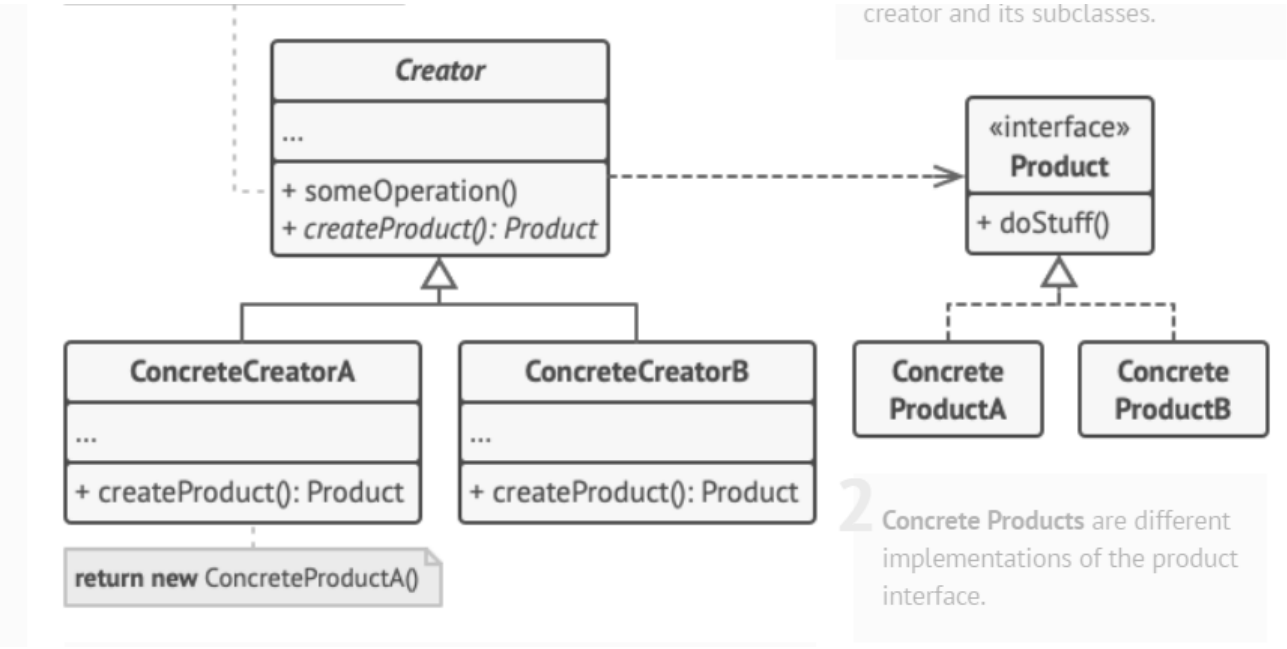
Design Patterns

Materiale citato: Slide del prof, https://refactoring.guru/design-patterns, https://en.wikipedia.org/wiki/God_object, https://www.linkedin.com/pulse/single-responsibility-principle-software-design-sanjoy-kumar-malik#:~:text=Separation%20of%20Concerns%20and%20SRP,and%20flexibility%20of%20software%20systems, https://stackoverflow.com/questions/2137201/composition-vs-delegation

Pattern Creazionali

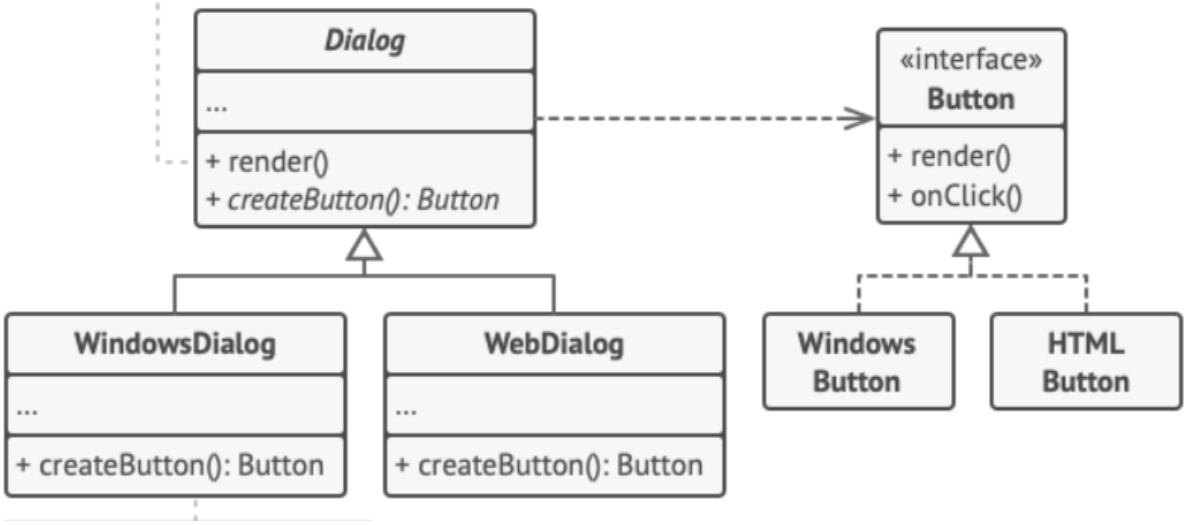
Factory method
L'abstract Factory è un pattern creazionale che delega alle sottoclassi l'istanzazione delle classi.
Quale problema risolve?
L'abstact factory torna molto utile quando non conosciamo a priori i tipi e le dipendenze degli oggetti che dobbiamo usare, favorisce il riuso del codice, e soddisfa i principi di open/closed e di single responsibility.
Diventa scomodo se si deve lavorare con un numero elevato di sottoclassi

Struttura



In questo modo le classi concrete dei creatori A e B riescono attraverso l'overriding a scegliere il prodotto che preferiscono.

Esempio



Il factory method può spesso essere visto come il punto d'inizio su cui basarsi, infatti questo pattern tende ad evolversi in un abstract factory, un builder o un prototype.

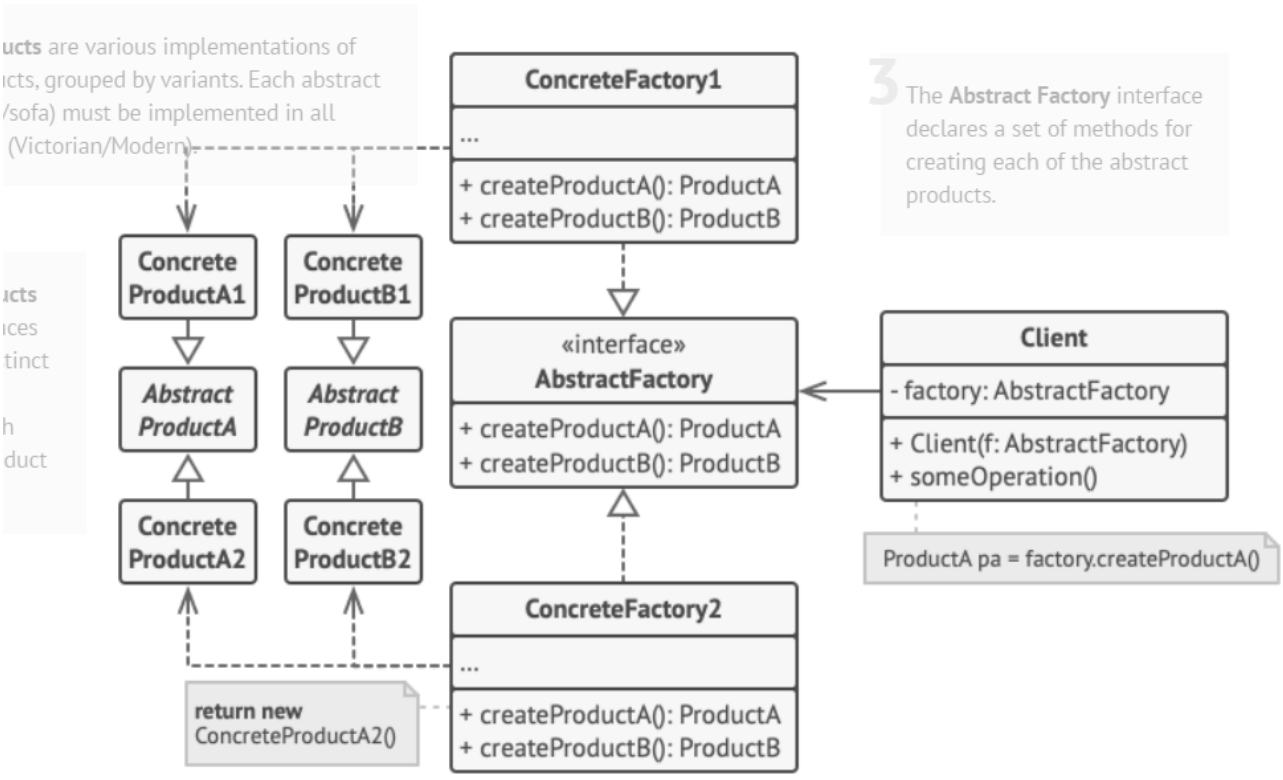
Abstract Factory

Pattern creazionale che permette di produrre *famiglie* di oggetti correlati fra loro.
N.B. Estensione del Factory Method in cui la famiglia è una sola.

Quale problema risolve

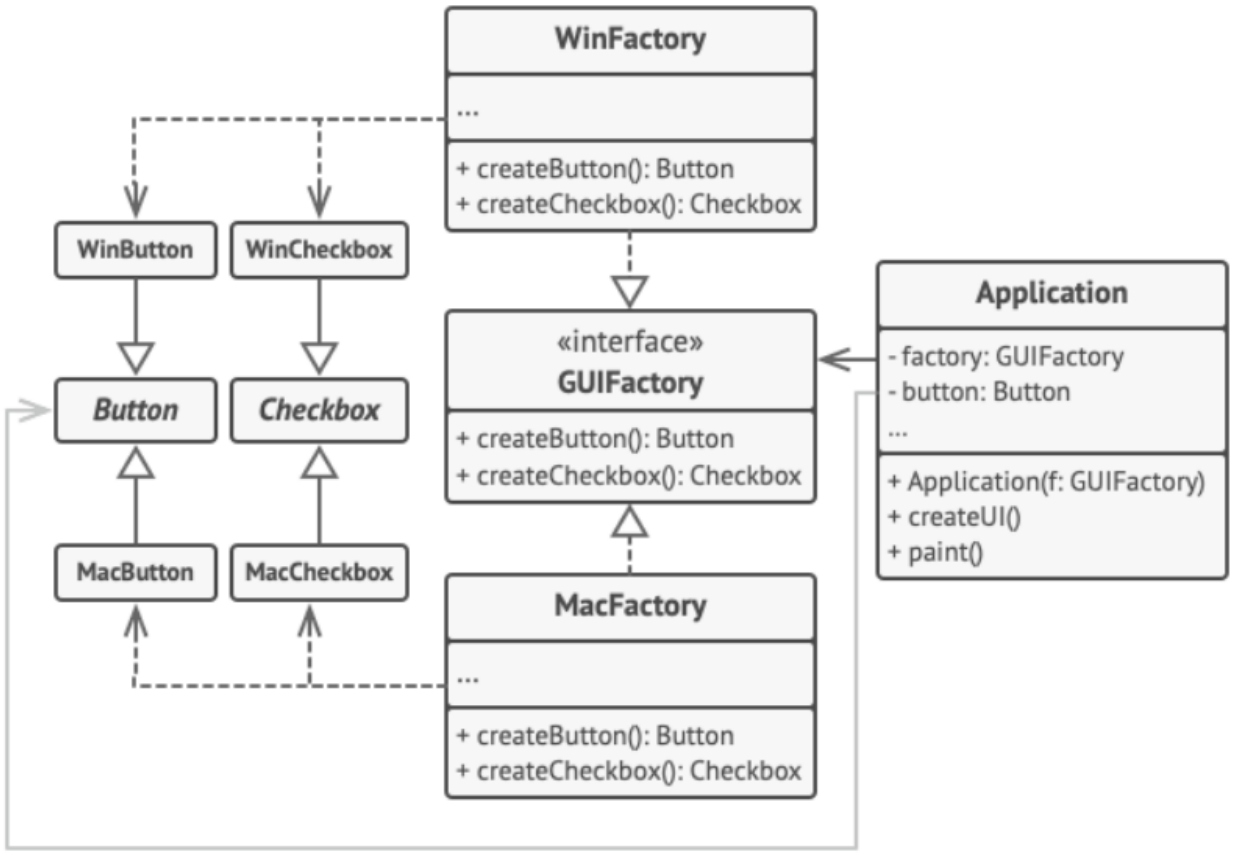
Supponendo che il nostro codice si evolva per supportare diverse famiglie di oggetti, ciò che va mantenuto è il principio di single responsibility, e va evitato che ad ogni aggiunta di un oggetto si vadano a modificare delle classi già scritte. (Open closed Principle)

Struttura



In questo caso le fattorie sono due, ed entrambe producono due oggetti, in questo caso il client si interfaccia con l'abstract factory ma gli viene nascosta l'implementazione interna.

Esempio applicativo



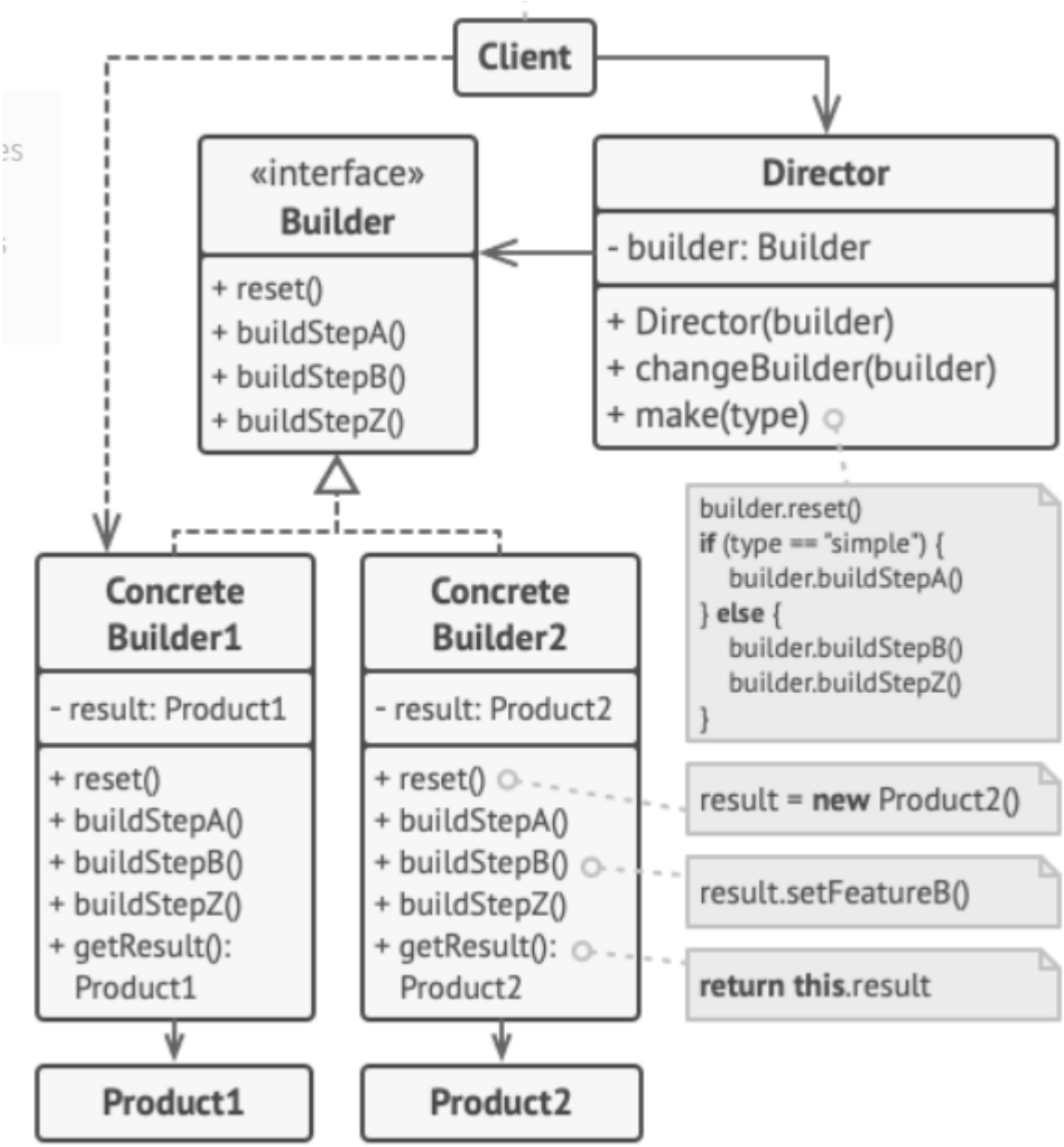
In questo caso le famiglie sono windows e mac e i prodotti sono button e checkbox.
Nel caso si debba utilizzare solo un'istanza per ogni famiglia di prodotto l'abstarct factory si implementa come Singleton.

Builder
Il Pattern Builder permette di separare la creazione dell'oggetto dalla sua rappresentazione attraverso la costruzione dell'oggetto step-by-step

Quale problema risolve?
Il builder è fondamentale quando un oggetto richiede molti parametri (si pensi alla costruzione di una casa) e molti di questi non sempre vengono utilizzati (ad esempio alcune case non hanno il piano superiore, il bagno extra, antifurto ecc.).

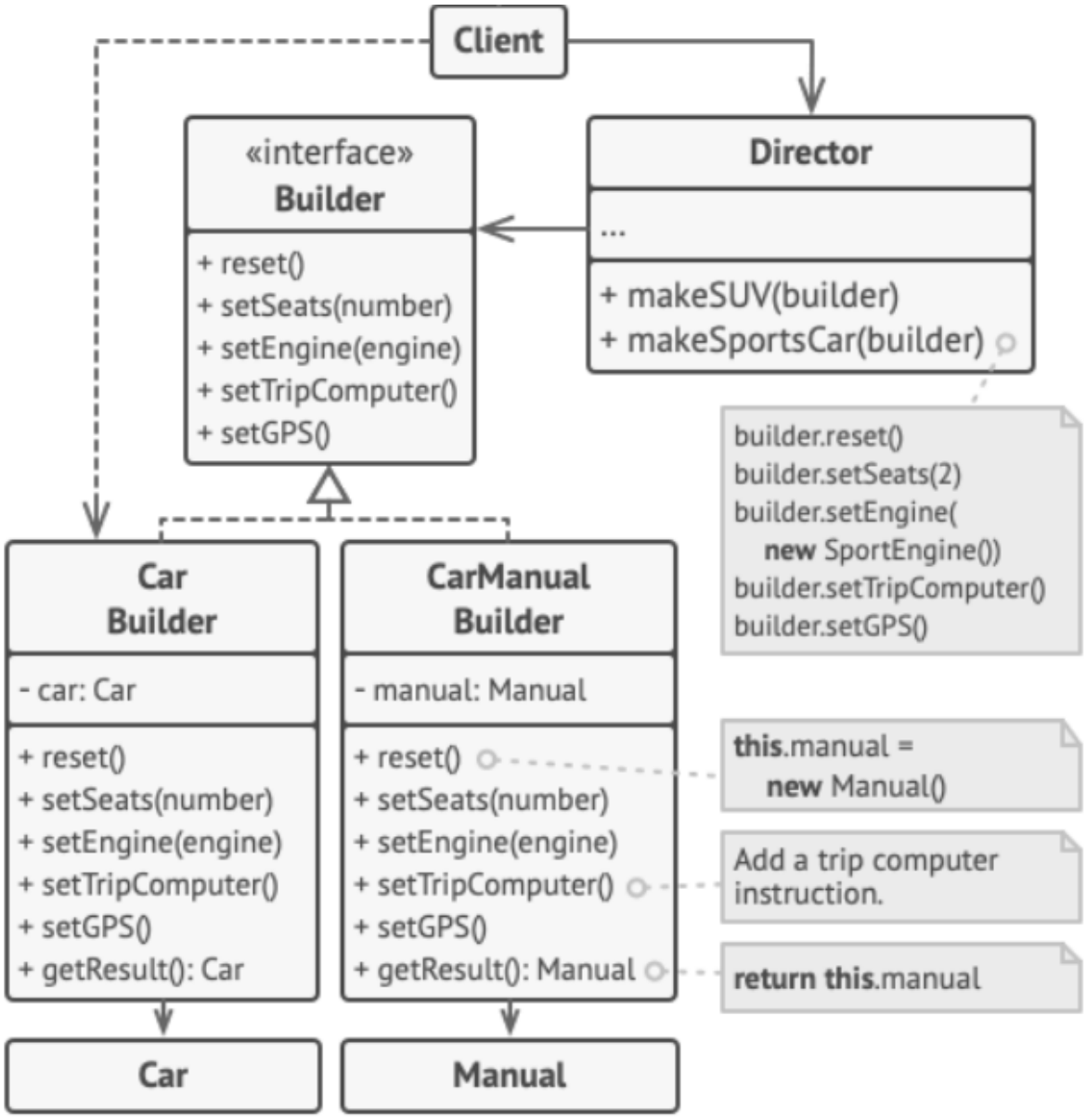
Il Builder si affianca ad un director per che lo aiuta e ordinare i vari step che devono essere implementati.

Struttura



Il director ha un riferimento al builder e delle istruzioni su come implementare i vari step, mentre il builder è un interfaccia che propone i vari step che le varie classi concrete devono seguire.

Esempio applicativo

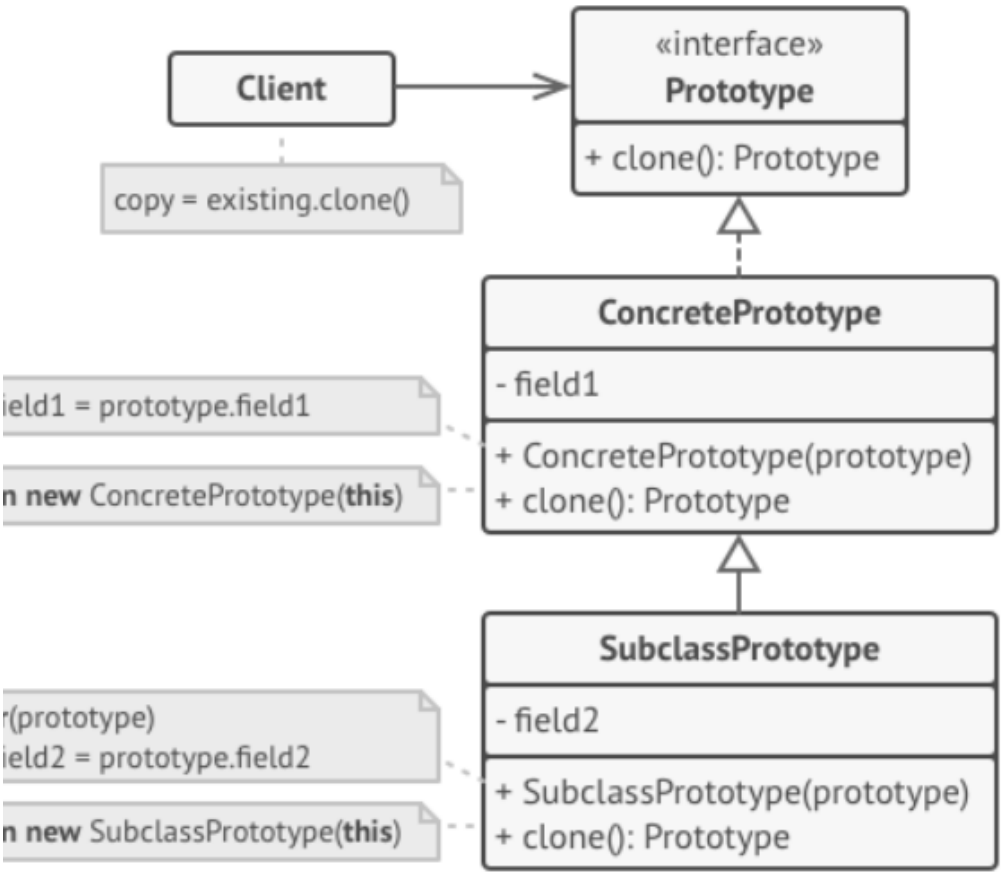


In questo caso abbiamo due builder che implementano gli stessi metodi ma in maniera completamente diversa, infatti lo stesso metodo viene visto in un modo dalla macchina e in un altro dal suo manuale.
Il pattern favorisce riutilizzo del codice, e il single responsibility principle.
Anche questo pattern può essere implementato un singleton

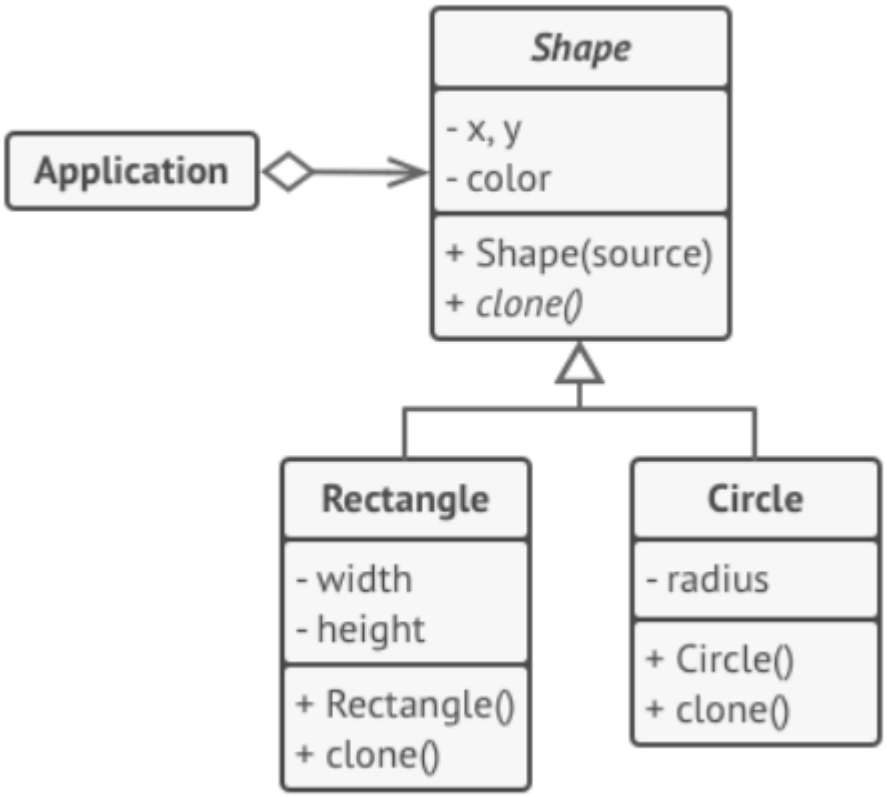
Prototype
Prototype è un pattern creazionale che permetta la clonazione di oggetti esistenti in modo tale che non si debba dipendere dalle classi.

Quale problema risolve?
Non sempre è possibile copiare tutti i campi della classe in quanto alcuni potrebbero essere privati, inoltre supponendo che invece la clonazione sia possibile supponendo pubblici i campi dell'oggetto, accedere alla classe ed esserne dipendenti non è una buona pratica ingegneristica.

Struttura



Vediamo che il prototype è un'interfaccia con un singolo metodo che viene ereditato da tutte le classi figlie.

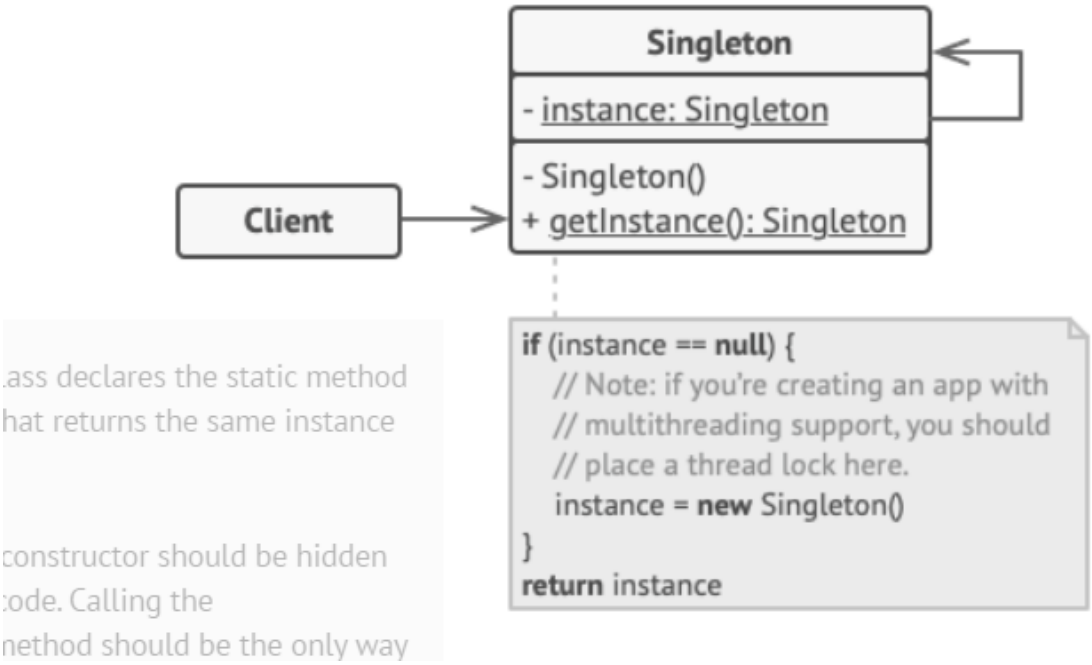


Si noti che siccome tutte le sottoclassi ereditano il metodo clone, l'implemenzione risulta insidiosa quando la dipendenza è circolare

Singleton
Pattern creazionale che ti assicura che la classe abbia una sola istanza in contemporanea ad un accesso globalizzato

Quale problema risolve?
Quando si lavora con oggetti esclusivi, come un database o un file, ed ognuno ne vuole una copia, l'unico modo possibile è tramite l'implementazione di un Singleton.

Struttura



La struttura è molto semplice, l'unica cosa a cui fare attenzione è il riferimento circolare alla classe.

Il problema principale di questo codice è che non rispetta il principio della singola responsabilità e il suo uso potrebbe mascherare della cattiva progettazione.

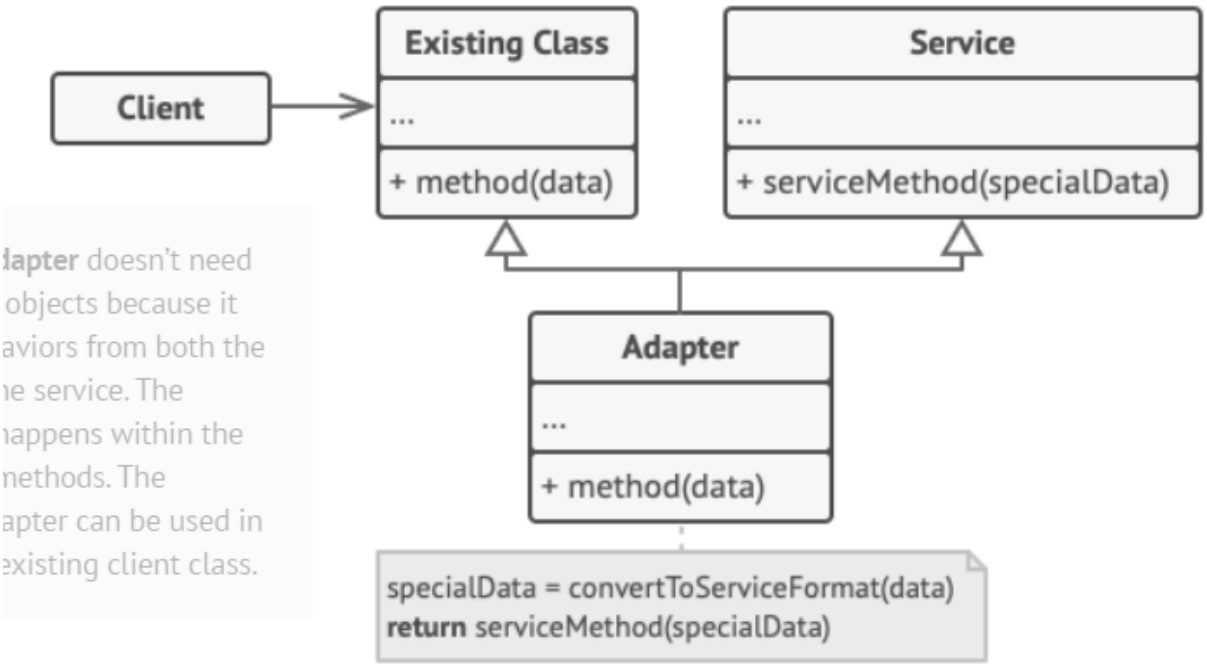
Pattern Strutturali

Adapter
L'adapter permette ad oggetti con interfacce diverse di interagire tra loro

Quale problema risolve?
Solitamente questo pattern viene utilizzato quando lo sviluppatore richiede un servizio di terze parti che ha un interfaccia incompatibile con il codice sviluppato.

Sono possibili due implementazioni: una tramite l'inheritance e una tramite la composizione, si parla nel primo caso di class adapter e nel secondo di object adapter.

Class Adapter



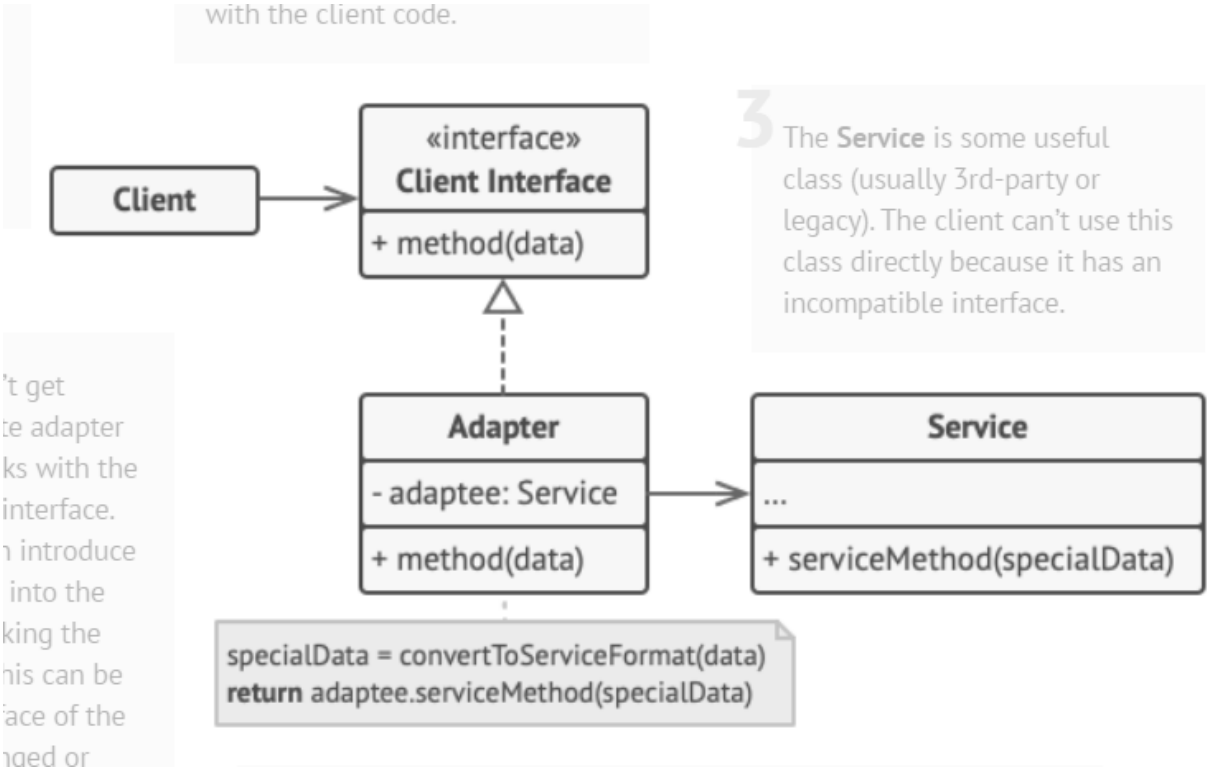
Vediamo come la classe adapter erediti sia la classe esistente che il servizio

Quali problemi potrebbero presentarsi?

L'ereditarietà (inheritance) ha un grosso downgrade implementativo: non è sempre disponibile Nel caso di java il servizio di terze parti che stiamo cercando di adattare potrebbe essere final.

Object Adapter

'This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.'



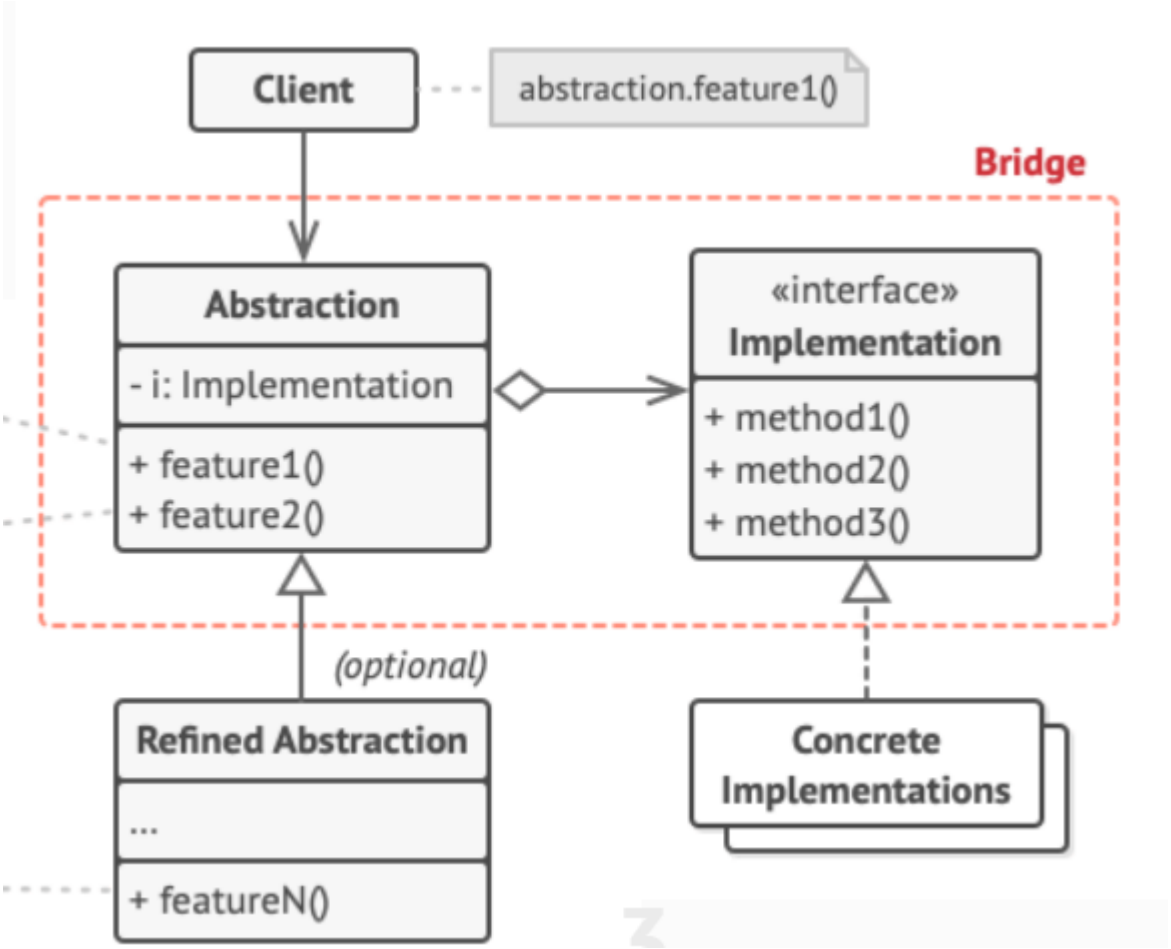
Come la maggior parte dei design patterns l'adapter soddisfa i primi due principi SOLID (in entrambe le sue implementazioni)

Bridge
Patterns strutturale che permette di separare una classe ampia in due gerarchie separate: abstraction e implementation

Quale problema risolve?
Disaccoppia l'implentazione dall'astrazione in modo in modo tale che ognuna venga sviluppata indipendentemente dall'altra.

> **Differenza tra astrazione e implementazione**
>
> *Abstraction* (also called *interface*) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own. It should delegate the work to the *implementation* layer (also called *platform*).

Struttura

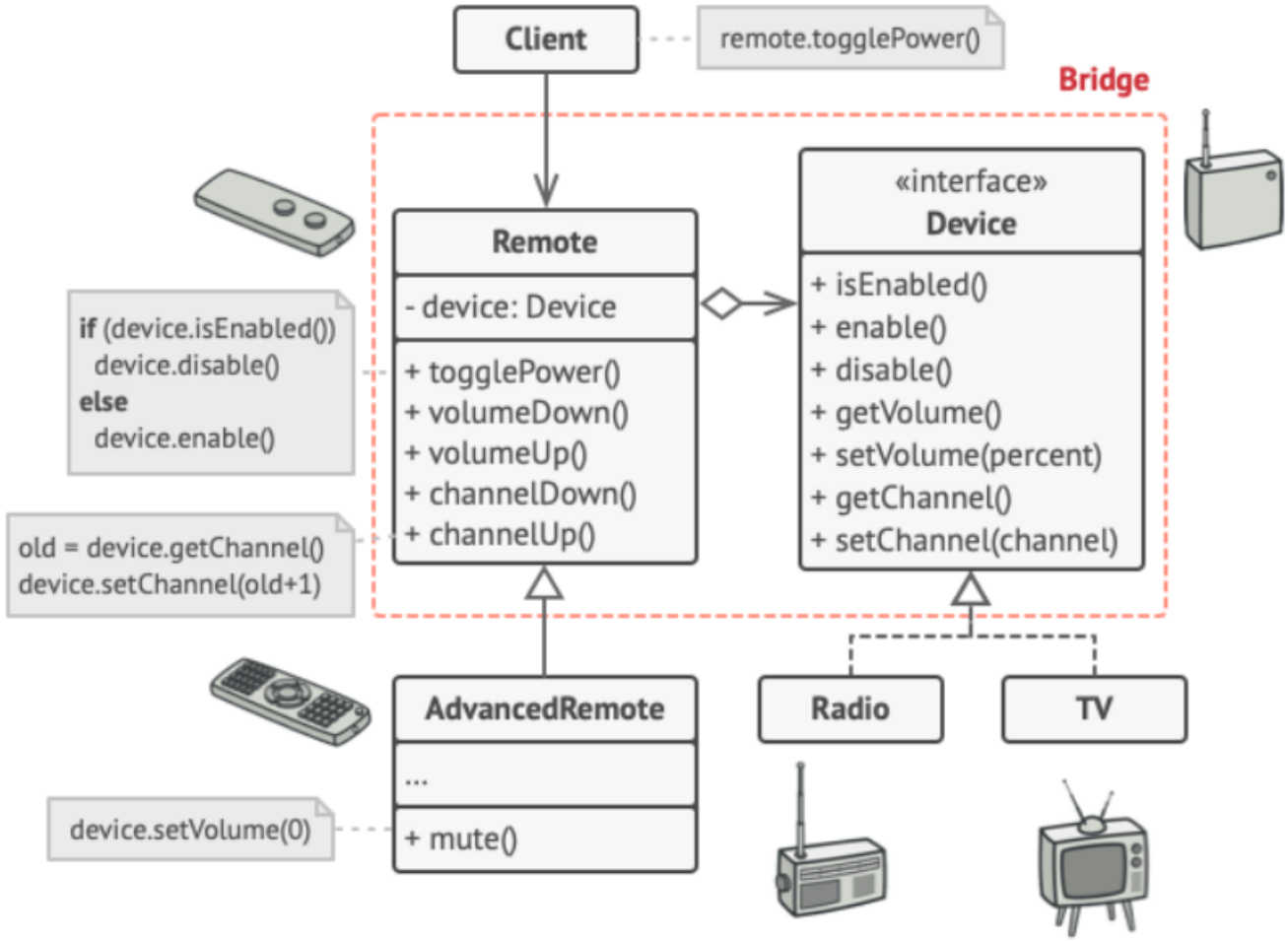


Notiamo che tramite il principio di composizione il vero lavoro viene svolto all'interno delle classi implementazione.

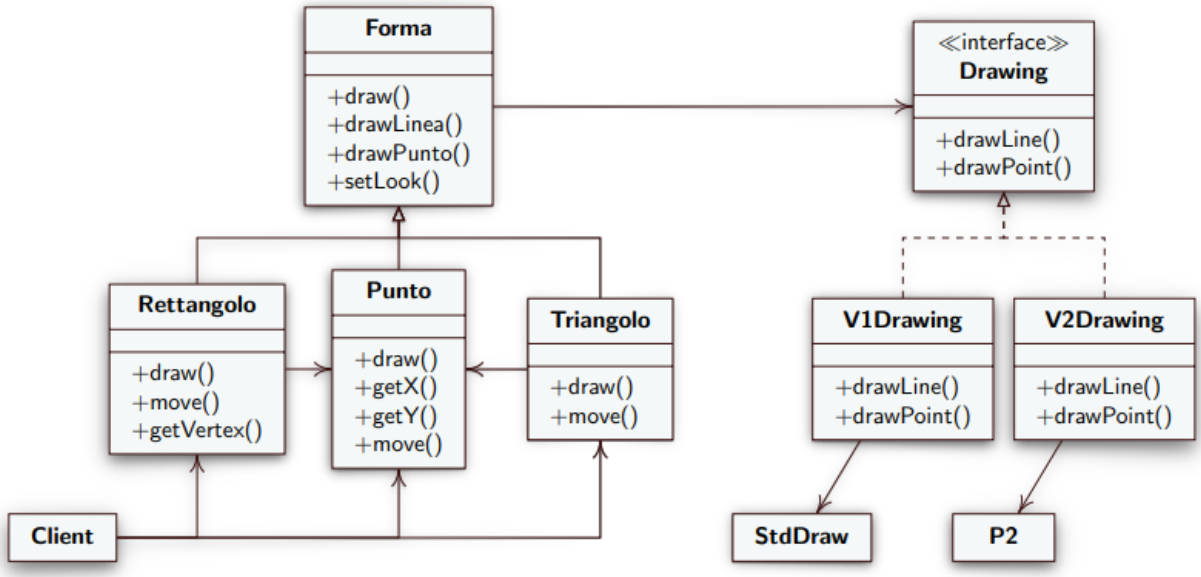
I protagonisti in dettaglio

- Abstraction definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto di tipo Implementor. Abstraction inoltra le richieste del client al suo oggetto Implementor
- RefinedAbstraction estende l'interfaccia definita da Abstraction
- Implementor definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere esattamente ad Abstraction, di solito Implementor fornisce operazioni primitive, mentre Abstraction definisce operazioni di più alto livello basate su tali primitive
- ConcreteImplementor implementa l'interfaccia di Implementor e fornisce le operazioni concrete

Esempi applicativi



La scelta dei ruoli è in alcuni casi arbitraria, in altri invece 'viene da se', in questo caso l'oggetto remote astrae il comportamento dell'oggetto device.

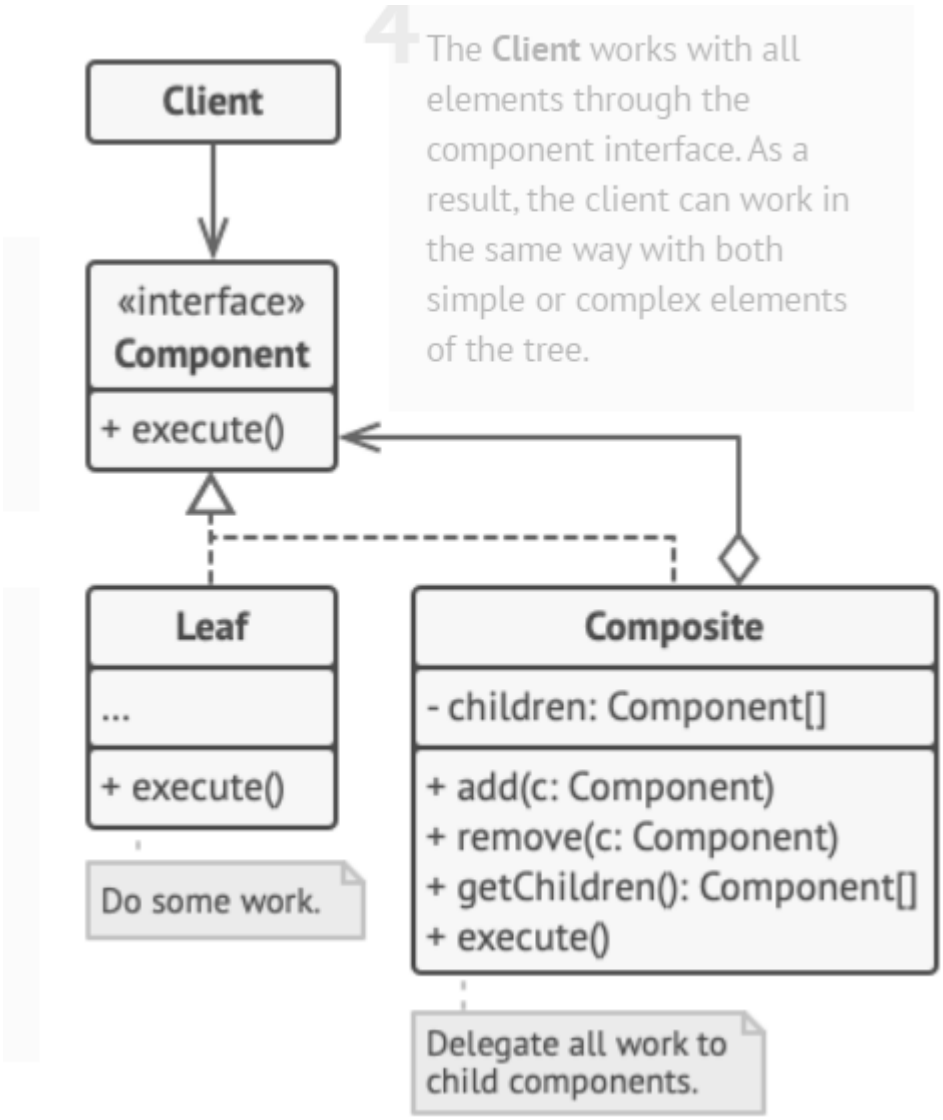


Il pattern rispetta i primi due principi SOLID, si sconsiglia l'uso quando si rischia di complicare una classe che ha un buon livello di coesione. In alcuni casi si può vedere il director del builder come abstractor e il builder come sua implementazione.

Composite
Pattern strutturale che permette comporre oggetti in strutture ad albero in modo tale che si lavori su di esse.

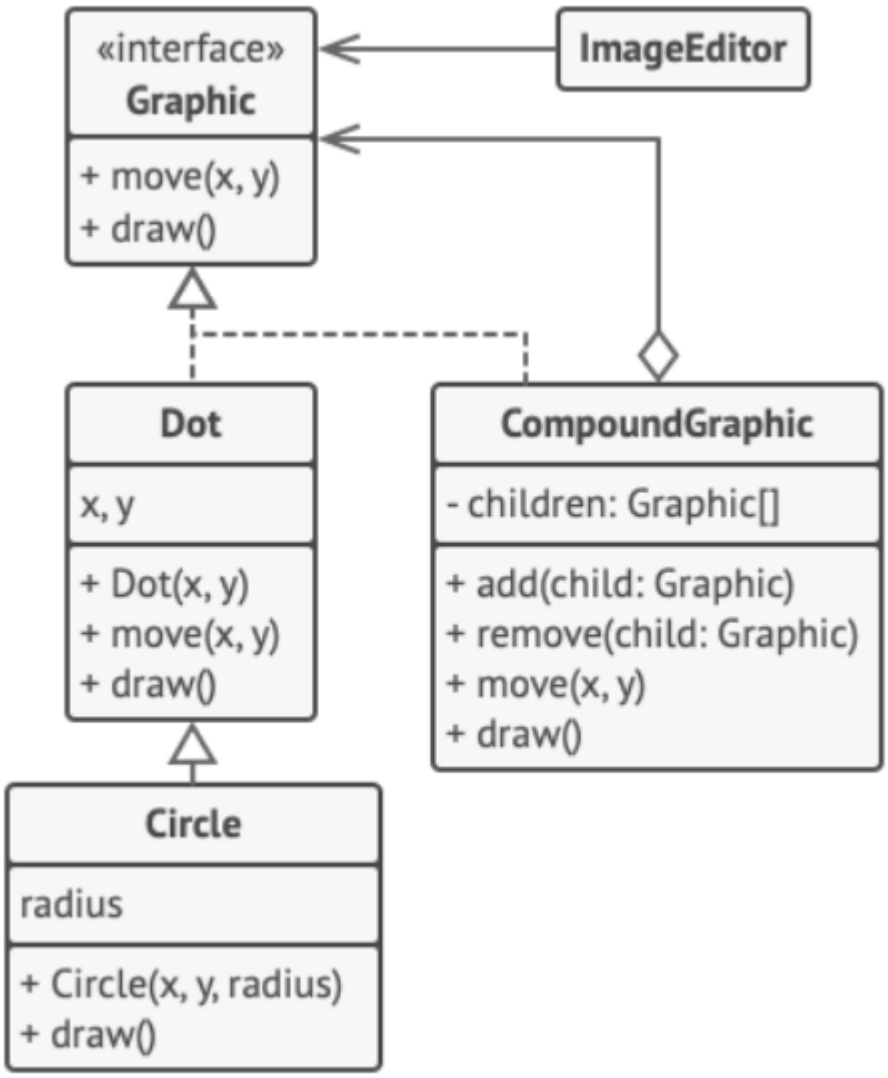
Quale problema risolve?
Spesso quando si lavora con componenti che lavorano tra di loro avere una struttura ad albero semplifica di molto il lavoro

Struttura



Notiamo due classi che implementano il metodo execute, la classe leaf è un nodo foglie, dalla proprietà di composizione notiamo che tutto il lavoro è delegato a questa classe, mentre il composite ha il ruolo di container.

Esempio applicativo

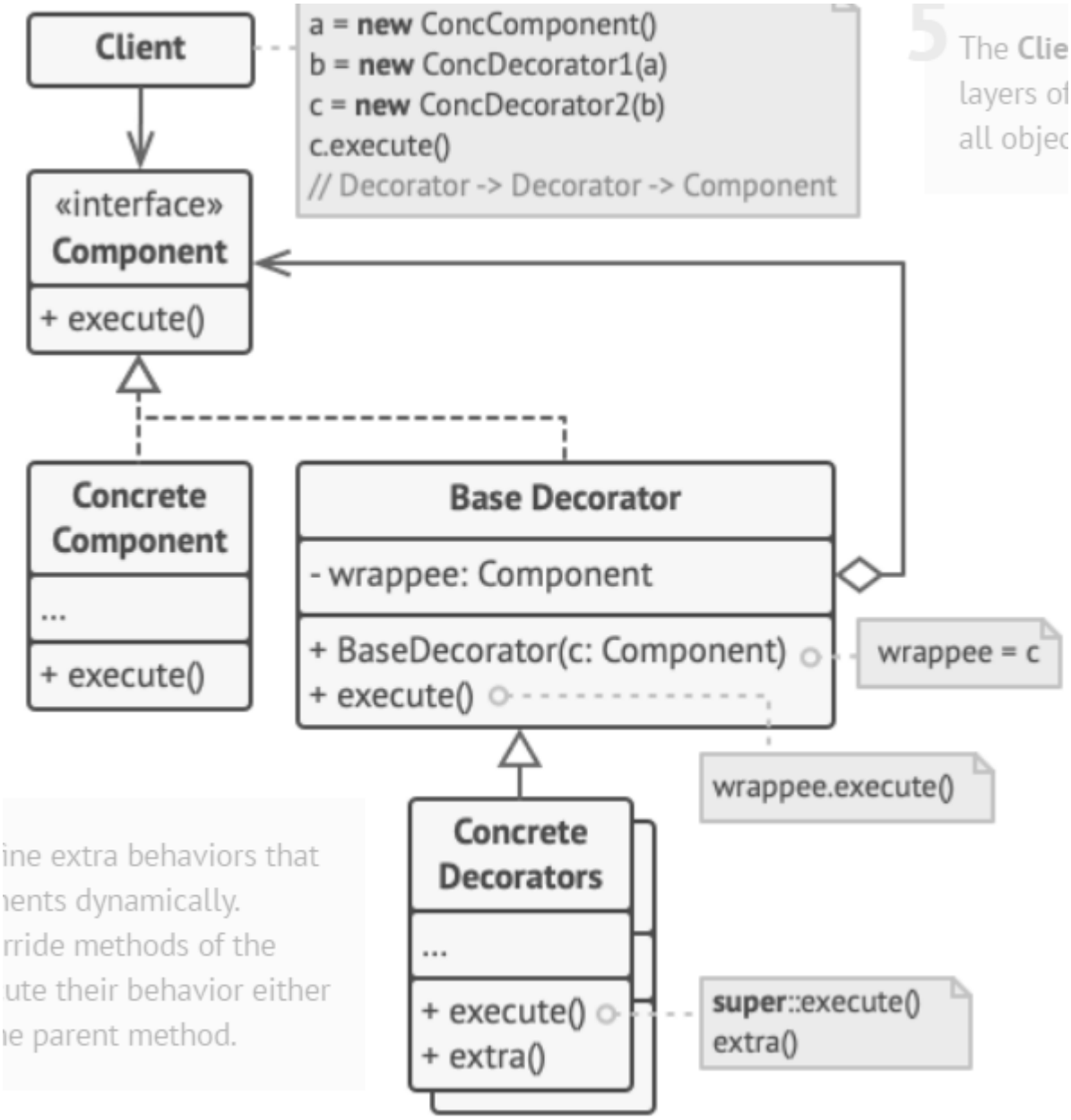


Notiamo che il nodo Leaf in questo caso è Circle che estende il comportamento della classe Dot.

Questo pattern viene spesso usato generalizzando troppo il comportamento dell'interfaccia, risultando meno leggibile.

Decorator
Pattern strutturale che permette di aggiungere nuove funzionalità all'oggetto in maniera dinamica.

Struttura

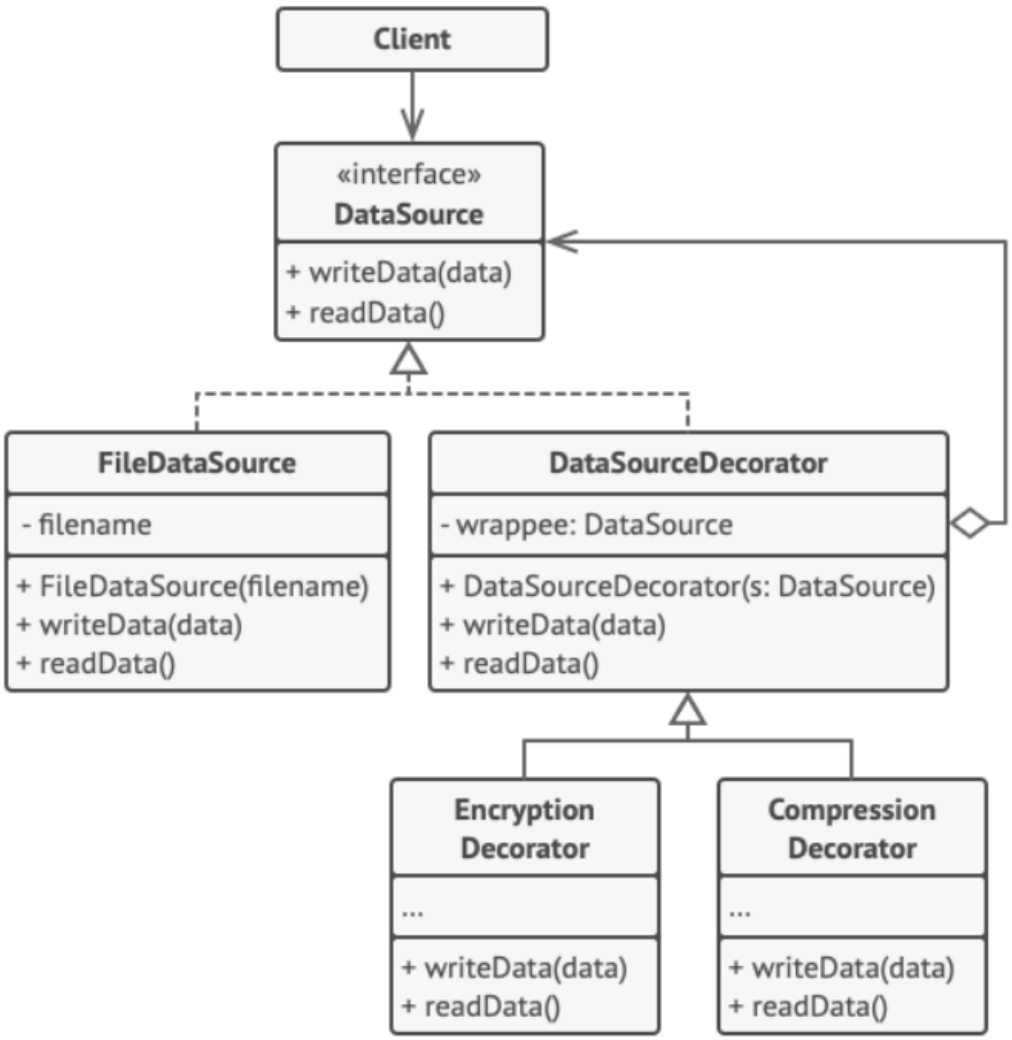


Struttura molto simile al composite, anche in questo caso attraverso la composizione abbiamo una delega di lavoro a al componente concreto, che in questo caso viene 'decorato' con nuove funzionalità

Perchè molti pattern insistono con la delega tramite composizione?

Prefer using composition over inheritance when you need to reuse code and the types don’t have an “is a” relationship. Additionally, if your types don’t have an “is a” relationship but you need polymorphism, use composition with interfaces.

Esempio applicativo



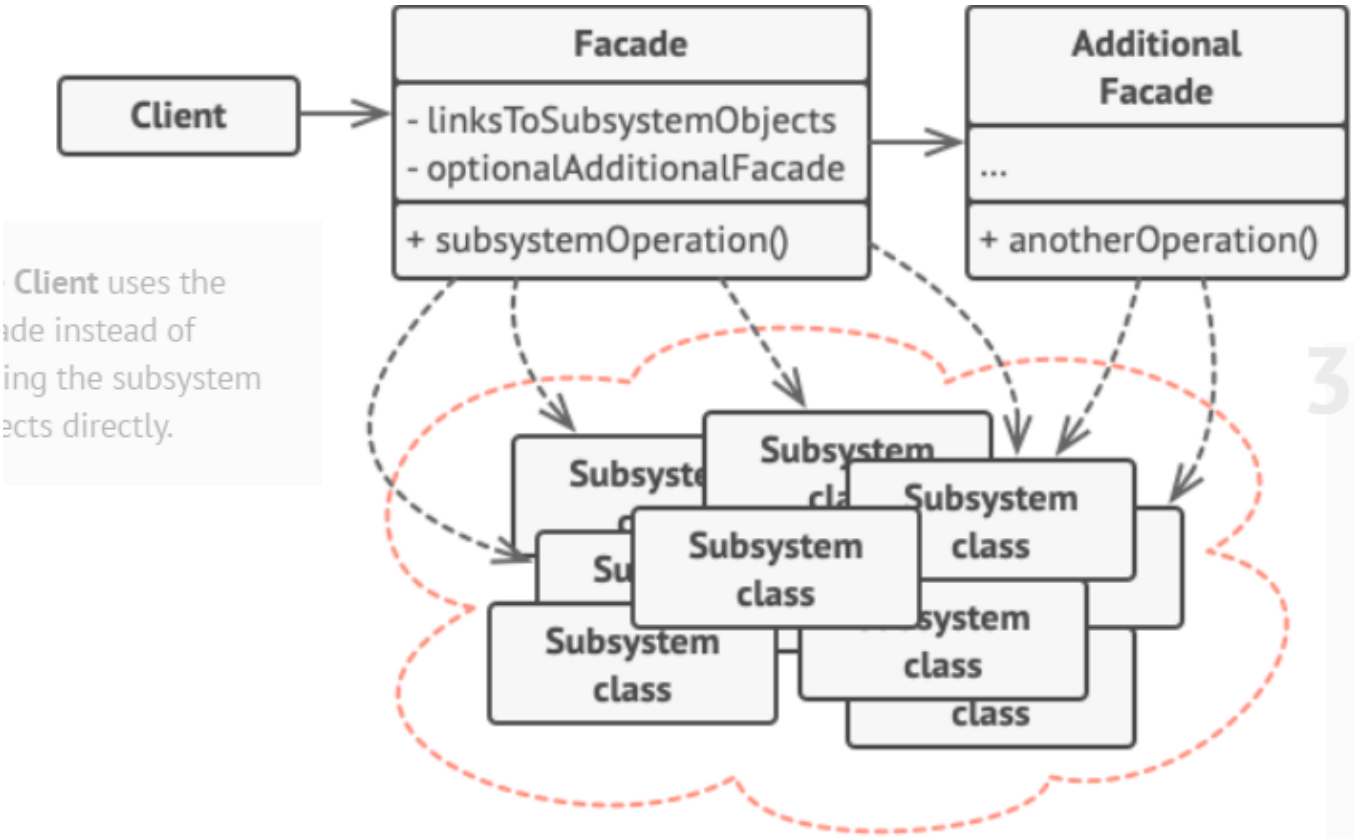
The encryption and compression decorators example.

In questo caso FileDataSource viene decorato con le funzionalità di 'Encryption' e 'Compression'.

Façade

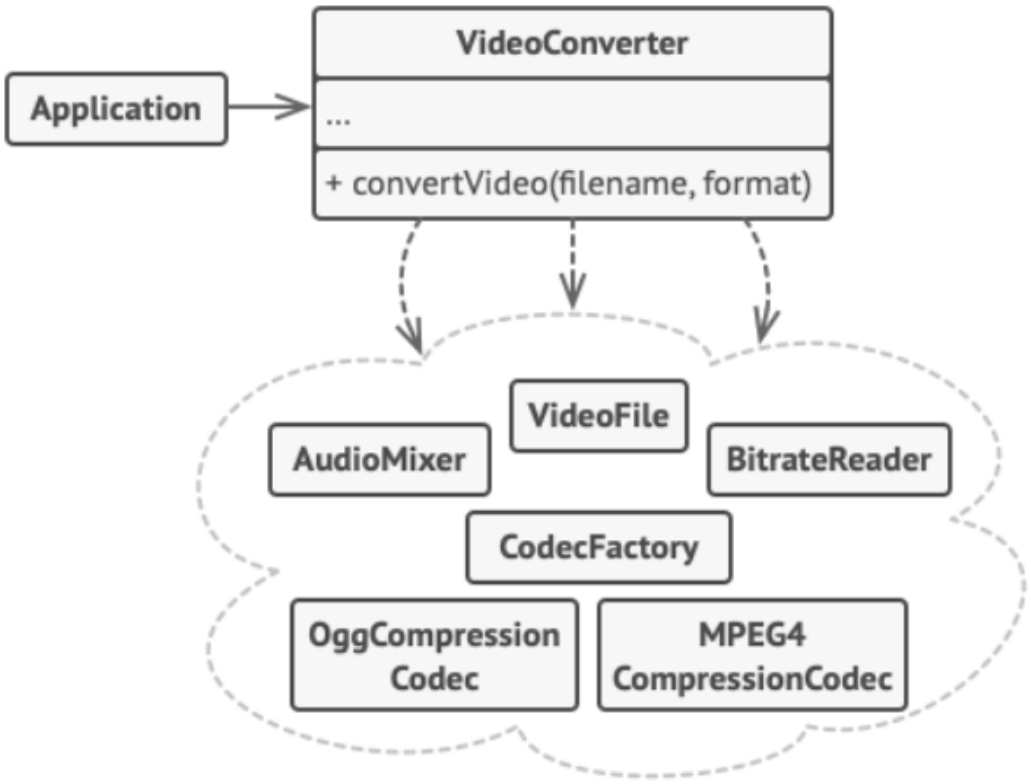
Pattern strutturale che fornisce un' interfaccia semplificata per lavorare con un set di classi (o una libreria).

Struttura



Notiamo che la classe façade fornisce dei campi che linkano alle sottoclassi che la utilizzano, il façade addizionale è facoltativo

Esempio applicativo



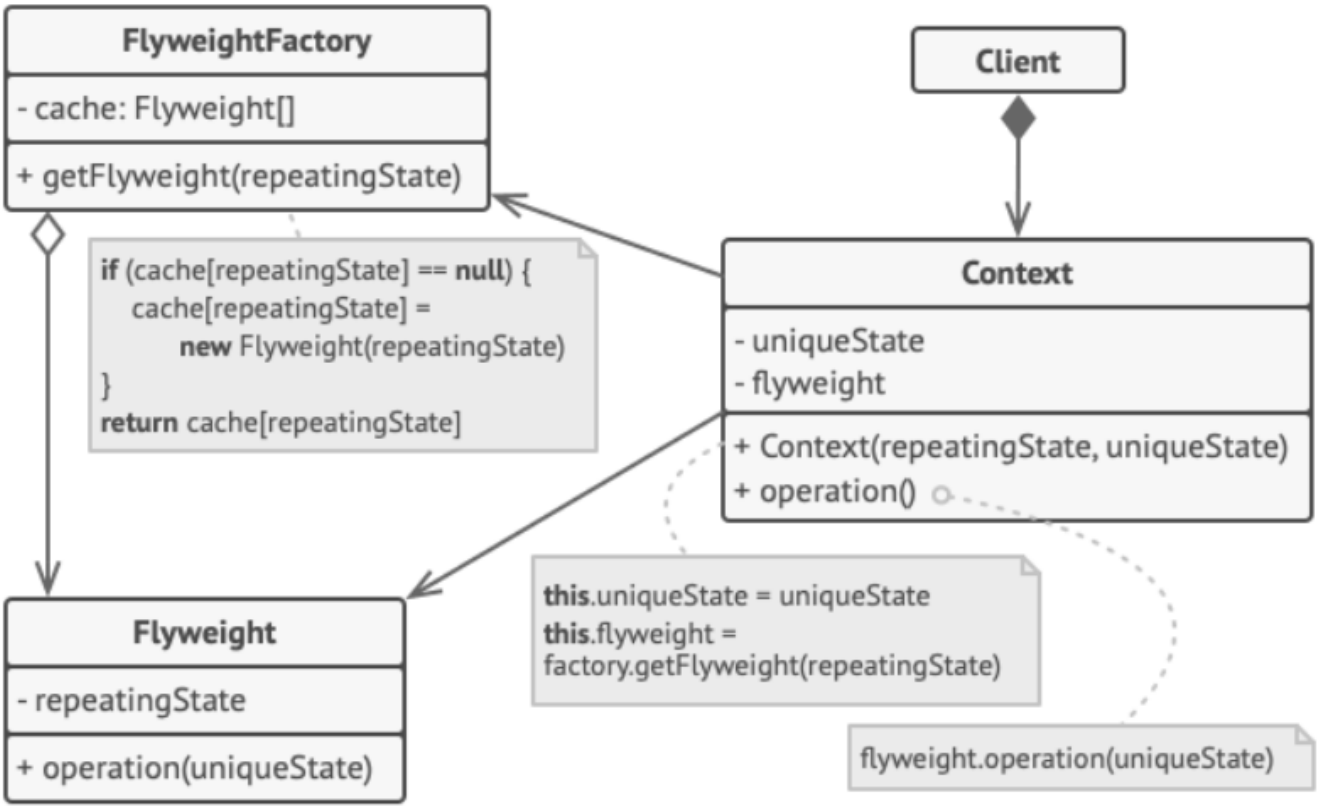
Bisogna prestare attenzione affinché il façade non diventi troppo importante all'interno del codice, si veda a proposito: https://en.wikipedia.org/wiki/God_object

Flyweight
Pattern strutturale che permette di allocare più oggetti nella ram attraverso la condivisione di parti in comune che descrivono lo stato di un oggetto

Applicabilità
Il pattern flyweight richiede alcuni requisiti minimi per poter essere utilizzato:

- Applicazione che necessita di un gran numero di oggetti
- Costi di memorizzazioni molto alti
- possibilità di condividere oggetti

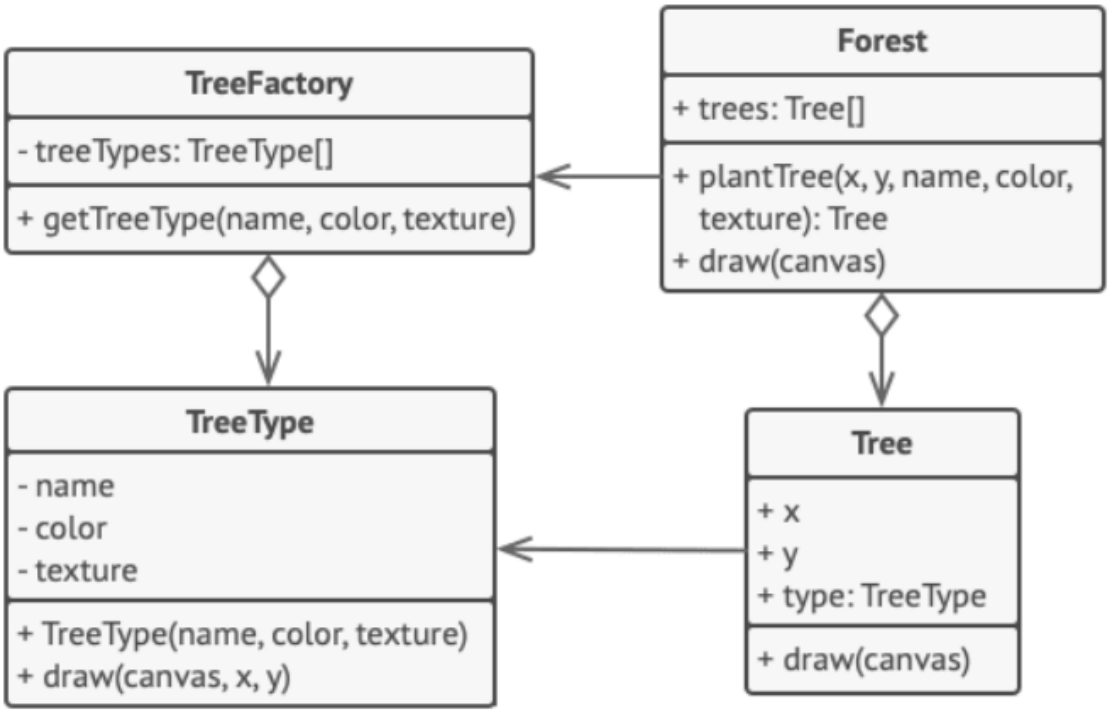
Struttura



Notiamo come la classe contesto e la classe flyweight lavorino in congiunta, in particolare:

- Context possiede lo stato unico dell'oggetto
- Flyweight possiede lo stato ripetuto, ovvero quello in comune fra i vari oggetti
  Inoltre la factory può essere considerata un metodo di utilità che memorizza nella cache una lista di flyweights

Esempio applicativo



In questo caso abbiamo alcuni flyweight che contengono i vari tipi di alberi, riducendo lo spreco di creazione di alberi diversi ogni volta che il client lo richiede.

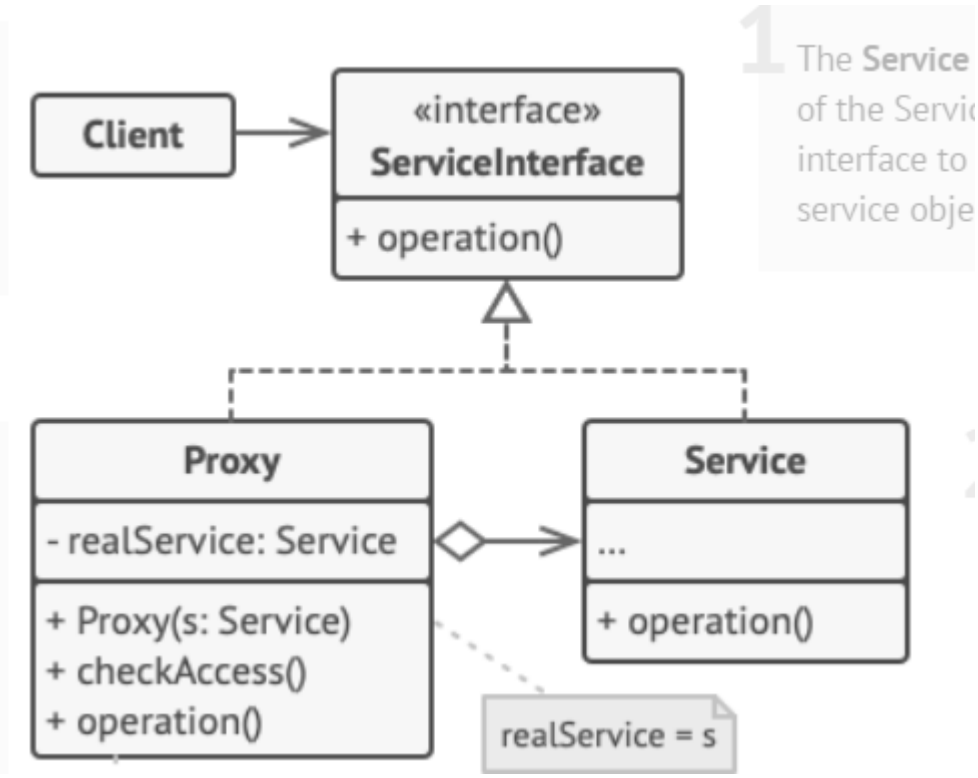
Alcuni difetti del flyweight derivano dalla sua complessità e dall'uso eccessivo di CPU

Proxy
Pattern strutturale che permetta di creare un 'sostituto' di un oggetto.

I vari tipi di proxy

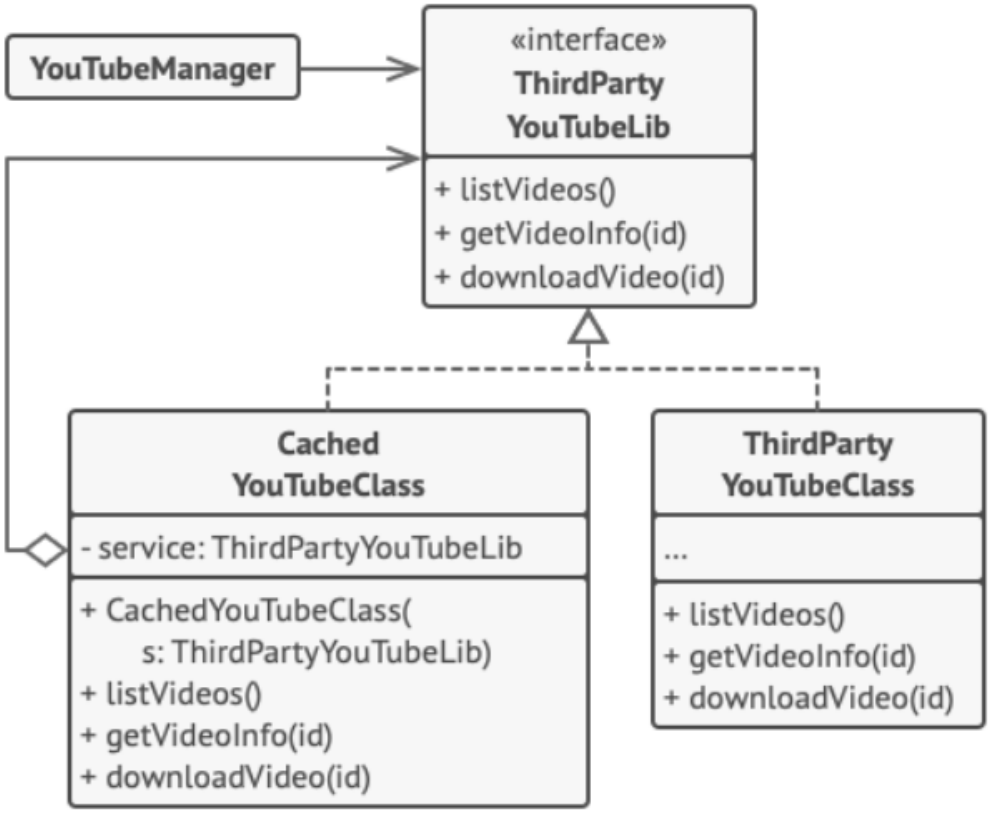
- Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
- Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
- Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.
- Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.
- Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.

Struttura



La classe Proxy ha un campo di riferimento che punta a un oggetto Service. Dopo che il proxy ha terminato l'elaborazione (ad esempio, inizializzazione lenta, registrazione, controllo degli accessi, memorizzazione nella cache, ecc.), passa la richiesta all'oggetto di servizio.

Un esempio con lazy initialization e caching



Caching results of a service with a proxy.

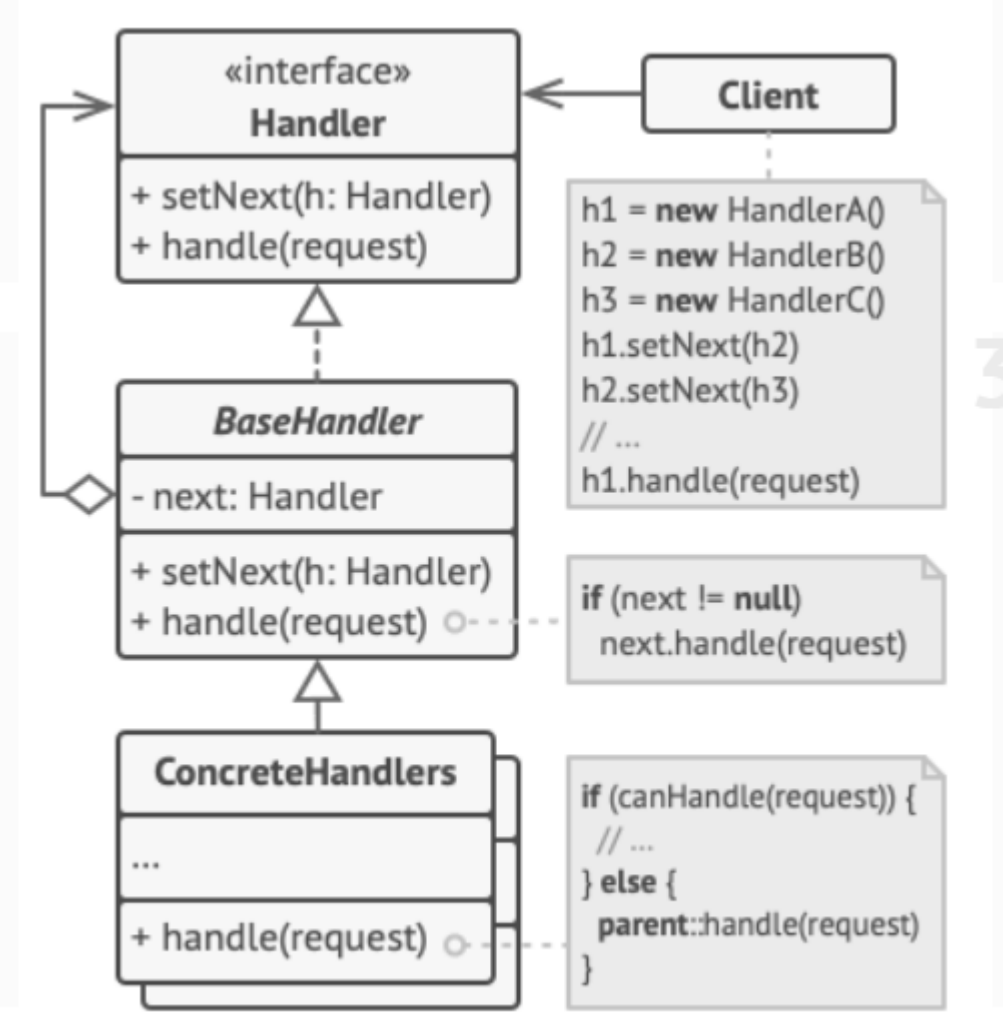
La classe CachedYoutubeClass offre i servizi di proxy e subito dopo il cliente (YoutubeManager) accede al servizio vero e proprio

Il pattern proxy permette l'implementazione di vari servizi, la conseguenza è che questi servizi potrebbero essere ritardati (delayed)

Pattern comportamentali

Chain of Responsibility
Pattern comportamentale che permette che una richiesta venga gestita da diversi handler

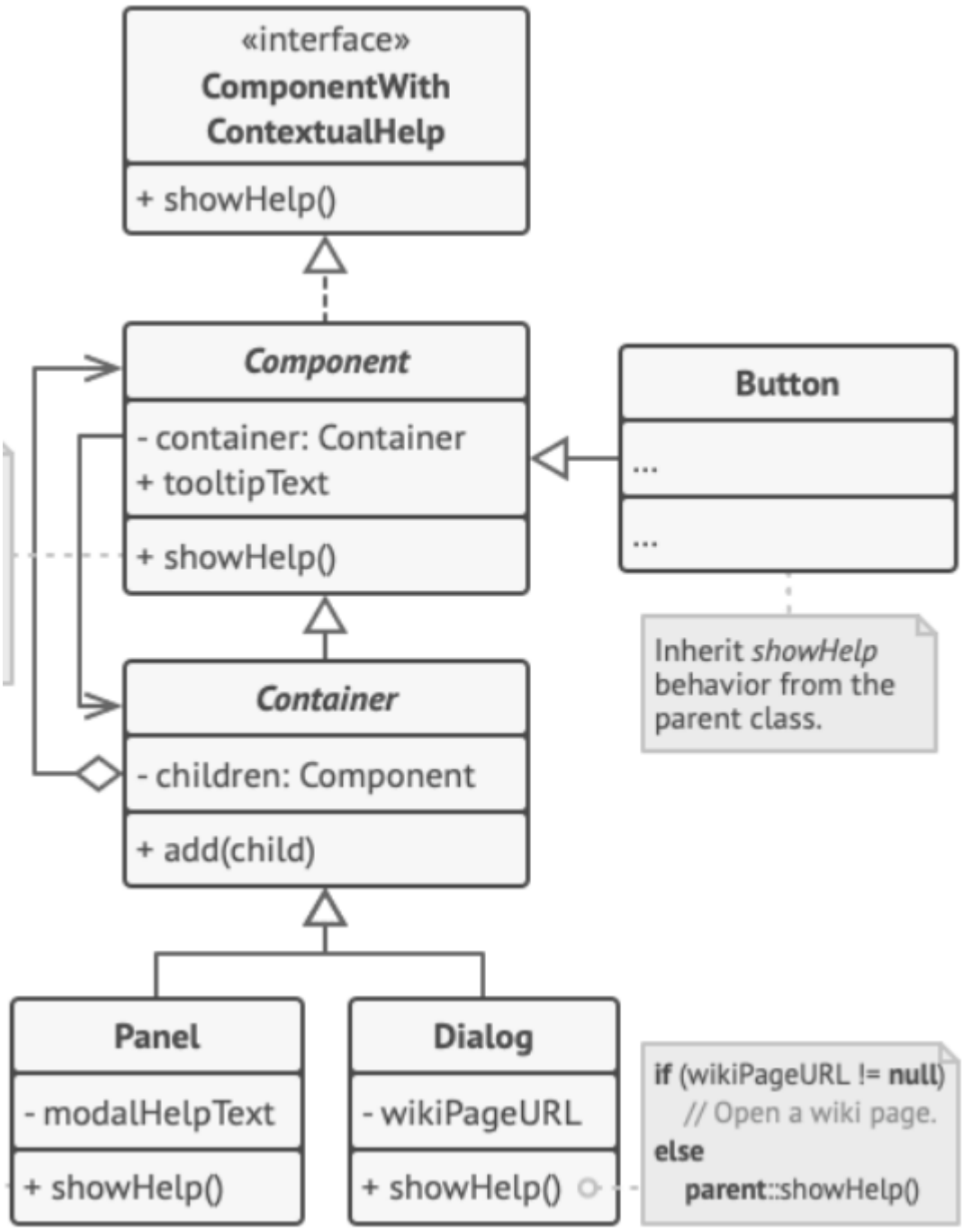
Struttura



Notiamo che esistono due tipi di handler:

1. Base Handler che contiene del codice in comune a tutti i concrete handlers (opzionale)
2. Concrete Handlers, sceglie se gestire la richiesta e se passarla all'handler successivo

Esempio applicativo con Composite Pattern



Il CoR risulta inefficiente nei casi in cui alcuni handlers non vengono utilizzati

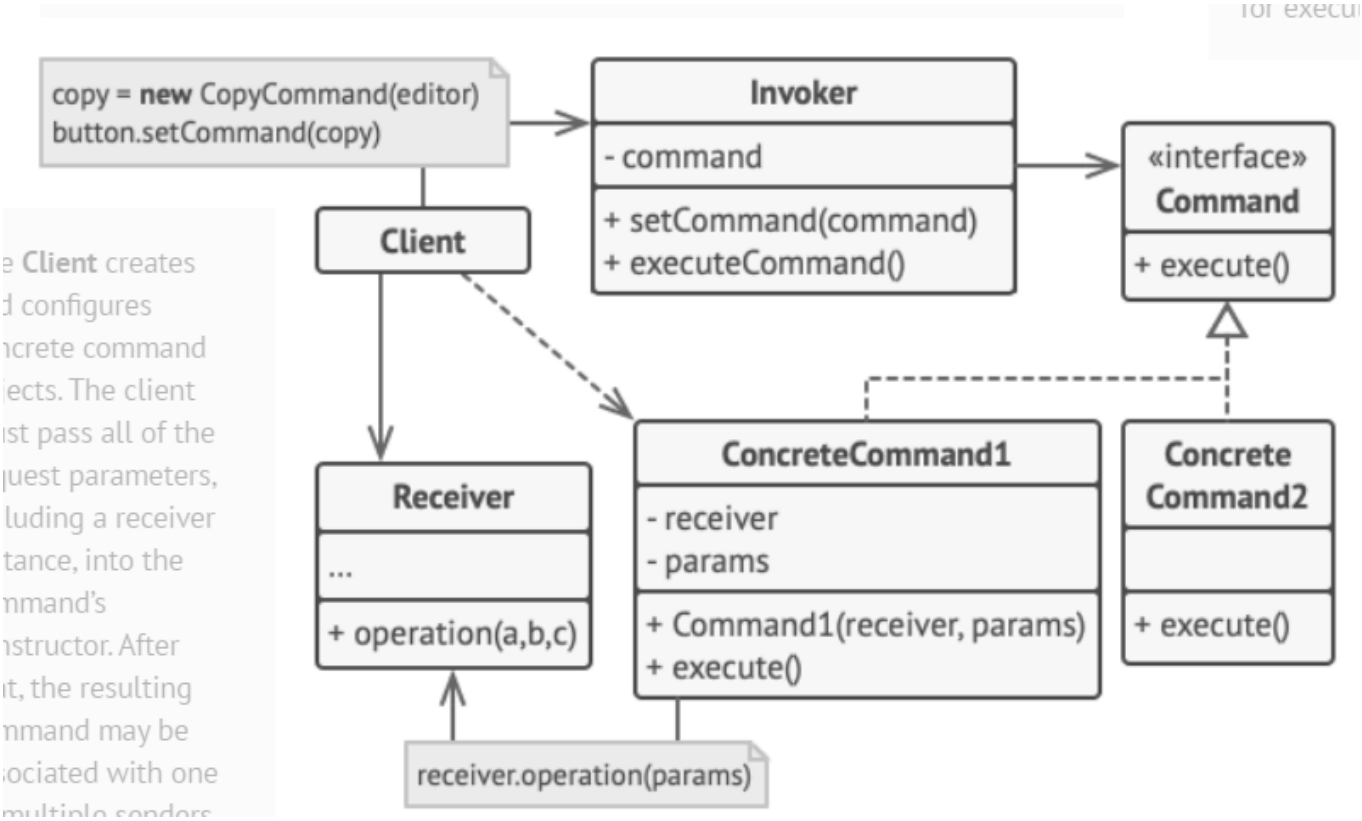
Command
Pattern comportamentale che trasforma richieste in oggetti stand-alone

Il Command sfrutta la corretta pratica ingegneristica di separazione degli interessi (separation of concerns), ovvero? non è la stessa cosa del Single Responsibility? Più o meno:

> **Separation of Concerns vs Single Responsibility Principle**

SoC and SRP are closely related, and adhering to both principles contributes to better software design.
SoC addresses the high-level architecture of a system and how different concerns are organized into separate modules or layers.
SRP focuses on the design of individual classes within those modules, ensuring that each class has a clear and singular responsibility.

Struttura

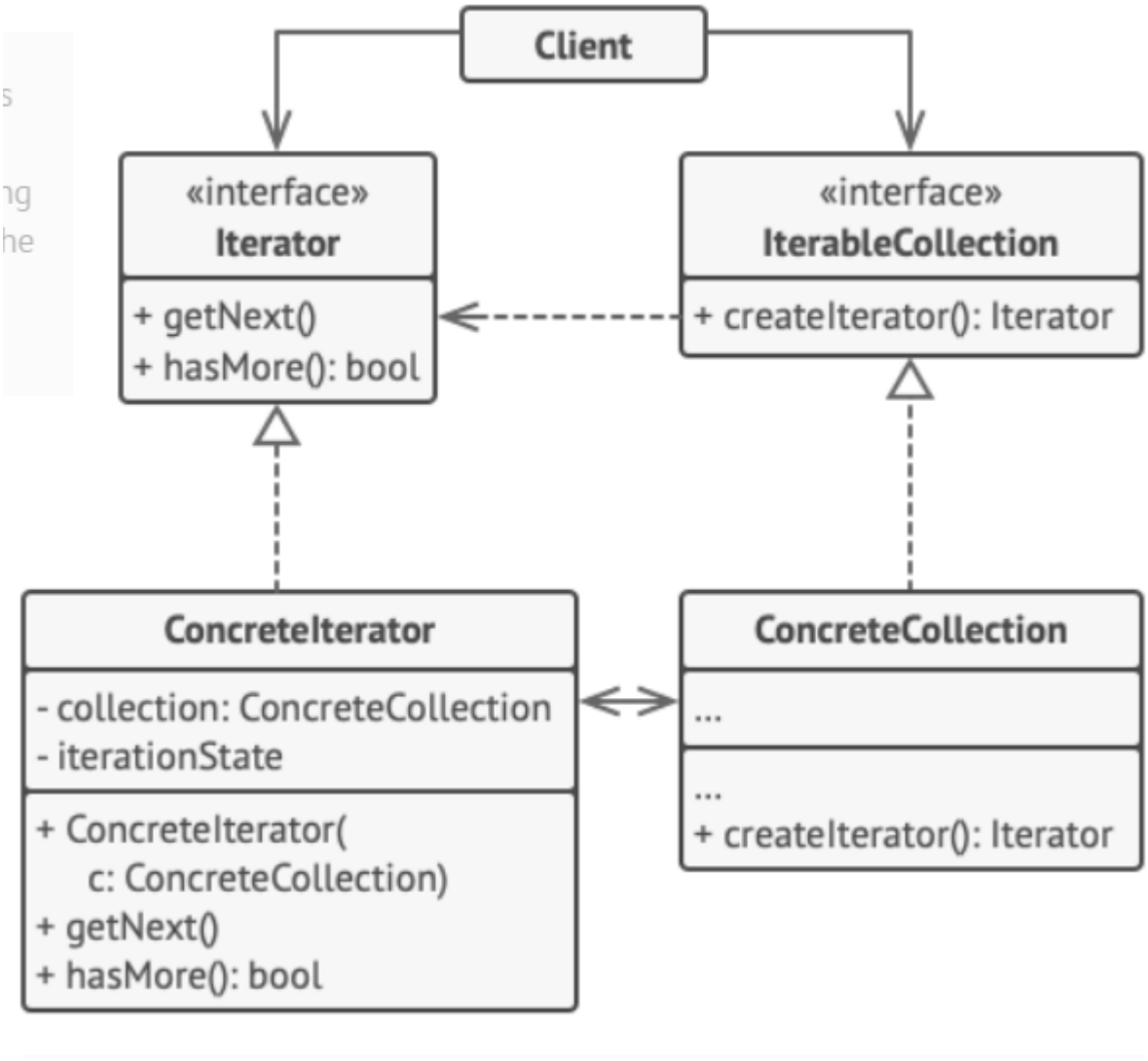


Due attori risaltano in questa struttura

- Invoker: inizializza la domanda e interagisce con il receiver tramite l'interfaccia Command
- Receiver fa il lavoro vero e proprio poiché è la classe che conosce la 'business logic'.

Il command può essere visto come la versione debole del Visitor

Iterator
Pattern comportamentale che permette di iterare tra gli oggetti nascondendo l'implementazione completa

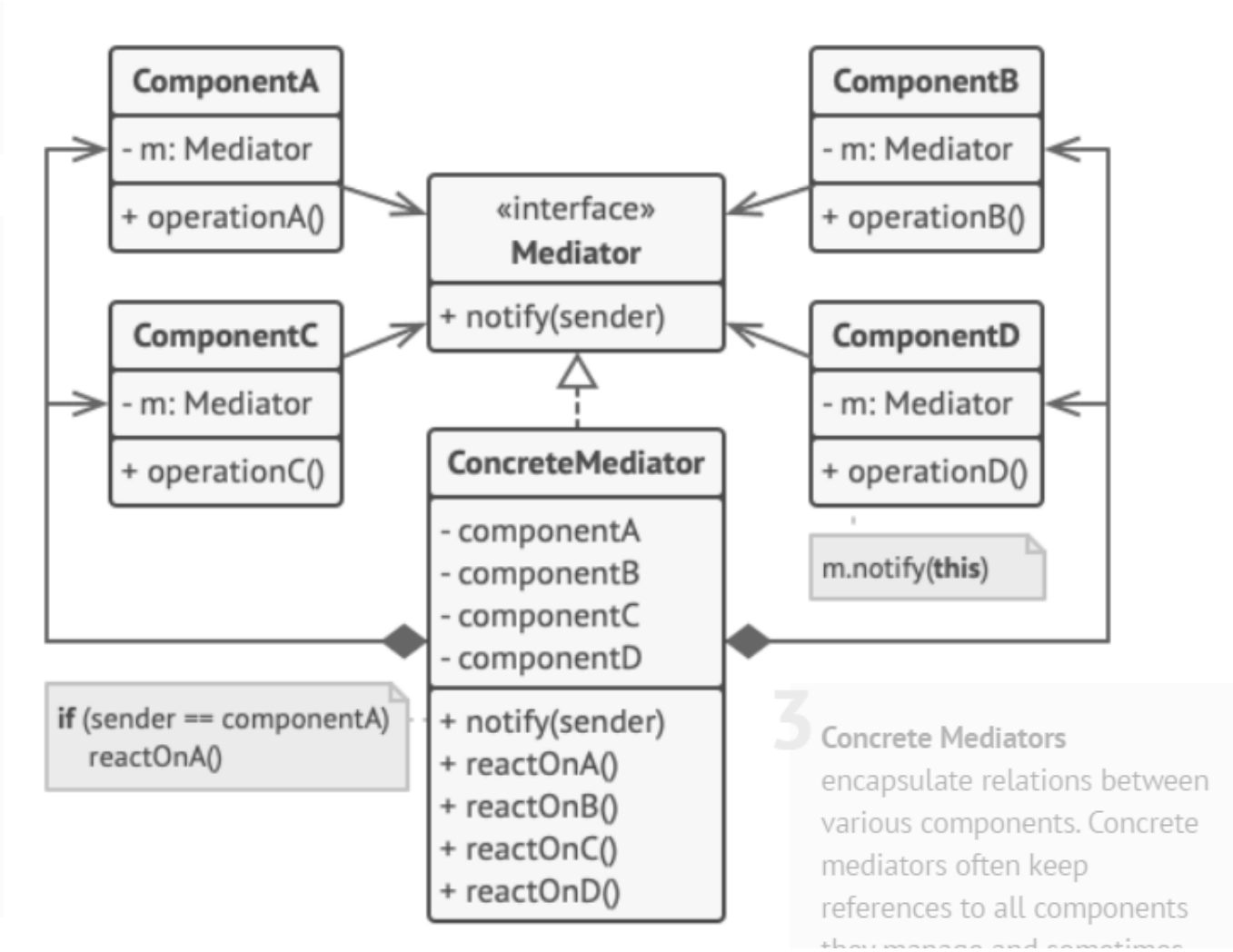


Notiamo che l'interfaccia IterableCollection implementa l'altra interfaccia Iterator in modo tale da permettere di creare varie versioni dell'iterator, la classe ConcreteIterator fornisce invece vari modi di costruire un iterator

Molto spesso questo pattern è inutile poiché esistono già delle collezioni predefinite

Mediator
Pattern comportamentale che permette di ridurre la dipendenza fra oggetti restringendo la loro comunicazione

Struttura



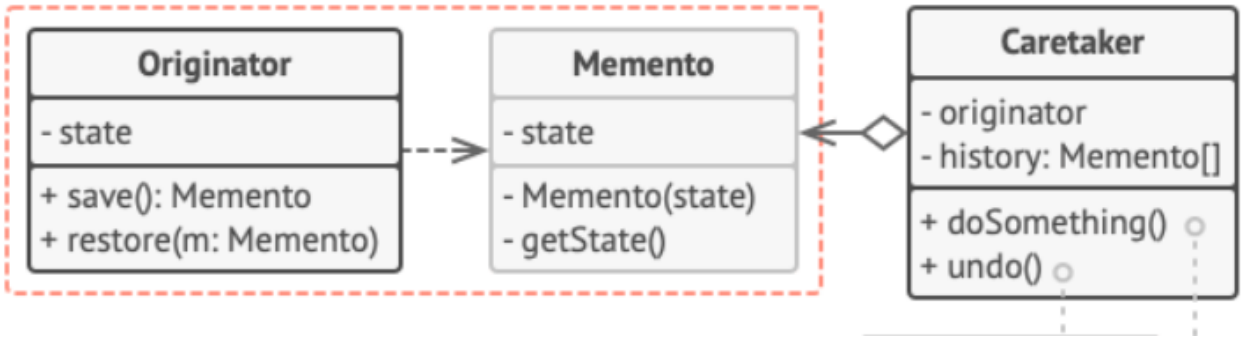
Ognuna delle varie componenti contiene della 'business logic' , mentre l'interfaccia mediator contiene un singolo metodo di utilità che serve a mettere in comunicazione i vari componenti, sarà compito dell'oggetto ConcreteMediator cercare di capire come questi oggetti dovranno comunicare.

Cosi come il façade, si rischia di far diventare troppo importante l'oggetto che lavora da mediatore,(God Object).

Memento
Pattern comportamentale il cui utilizzo principale è la il salvataggio di stati precedenti dell'oggetto

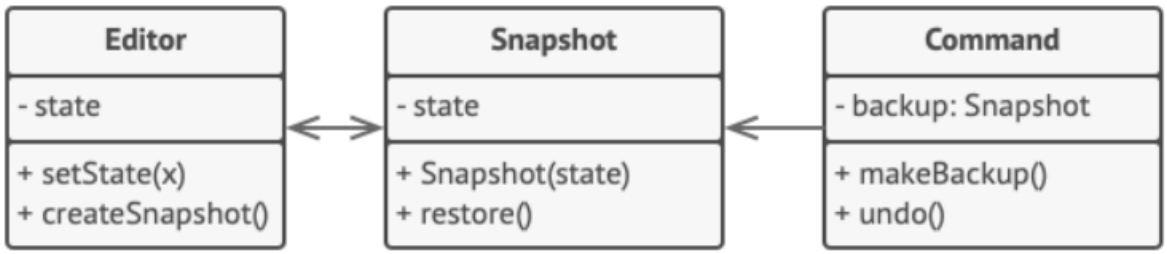
Il pattern Memento è in grado di catturare lo stato di un oggetto cercando di mantenere l'incapsulazione e di riproporne lo stati quando richiesto

Struttura
Esistono diversi tipi di implementazione, la maggior parte si basa sull'utilizzo di classi innestate



Notiamo che l'oggetto Originator è in grado di catturare il proprio stato e passarlo all'oggetto Memento, mentre la classe di utilità Caretaker tiene traccia dei vari snapshot

Esempio applicativo



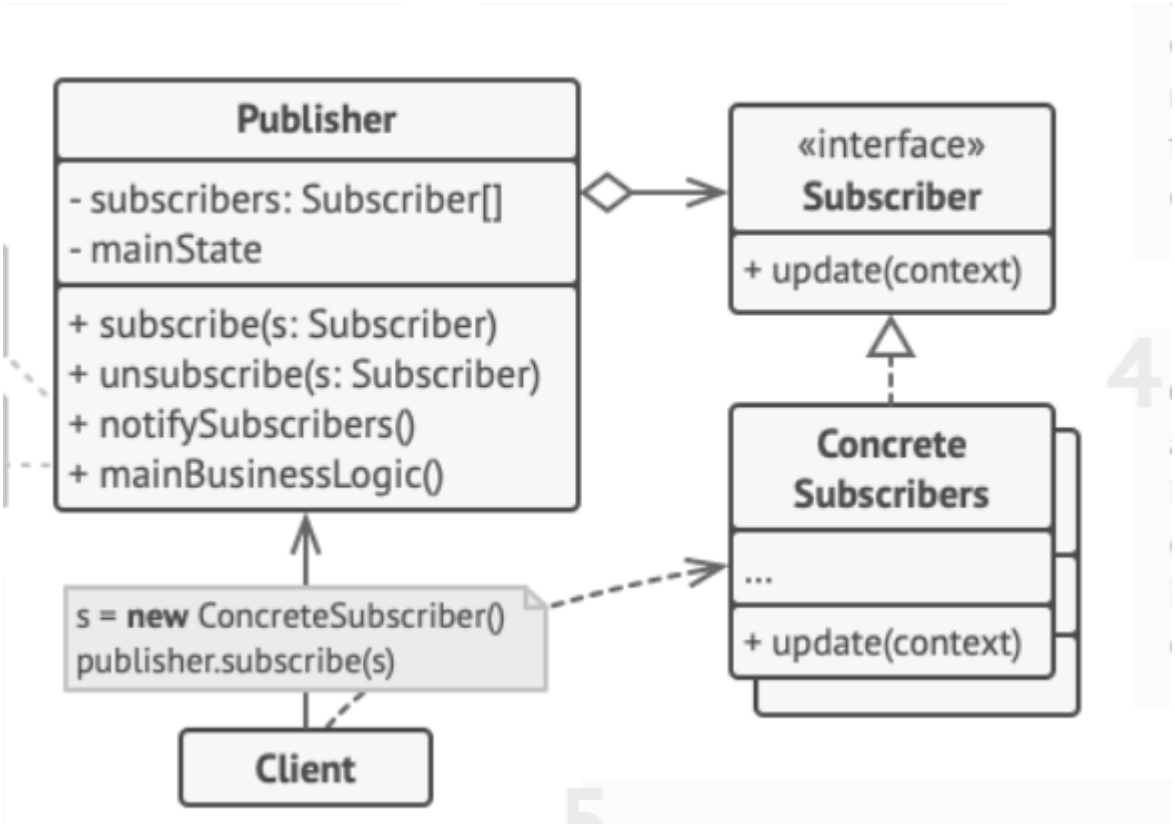
L'applicazione più comune di questo pattern riguarda l'uso di editor.

L'utilizzo di questa implementazione potrebbe richiedere un grosso utilizzo di RAM quando il numero di snapshot risulta troppo elevato.

Observer
Pattern comportamentale che modello il comportamento di una iscrizione da parte di più classi a un server (più in generale a qualcuno che deve comunicare qualcosa)

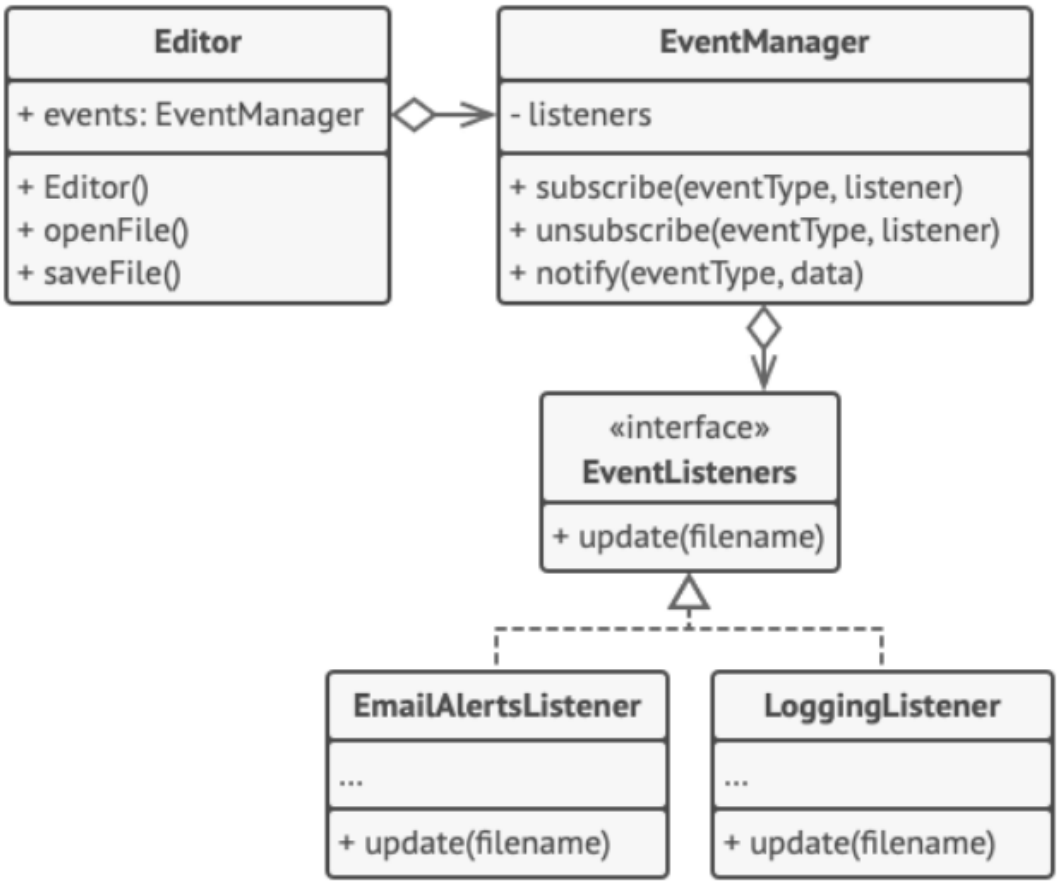
Questo Pattern è molto utile quando più oggetti devono rimanere in ascolto.

Struttura



La classe Publisher invia una notifica di cambiamento di stato ai subscriber e quest'ultimi agiscono di conseguenza

Esempio applicativo

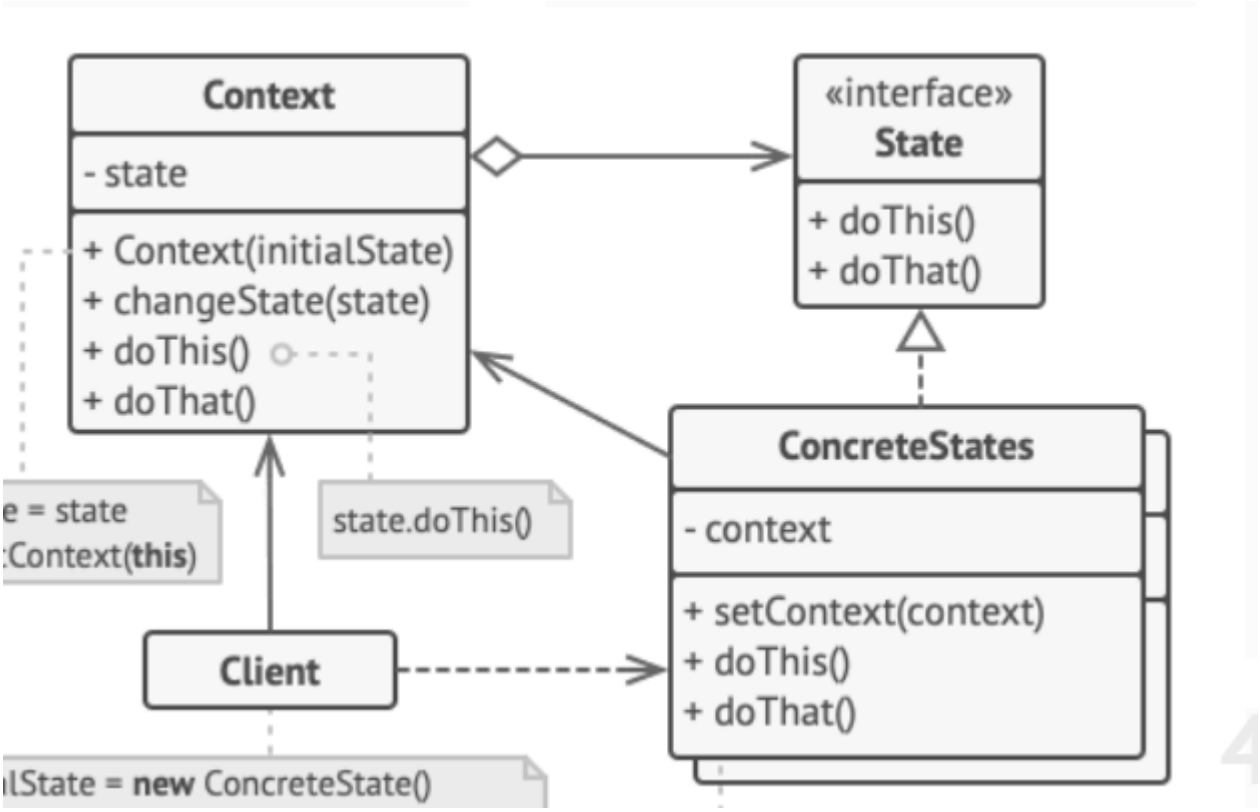


Notifying objects about events that happen to other objects.

Un editor potrebbe servirsi di alcuni oggetti che rimangono in ascolto per modificare on oggetto di tipo file in base al tipo di azione richiesta

State
Pattern comportamentale che permette ad un oggetto di alterare il proprio comportamento in base ai cambiamenti interni dello stato dell'oggetto.

Struttura



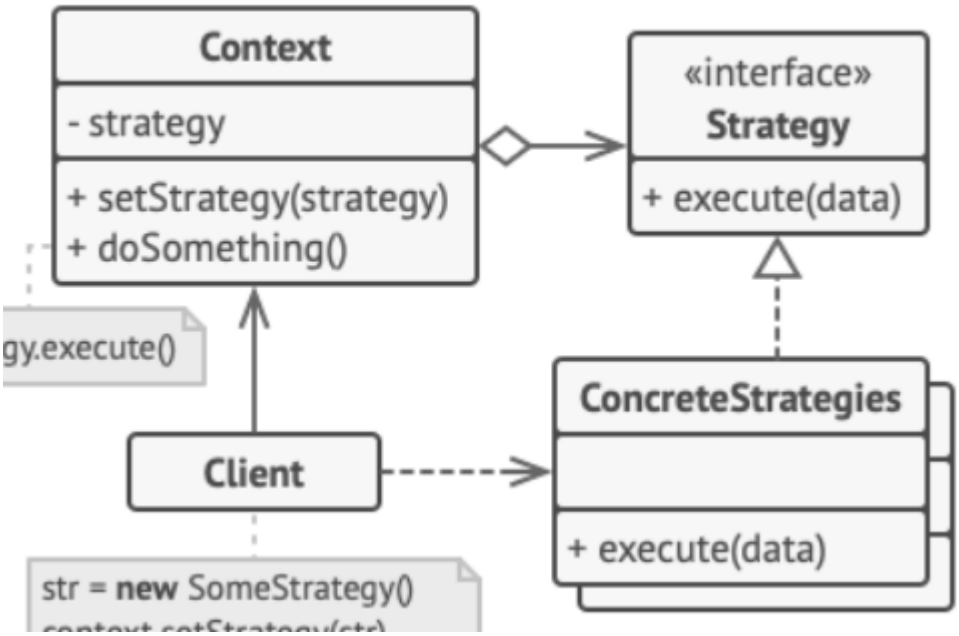
Notiamo che l'interfaccia State viene implementata dalle varie classi concrete che implementano la propria soluzione in base allo stato che fornisce la classe Context.
Inoltre gli stati concreti aggiornano il contesto tramite il metodo setContext()

Questo pattern non presenta troppe controindicazione, potrebbe essere non necessario ('overkill') nei casi in cui gli stati dell'applicazione sono pochi.

Strategy
Pattern comportamentale che permette di definire una famiglia di algoritmi in classi separate e interscambiabili.

Questo pattern risulta molto utile nei casi in cui un dato problema potrebbe richiedere diverse soluzioni.

Struttura



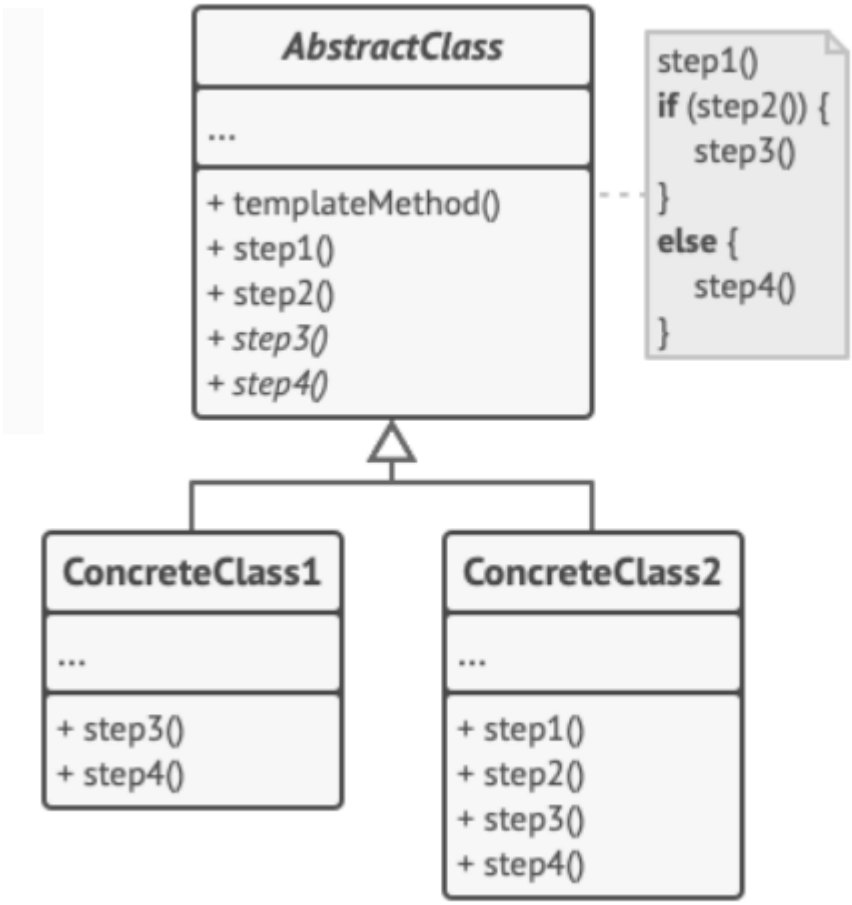
La classe Context definisce la strategia e le varie classi concrete la eseguono

Questo pattern va usato solo nei casi in cui il Client che si interfaccia al pattern conosce bene la differenza fra i vari algoritmi e quando conviene usare uno rispetto all'altro.

Template Method

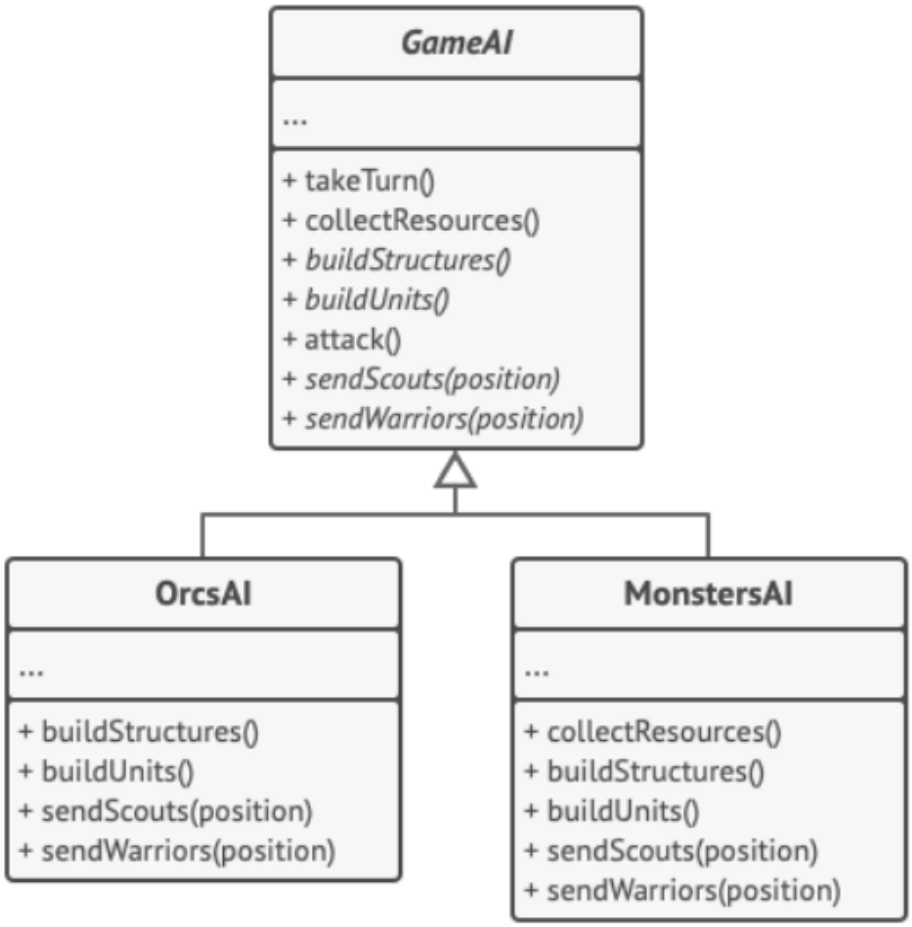
Pattern comportamentale che permette di definire lo scheletro di un algoritmo, lasciando alle sottoclassi la responsabilità di scegliere in che modo utilizzarlo.

Struttura



Notiamo che non tutti i metodi necessitano di un override.

Esempio applicativo

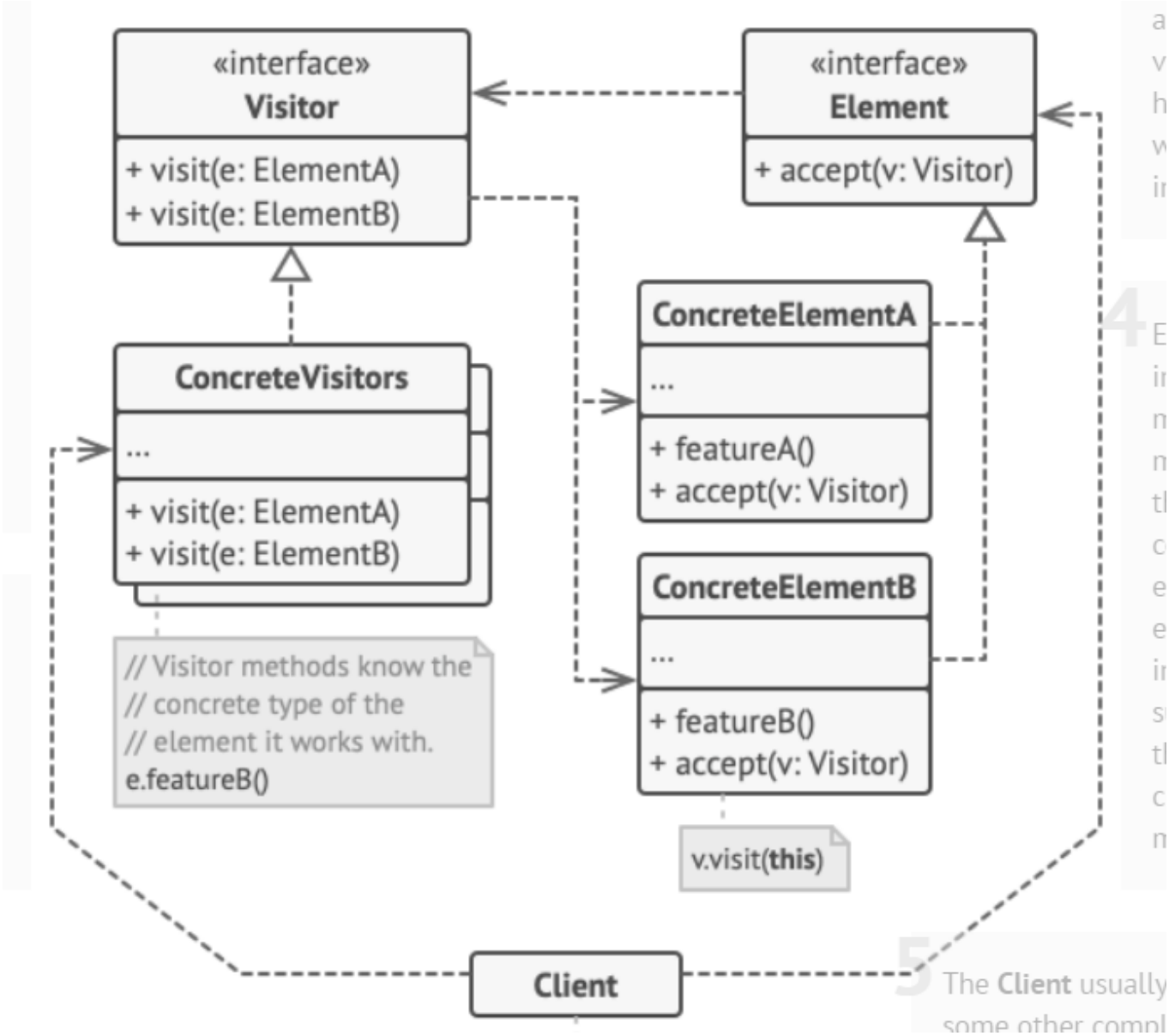


OrcsAI e MonsterAI scelgono alcuni metodi di cui fare override.

Bisogna fare attenzione all'overiding dei metodi della classe astratta poichè potrebbe violare il principio di Liskov.

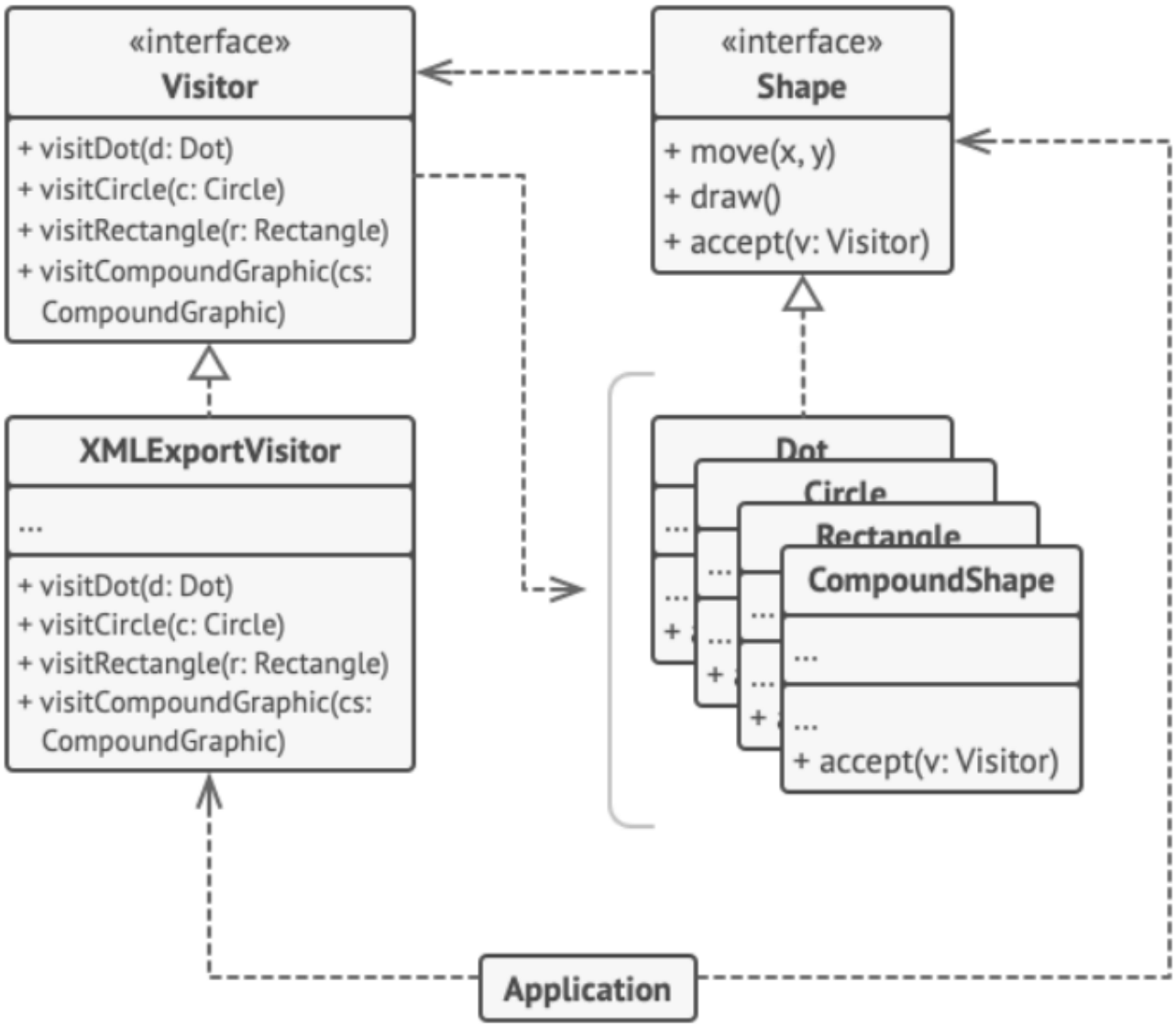
Visitor
Pattern comportamentale che permette di separare gli algoritmi dagli oggetti.

Struttura



Vediamo come i due oggetti Element e Visitor sono fortemente interfogliati, infatti gli oggetti Element estendono il comportamento dell'interfaccia Visitor e gli oggetti Visitor estendono il comportamento dell'interfaccia Element.

Esempio applicativo


«interface»
Visitor
+ visitDot(d: Dot)
+ visitCircle(c: Circle)
+ visitRectangle(r: Rectangle)
+ visitCompoundGraphic(cs: CompoundGraphic)
«interface»
Shape
+ move(x, y)
+ draw()
+ accept(v: Visitor)
XMLExportVisitor
...
+ visitDot(d: Dot)
+ visitCircle(c: Circle)
+ visitRectangle(r: Rectangle)
+ visitCompoundGraphic(cs: CompoundGraphic)
Dot
Circle
Rectangle
CompoundShape
...
...
+ accept(v: Visitor)
Application