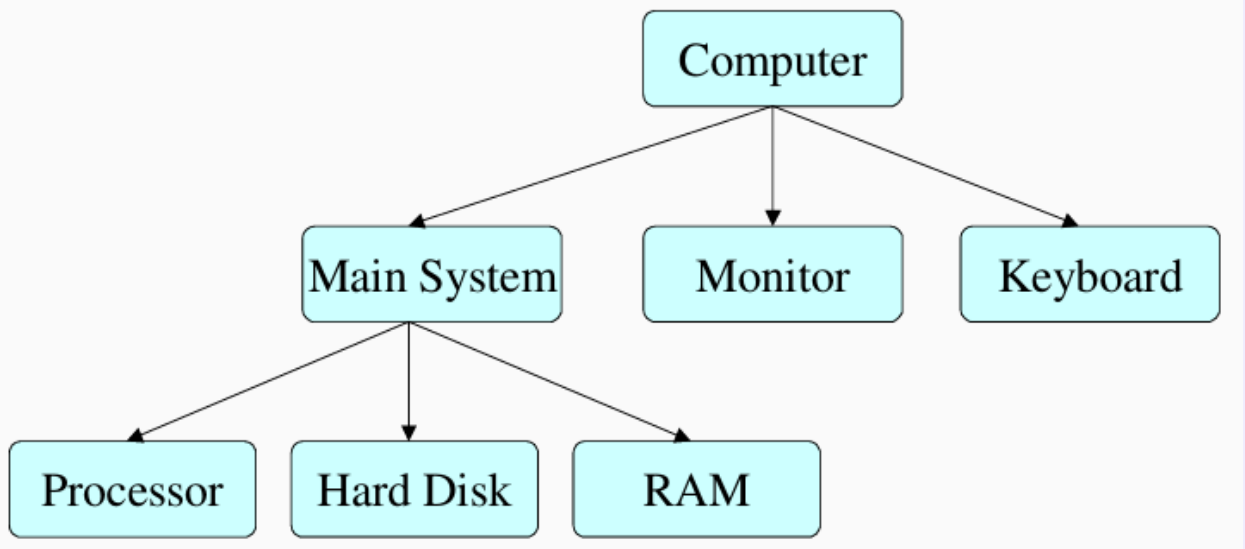
Implementa la stessa interfaccia per tutti gli elementi contenuti.
Il pattern definisce una classe astratta `Component` che deve essere estesa da due sottoclassi: `

- `Leaf` (per i singoli componenti) e
- `Composite` (per i componenti composti, che possono contenere sia `Leaf` che altri `Composite`).



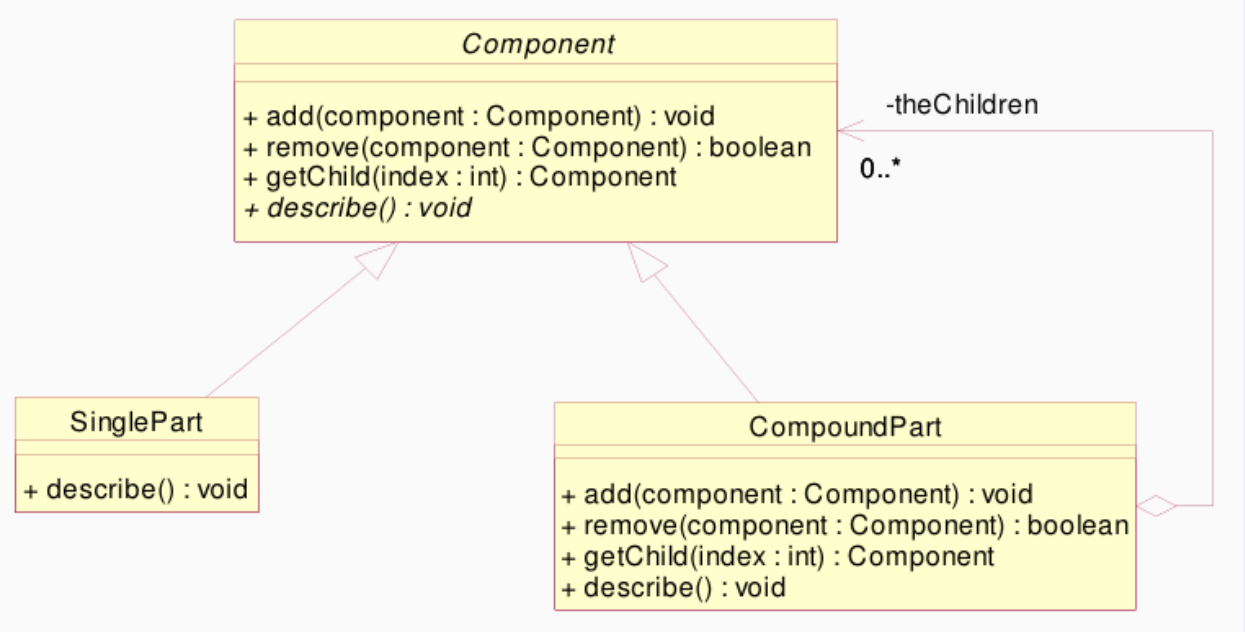
Esempio

Rappresentare in un magazzino componenti di computer, che possono essere pezzi singoli (es. "monitor", "tastiera") o composti (es. "main system" formato da "processore", "disco rigido", "RAM"; o "computer" formato da "main system", "monitor", "tastiera").



La classe `Component` (astratta) definisce l'interfaccia comune. `SinglePart` estende `Component` per i pezzi singoli. `CompoundPart` estende `Component` per i pezzi composti, gestendo una collezione di figli e implementando metodi per aggiungere/rimuovere componenti.

Il client interagisce con tutti tramite l'interfaccia `Component`.



In breve

Il pattern composite permette di costruire strutture **ricorsive** (ad esempio un albero di elementi) in modo che ad un cliente (una classe che usa la struttura) l'intera struttura sia vista come una singola entità.
Quindi l'interfaccia alle entità atomiche (foglie) è esattamente la stessa dell'interfaccia delle entità composte.

In essenza tutti gli elementi della struttura hanno la stessa interfaccia senza considerare se sono composti o atomici.

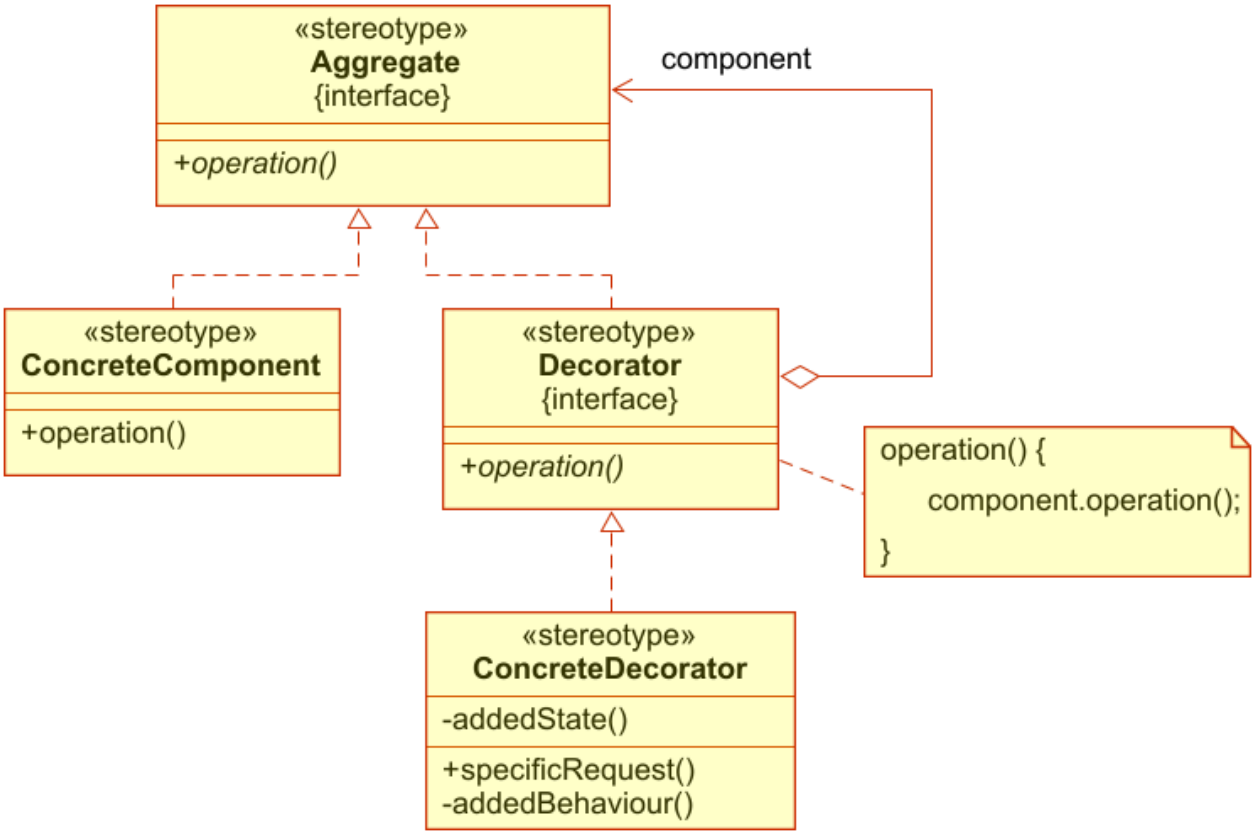
Decorator

Nome: Decorator
Problema: Come permettere di assegnare una o più responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica? Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?
Soluzione: Inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità.

> È anche noto come Wrapper.

Permette di aggiungere responsabilità a oggetti individualmente, dinamicamente e in modo trasparente, senza impatti sugli altri oggetti.
Le responsabilità possono essere ritirate.

Evita l'esplosione delle sottoclassi che si verificherebbe per supportare un ampio numero di estensioni e combinazioni di esse.



Composite vs Decorator

- Il pattern Composite fornisce un'interfaccia comune a elementi atomici (foglie) e composti.
- Il pattern Decorator fornisce caratteristiche aggiuntive a elementi atomici (foglie), mantenendo un'interfaccia comune.

Pattern Comportamentali

Observer

Nome: Observer

Problema: Diversi tipi di oggetti subscriber (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher (editore). Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

Soluzione: Definisci un'interfaccia *subscriber* o *listener* (ascoltatore). Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

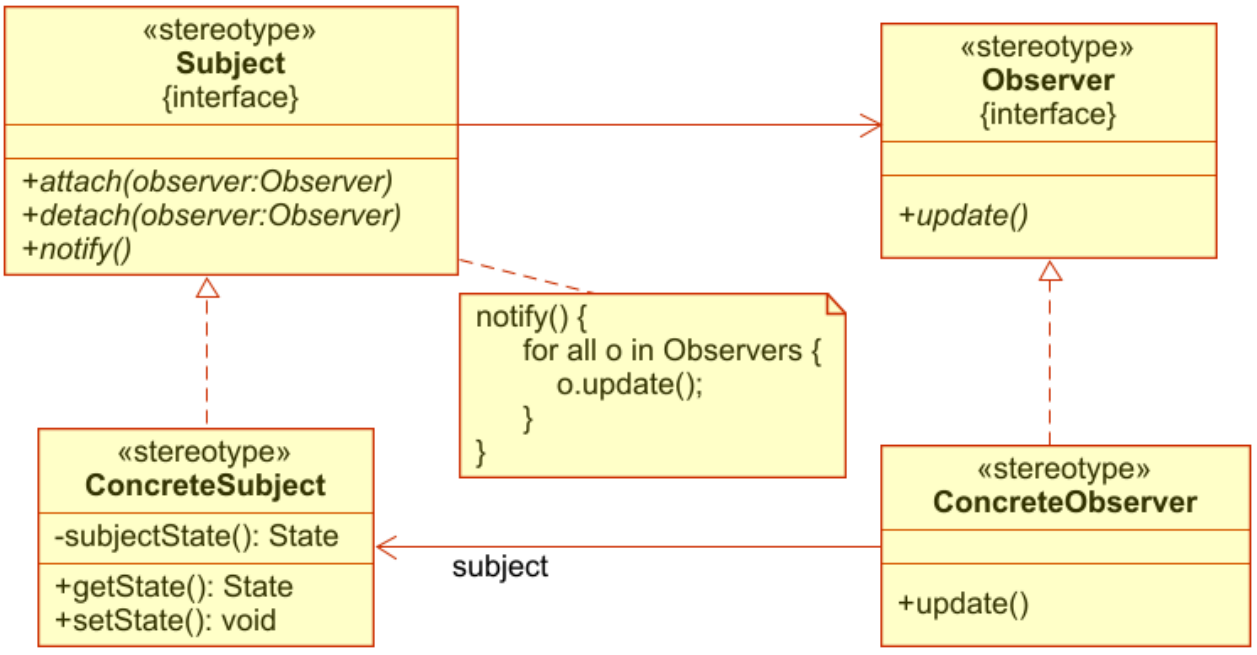
È noto anche come *Dependents* e *Publish-Subscribe*.

Definisce una dipendenza uno-a-molti: quando lo stato di un oggetto cambia, viene notificato a tutti gli oggetti dipendenti che vengono automaticamente aggiornati.

L'oggetto che notifica non fa assunzioni sulla natura degli oggetti notificati, risultando in un disaccoppiamento.

Il numero di oggetti affetti non è noto a priori. I publisher conoscono i subscriber solo tramite un'interfaccia, e i subscriber possono registrarsi/cancellarsi dinamicamente.

Spesso è associato al pattern architetturale Model-View-Controller (MVC), dove le modifiche al modello sono notificate alle viste (osservatori).



`Observable` è una classe che, tramite l'estensione, fornisce i metodi di base del `Subject`. Questo vieta la possibilità che il `Subject` possa essere implementato contemporaneamente come una estensione di un'altra classe.

Nota:

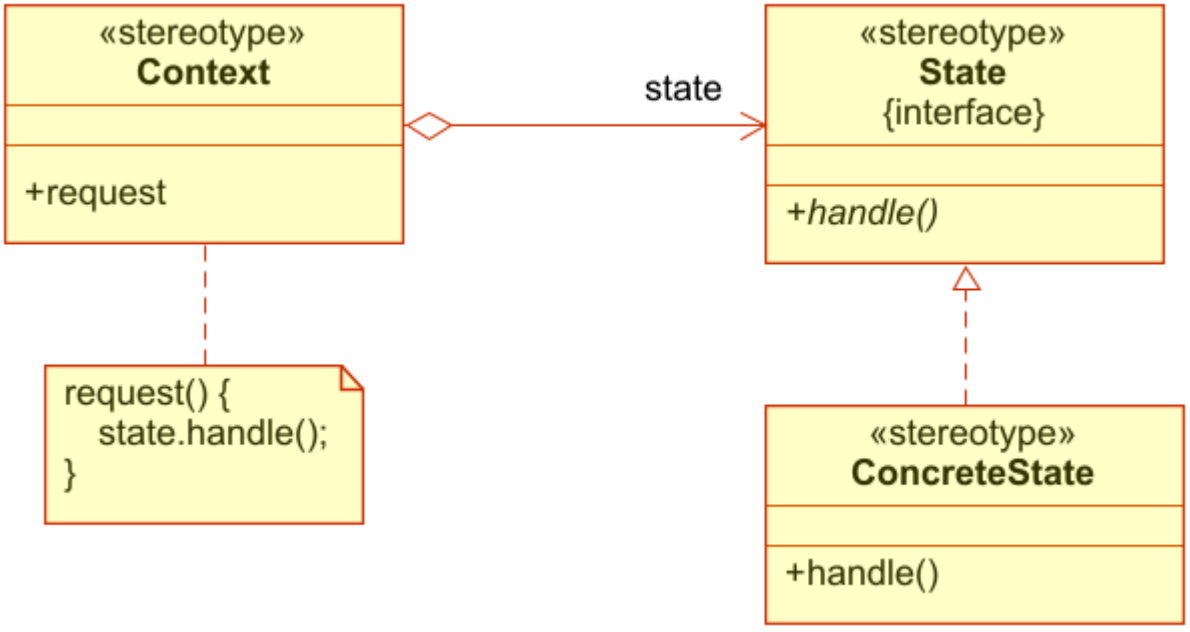
Nel pattern GoF originale, Subject è un'interfaccia, ma Java ha scelto di fornirlo come classe concreta per semplificare l'uso (`java.util.Observable`).

State

Nome:	State
Problema:	Il comportamento di un oggetto dipende da suo stato e i suoi metodi contengono logica condizionale per casi che riflette le azioni condizionali che dipendono dallo stato. C'è un'alternativa alla logica condizionale?
Soluzione:	Crea delle classi stato per ciascuno stato, che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato dall'oggetto contesto all'oggetto stato corrente corrispondente. Assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.

Permette a un oggetto di modificare il suo comportamento quando cambia il suo stato interno.
Può sembrare che l'oggetto modifichi la sua classe.

Questo pattern incapsula il modo in cui le operazioni di un oggetto (`Context`) vengono svolte quando si trova in un determinato stato.
Ogni classe (`ConcreteState`) rappresenta un singolo stato possibile del `Context` e implementa un'interfaccia comune (`State`) contenente le operazioni che il `Context` delega allo stato.



Strategy

Nome: Strategy

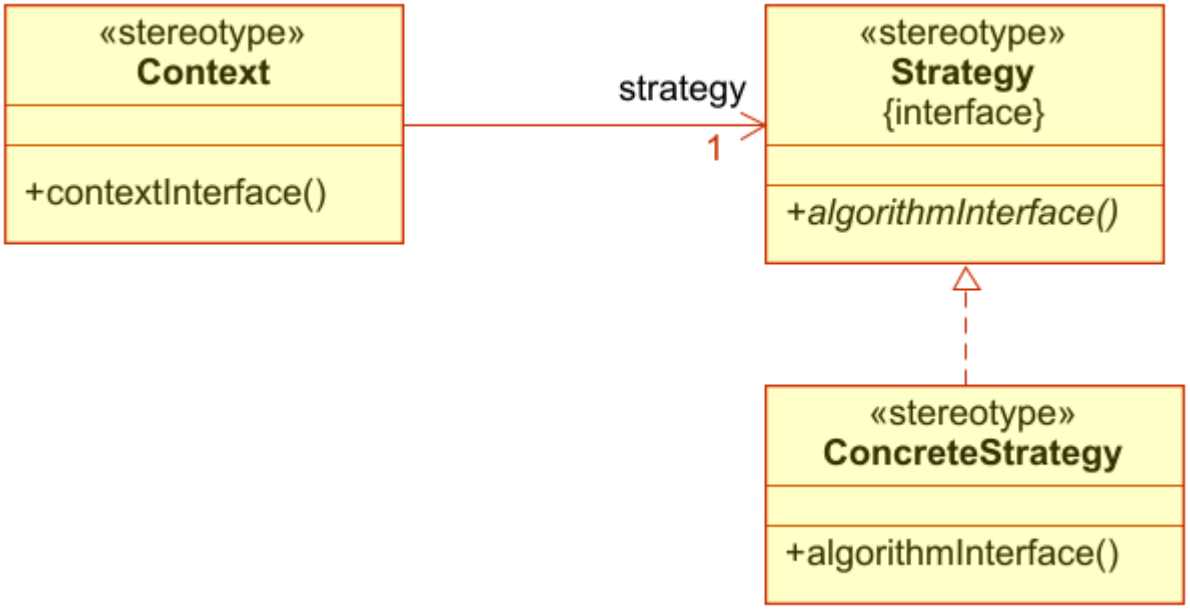
Problema: Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?

Soluzione: Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune.

L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo, ed è associato a un oggetto strategia che implementa l'algoritmo.

Strategy consente la definizione di una **famiglia di algoritmi**, incapsulando ognuno e rendendoli intercambiabili tra loro.
Permette di modificare gli algoritmi **indipendentemente** dai clienti che li usano, disaccoppiando gli algoritmi dai clienti e consentendo a un client di usare indifferentemente l'uno o l'altro algoritmo.
È utile quando è necessario modificare il comportamento di una classe a runtime.

Usa la **composizione** invece dell'ereditarietà: i comportamenti non dovrebbero essere ereditati ma incapsulati usando dichiarazioni di interfaccia.



Strategy vs State

Sono molto simili.

- Il pattern **State** si occupa di **cosa (stato o tipo) un oggetto è (al suo interno)** e incapsula un comportamento dipendente dallo stato, permettendo di fare cose diverse in base allo stato, sollevando il chiamante dall'onere di gestire ogni stato possibile.

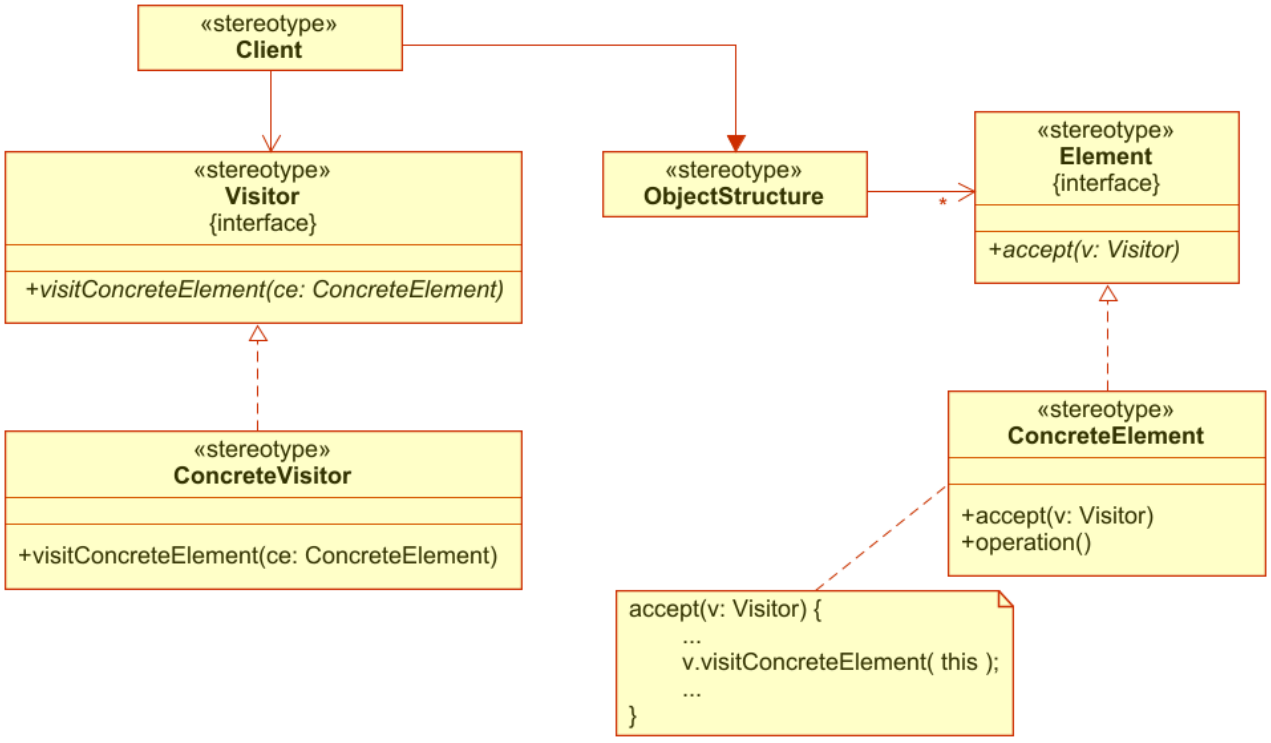
- Il pattern **Strategy** si occupa del **modo in cui un oggetto esegue un determinato compito**: incapsula un algoritmo, consentendo che implementazioni diverse della stessa funzionalità possano sostituirsi a seconda della strategia richiesta.

Visitor

Nome:	Visitor
Problema:	Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo degli elementi?
Soluzione:	Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad un'interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

Offre flessibilità delle operazioni, organizzazione logica, visita di vari tipi di classe e mantenimento di uno stato aggiornabile a ogni visita.

Le diverse modalità di visita della struttura possono essere definite come sottoclassi del `Visitor`.


«stereotype»
Client
«stereotype»
Visitor
{interface}
+visitConcreteElement(ce: ConcreteElement)
«stereotype»
ObjectStructure
«stereotype»
Element
{interface}
+accept(v: Visitor)
*
«stereotype»
ConcreteVisitor
+visitConcreteElement(ce: ConcreteElement)
«stereotype»
ConcreteElement
+accept(v: Visitor)
+operation()
accept(v: Visitor) {
...
v.visitConcreteElement(this);
...
}