



Design Patterns

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Sommario

1 Riuso

2 Approcci

Riuso - generalità

Riuso tipico approccio ingegneristico alla costruzione di sistemi.
Sistemi risultano da integrazione di sottosistemi spesso non “banali”.

Motivazioni e benefici legati al riuso:

- ridotti costi di sviluppo,
- sviluppo da parte di specialisti,
- ridotti tempi di rilascio,
- aumentata qualità del software
- ridotto rischio del processo,

Riuso e tipiche problematiche

Problematiche tipiche nel riuso del software:

- Incremento nei costi di gestione del sistema
- Mancanza di strumenti di supporto al riuso
- Sindrome del “NIH”
- Gestione di libreria di componenti
- Integrazione ed adattamento di componenti

Riuso - generalità

Riuso può interessare **manufatti** ma anche riuso a **livello concettuale**

Riuso di manufatti:

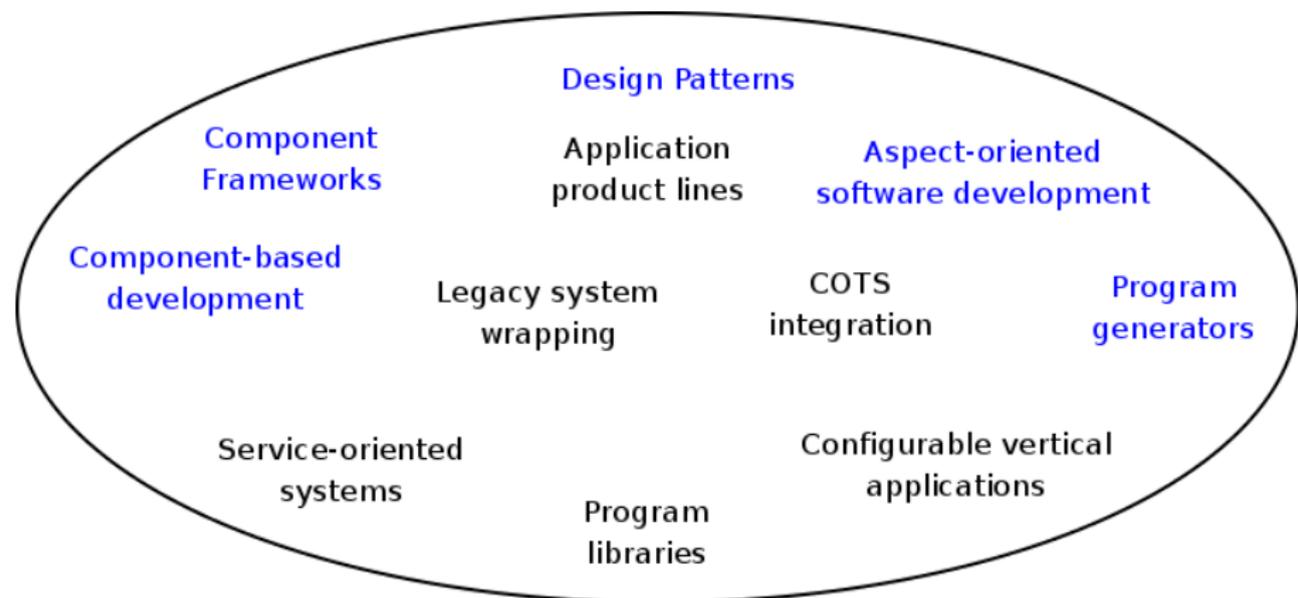
- Riuso di intere applicazioni
- Riuso di componenti e sottosistemi
- Riuso di oggetti e librerie

Sommario

1 Riuso

2 Approcci

Riuso - una panoramica



Quale approccio?

Fattori che possono guidare nella scelta di un approccio piuttosto che un'altro:

- Tempistiche dello sviluppo
- Attesa longevità dell'applicazione
- Conoscenze e capacità del team di sviluppo
- Caratteristiche di criticità del software
- Dominio applicativo
- Piattaforma di esecuzione

Design Patterns

Definizione: un pattern descrive un **problema che ricorre spesso** e propone una **possibile soluzione** in termini di organizzazione di classi/oggetti che generalmente si è rilevata efficace a risolvere il problema stesso.

Design Pattern sono caratterizzati da quattro elementi principali:

- **Nome** - riferimento mnemonico che permette di aumentare il vocabolario dei termini tecnici e ci permette di identificare il problema e la soluzione in una o due parole
- **Problema** - descrizione del problema e del contesto a cui il pattern intende fornire una soluzione
- **Soluzione** - descrive gli elementi fondamentali che costituiscono la soluzione e le relazioni che intercorrono tra questi
- **Conseguenze** - specifica le possibili conseguenze che l'applicazione della soluzione proposta può comportare. Si riferiscono ad esempio a possibili problemi di spazio o efficienza della soluzione, oppure ad applicabilità con specifici linguaggi di programmazione

Design Patterns e documentazione

Una volta ben definito e generalmente accettato da una comunità di sviluppatori il pattern diventa anche un'ottimo strumento per **documentare il software**.

Esistono esempi di applicazioni interamente descritte attraverso l'uso di design pattern (e.g. JUnit)

Testo di Riferimento:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Pattern: elements of reusable Object Oriented software
Addison Wesley

Materiale di studio a:

<https://refactoring.guru/design-patterns>

http://it.wikipedia.org/wiki/Design_pattern

Design Pattern

perché è una soluzione efficace?

Si riferisce ad un riuso di un'attività di progettazione dunque:

- **Aspetto Economico**: riduce i tempi ed i costi di progetto dei singoli moduli
- **Aspetto Tecnico**: riduce i rischi di progetto e possibilità di errori nel progetto stesso. Soluzione attuata e riutilizzata più volte e di cui si possono prevedere le caratteristiche a priori (e.g. comportamento non-funzionale)
- **Aspetto Antropologico**: semplifica la comprensione del progetto da parte di terzi fornendo livello di astrazione più chiaro

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

La definizione di collezioni di pattern in uno specifico dominio applicativo sono il primo passo per poter stabilire un contesto di riuso

Nel libro della GoF i pattern sono descritti attraverso i seguenti punti:

- **Nome**: sintetizza l'essenza del pattern. Obiettivo è ampliare il vocabolario di progetto.
- **Intento**: Descrizione che cerca di rispondere alla domanda “cosa fa il DP?” “A quale problema di progettazione si rivolge?”
- **Aka** (Also known as): eventuali altri nomi che identificano il pattern
- **Motivazioni**: Scenario che illustra il problema e come il pattern fornisce una soluzione
- **Applicabilità**: situazioni di applicabilità e come poter riconoscere tali situazioni

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

- **Struttura:** rappresentazione grafica tramite UML delle classi coinvolte e delle loro relazioni
 - **Partecipanti:** classi che partecipano al pattern, loro relazioni e responsabilità (class diagram, object ed activity diagram)
 - **Collaborazioni:** Come collaborano le varie classi per raggiungere gli obiettivi (sequence diagram)
- **Conseguenze:** vantaggi e svantaggi dell'uso del pattern e possibili effetti collaterali nell'uso del pattern
- **Implementazione:** tecniche che e suggerimenti per l'implementazione con riferimento anche a specifici linguaggi di programmazione
- **Codice sorgente di esempio:** frammenti di codice che forniscono una guida per l'implementazione
- **Usi noti:** esempi di uso in sistemi esistenti
- **Patterns correlati:** differenze e relazioni più importanti con altri pattern. Tipico uso concomitante.

Classificazione dei Pattern

Elenco di pattern può essere corposo (GoF ha definito 23 pattern generali) risulta importante cercare di identificare caratteristiche per la loro classificazione. Ciò semplifica anche lo studio e la comprensione dei pattern stessi. Sono stati identificati due criteri principali:

- A cosa si riferisce il pattern
 - **Oggetti**: relazioni fra oggetti che possono modificarsi a tempo di esecuzione
 - **Classi**: riguardano relazioni tra classi e sottoclassi
- Cosa fa il pattern (scopo)
 - **Creazionali**: meccanismi di creazione di oggetti che aumentano flessibilità e riuso di codice
 - **Strutturali**: definiscono composizioni di classi e di oggetti in strutture più grandi mantenendo flessibilità ed efficienza
 - **Comportamentali**: definisce come classi e oggetti interagiscono e distribuiscono fra loro delle responsabilità

Certamente è possibile classificarli secondo altri criteri

Object vs. Class pattern

Pattern creazionali di classe risolvono il problema della creazione di oggetti attraverso la **delega a sottoclassi**.

Per contro pattern di oggetti creazionali risolvono il problema attraverso la definizione di **specifiche interazioni tra oggetti**.

Pattern comportamentali di classe **usano ereditarietà** per definire algoritmi e flussi di controllo.

Per contro i pattern di oggetti comportamentali definiscono interazioni tra oggetti che permettono di svolgere una determinata attività

GoF classificazione

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	<i>Adapter (class)</i>	Interpreter Template Method
	Object	<i>Abstract Factory</i> Builder Prototype <i>Singleton</i>	Adapter (object) Bridge Composite Decorator Façade Flyweight <i>Proxy</i>	Chain of Responsibility Command Iterator Mediator Memento <i>Observer</i> State <i>Strategy</i> Visitor

In che modo i DP contribuiscono a semplificare la progettazione

- **Identificare gli oggetti necessari:** scomporre sistema in oggetti è compito difficile. Pattern aiutano nell'identificazione di oggetti che rappresentano astrazioni non banali a partire dalla definizione del problema che si vuole risolvere.
- **Determinare granularità degli oggetti:** l'applicazione di un pattern porta con se scelte riguardanti la granularità. Ci sono pattern che permettono di definire oggetti come composizione di oggetti più semplici oppure pattern che permettono di rappresentare sottosistemi completi con semplici oggetti
- **Definizione delle interfacce:** l'applicazione di un pattern porta con se le scelte riguardanti le interfacce ed i meccanismi di interazione tra gli oggetti
- **Definire implementazione:** applicare un pattern comporta specifiche scelte implementative. In particolare vi sono pattern che si focalizzano su **ereditarietà** mentre altri si focalizzano su meccanismi di **composizione**

Pattern Creazionali

DP creazionali

Astraggono il meccanismo di generazione di istanze di una classe con l'obiettivo di rendere il sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati. La creazione degli oggetti viene tipicamente gestita da apposite “strutture”.

Un DP creazionale che opera su classi userà ereditarietà per definire la classe da instanziare. Mentre DP che operano su oggetti useranno meccanismi di delega.

In generale semplificano la **composizione di classi** e la favoriscono rispetto all'uso dell'ereditarietà.

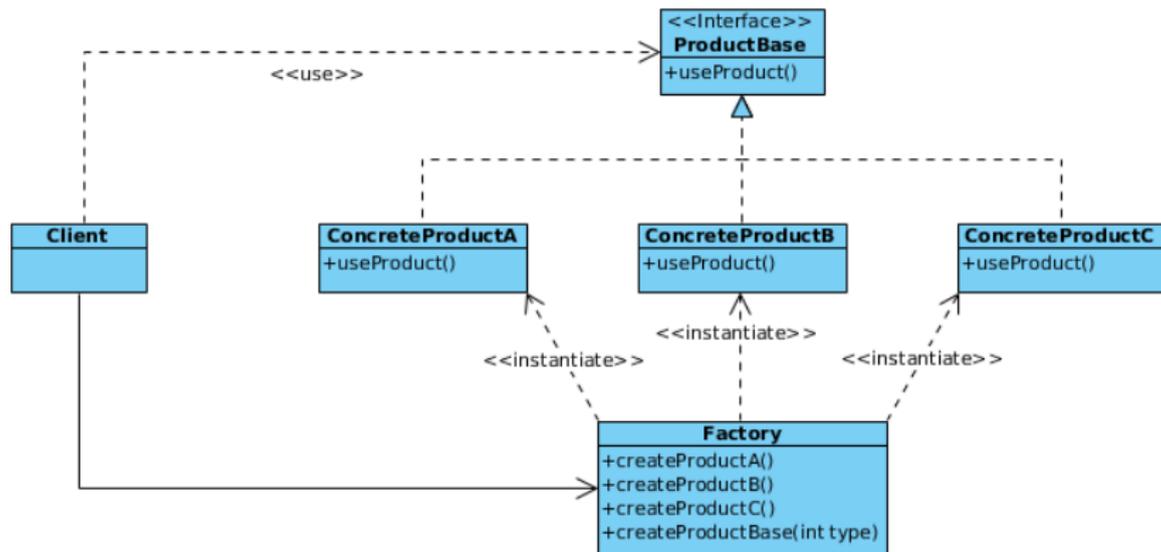
I pattern creazionali forniscono molta flessibilità nel **cosa, il come, il chi ed il quando della creazione di oggetti**

Factory Method

- **Scopo:** fornire un meccanismo per la creazione di oggetti simili (e.g. che implementano la stessa interfaccia) senza specificare quali siano le loro classi concrete
- **Motivazione:** si ha bisogno di determinare l'esatto tipo dell'oggetto da instanziare solo a run-time
- **Sinonimi:** Virtual Constructor
- **Applicabilità:**
 - sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti
 - si vuole una libreria (i.e. insieme di classi) che esponga soltanto l'interfaccia e non la sua implementazione

<https://refactoring.guru/design-patterns/factory-method>

Factory Method - Struttura



Factory Method

● Conseguenze:

- isola classi concrete
- consente di cambiare in modo semplice l'implementazione ed in comportamenti esposti da un insieme di prodotti
- promuove il riuso di prodotti/artefatti/codice
- aggiunta e supporto di nuovi di prodotti semplice

● Partecipanti:

- **Factory** (`Factory`) dichiara ed implementa i meccanismi di creazione per un insieme oggetti simili (e.g. che condividono la stessa interfaccia)
- **AbstractProduct** (`ProductBase`) dichiara un'interfaccia (i.e. classe astratta o interface) per una tipologia di oggetti prodotto
- **ConcreteProduct** (`ConcreteProductA`, `ConcreteProductB`, `ConcreteProductC`) definisce un oggetto prodotto che dovrà essere creato dalla corrispondente factory implementa l'interfaccia definita da `AbstractProduct`
- **Client** (`Client`) crea istanze di `ConcreteProduct` attraverso la `Factory` utilizza soltanto l'interfaccia dichiarate in `AbstractProduct`

Abstract Factory

- **Scopo**: definire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le classi concrete.
- **Motivazione**: spesso ci si trova di fronte al problema di voler istanziare un oggetto senza specificare precisamente la classe. Comportamento chiaramente non possibile con meccanismi di creazione tipo `new`. Caso tipico delle interfacce grafiche e delle interfacce che mantengono coerenza con il tema in uso.
- **Applicabilità**: utilizzato quando:
 - sistema indipendente dalle modalità di creazione, composizione;
 - sistema configurabile dipendentemente dalle caratteristiche di una tra piú tipologie di oggetto
 - libreria di classi fornendo solo interfaccia ma non implementazioni specifiche

<https://refactoring.guru/design-patterns/abstract-factory>

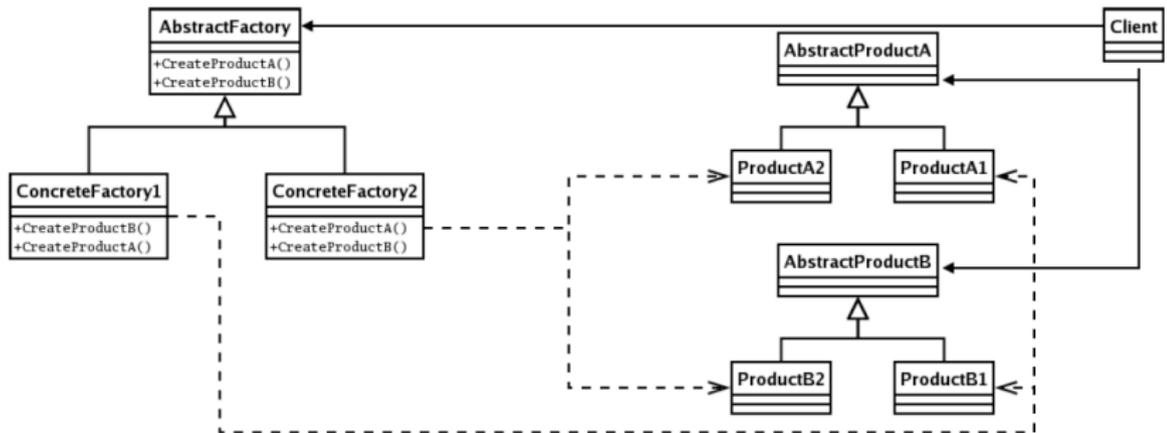
Abstract Factory

- **Conseguenze:**

- isolamento delle classi concrete. Gli oggetti non contengono dettagli che si riferiscono ai meccanismi di creazione degli oggetti. La creazione di un particolare oggetto è visto come un normale servizio fornito da classi preposte allo scopo.
- Risulta semplice la modifica/aggiunta di classi rappresentanti factory concrete. L'uso di queste risulta estremamente localizzato. Dunque prodotti diversi si ottengono modificando la factory.
- Il supporto a nuove tipologie di prodotto che richiederebbero di modificare l'abstract factory risulta complessa in quanto richiede di modificare tutte le classi concrete.

Abstract Factory

- Struttura:



Abstract Factory

- **Partecipanti:**

- **AbstractFactory:** Dichiarata un'interfaccia per le operazioni di creazione di oggetti astratti
- **ConcreteFactory:** implementa le operazioni di creazione di oggetti concreti
- **AbstractProduct:** Dichiarata un'interfaccia per una tipologia di oggetti
- **ConcreteProduct:** definisce un oggetto che dovrà essere creato dalla corrispondente factory concreta
- **Client:** utilizza soltanto le interfacce dichiarate sopra

- **Collaborazioni:**

- spesso esiste una sola istanza di una ConcreteFactory a run-time per le diverse tipologie
- La classe AbstractFactory delega la creazione di oggetti alle sue sottoclassi.

Abstract Factory

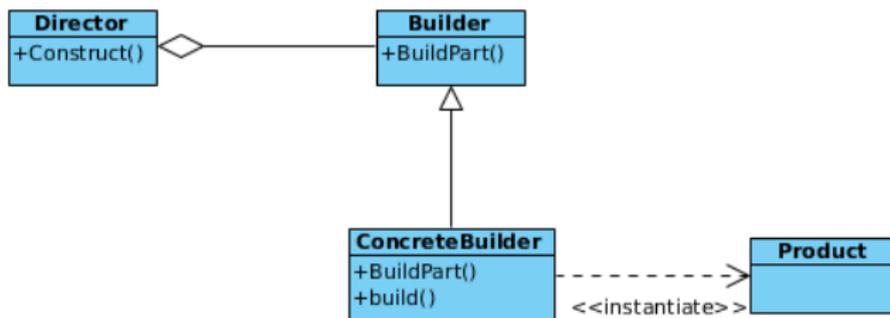
- Implementazione: ...
- Utilizzi noti: ...
- **Pattern correlati**: associato ad altri pattern creazionali in particolare al Singleton per avere classi concrete che non ammettono istanze multiple.

Builder

- **Scopo:** Separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa generare differenti rappresentazioni.
- **Motivazione:** Si immagini un generatore di profili per un gioco di ruolo. Il modo più semplice di popolare il sistema è permettere al computer di creare i profili. Ma se si volessero selezionare alcune caratteristiche quali professione, genere, colore dei capelli, etc. la generazione diventa un processo passo-passo che termina quando tutte le caratteristiche sono state selezionate. Il Builder pattern permette di creare differenti caratteri di un oggetto evitando il problema del “constructor pollution”.

<https://refactoring.guru/design-patterns/builder>

Builder - Struttura



Builder

- **Applicabilità:**

- da usare quando algoritmo di creazione di oggetto complesso deve essere reso indipendente dalle parti
- processo di creazione diverso dalla rappresentazione

- **Partecipanti:**

- **Director** (`Director`): costruisce l'oggetto usando algoritmi complessi e l'interfaccia di builder.
- **Builder** (`Builder`): specifica un'interfaccia (classe astratta) per la costruzione di parti di oggetti complessi
- **ConcreteBuilder** (`ThiefBuilder`, `WarriorBuilder`, `MageBuilder`): fornisce un'interfaccia per la costruzione del prodotto semplificandone gli aspetti. Fornisce un'interfaccia per recuperare oggetto creato.
- **Product** (`Hero`): prodotto complesso che deve essere creato

Builder

- **Conseguenze:**

- Isola il codice di costruzione da quello di rappresentazione
- controllo più fine sul processo di costruzione
- più facile cambiare la rappresentazione interna degli oggetti

- **Utilizzi noti:**

- `java.lang.StringBuilder`
- `java.nio.ByteBuffer` as well as similar buffers such as `FloatBuffer`, `IntBuffer` and so on.
- `java.lang.StringBuffer`
- All implementations of `java.lang.Appendable`
- Apache Camel builders
- Apache Commons `Option.Builder`

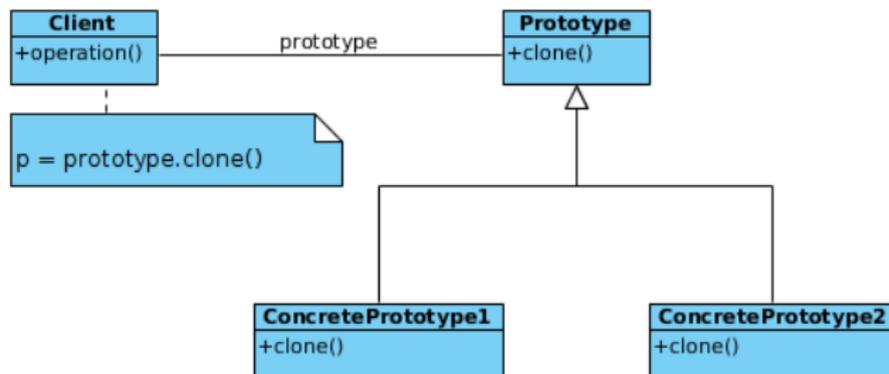
- **Pattern correlati:**

Prototype

- **Scopo:** Permette di specificare oggetti a partire da un prototipo che può essere poi dettagliato
- **Motivazione:** Si vogliono realizzare artefatti complessi a partire da framework generali per quel contesto. Ad esempio costruire editor di modellazione per specifiche notazioni riusando meta-modelli definiti per la notazione.
- **Sinonimi:**
- **Applicabilità:** quando il sistema deve essere indipendente da come è prodotto, creato, e rappresentato:
 - quando le classi da istanziare sono create a run-time (dynamic loading)
 - evitare di costruire gerarchie di classi “factory” che rappresentano parallelamente la gerarchia delle classi
 - quando le istanze della classe possono avere poche differenti combinazioni di stato, è più conveniente usare prototipi piuttosto che creare classi manualmente
 - quando processo di creazione è costoso rispetto al cloning.

<https://refactoring.guru/design-patterns/prototype>

Prototype - Struttura



Prototype

- **Conseguenze:** Come per altri pattern nasconde il prodotto concreto dal client riducendo il numero di “nomi” conosciuti allo stesso.
 - Aggiunta rimozione di prodotti a run-time più facile
 - specifica di nuovi oggetti variando soltanto alcuni valori
 - specifica di nuovi oggetti variando la struttura
 - ridotto subclassing
 - configurare applicazione configurando dinamicamente le classi
- **Partecipanti:**
 - **Prototype:** dichiara interfaccia per cloning
 - **ConcretePrototype:** implementa operazione di cloning
 - **Client:** crea un nuovo oggetto richiedendo cloning

Prototype

- Implementazione:
 - uso di un prototype manager
 - operazione di “clone”
 - inizializzazione dei cloni

Java include interfaccia generale per cloning “java.lang.Cloneable”.
Attenzione a gestione di riferimenti condivisi in oggetti complessi.

- Utilizzi noti: . . .
- **Pattern correlati:** una “Abstract Factory” potrebbe usare prototipi per per creare oggetti. Il prototipo potrebbe essere utile per pattern quali “Composite” e “Decorator”

Singleton

- **Scopo:** Fare in modo che a run-time esista al più un'istanza di una classe e fornire un punto globale di accesso a tale istanza
- **Motivazione:** È spesso importante avere una sola istanza di una classe a run-time. Si consideri ad esempio il caso del File System Manager o del Window Manager di un sistema. (Kdm o Gdm per chi usa Linux). Dunque come possiamo fare in modo da garantire che a run-time non sia possibile avere più istanze di una stessa classe e che tale istanze sia facilmente accessibile ai potenziali utilizzatori?
- **Applicabilità:** il pattern singleton può essere usato quando:
 - deve esistere una sola istanza di una classe e punto di accesso noto agli utilizzatori
 - quando l'unica istanza deve poter essere estesa attraverso definizione di sottoclassi ed i client non devono essere modificati come conseguenza di ciò

<https://refactoring.guru/design-patterns/singleton>

Singleton

- **Conseguenze:** controllo degli accessi all'istanza, spazio dei nomi ridotto, possibilità di estensione ad aver un numero n di istanze, facile manutenzione
- **Struttura:**

Singleton
-uniqueInstance
-singletonData
+getInstance()
+SingletonOperation()
+getSingletonData()

- **Partecipanti:**
 - **Singleton:** definisce un'operazione `getInstance` che permette ai client di accedere all'unica istanza disponibile della classe. La detta operazione deve essere un'operazione di classe ovvero in Java dovrà essere marcata dalla parola chiave `static`

Singleton

- **Collaborazioni:** I client possono accedere ad un'istanza di singleton soltanto invocando l'operazione `getInstance`
- **Conseguenze:** Il pattern Singleton offre importanti benefici:
 - Accesso controllato ad un'unica istanza
 - Riduzione dello spazio dei nomi
 - Raffinamento delle operazioni e della rappresentazione interna
 - Gestione di un numero variabile di istanze
- **Implementazione:** si possono fare alcune considerazioni sui differenti meccanismi messi a disposizione dai differenti linguaggi OO per la definizione di operazioni di classe o per altri meccanismi che possono influenzare l'implementazione con un dato linguaggio.
- **Utilizzi noti:** ...
- **Pattern correlati:** altri pattern creazionali potrebbero usare il pattern singleton per esempio per risolvere gli specifici problemi "creazionali" associandoli alla necessità di avere una sola istanza.

Combinazioni e concorrenza

- Come si comportano le classi singleton in presenza di clienti concorrenti?
- Come combinare Abstract Factory e Singleton?

Pattern Strutturali

Pattern Strutturali

I pattern strutturali descrivono come comporre oggetti o classi per creare e riutilizzare strutture complesse fornendo interfacce più adatte all'uso da parte di oggetti client.

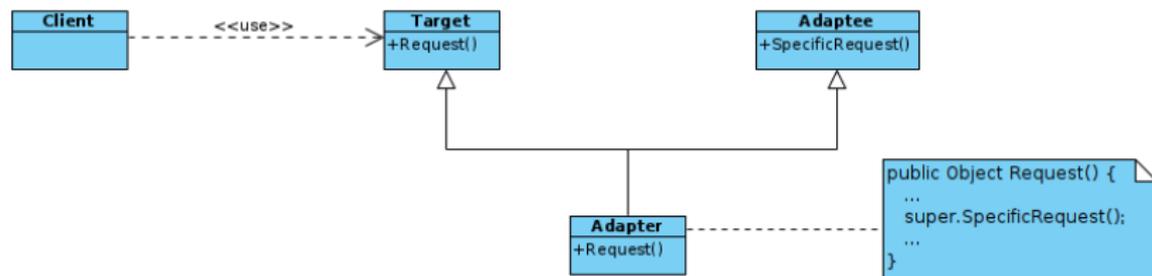
- **basati su classi:** utilizzano l'ereditarietà per comporre interfacce o implementazioni esempio: ereditarietà multipla combina due o più classi per ottenere una classe con tutte le proprietà delle superclassi
- **basati su oggetti:** modellano le modalità di composizione di oggetti per realizzare nuove funzionalità hanno maggiore flessibilità poiché è possibile cambiare la composizione durante l'esecuzione

Adapter

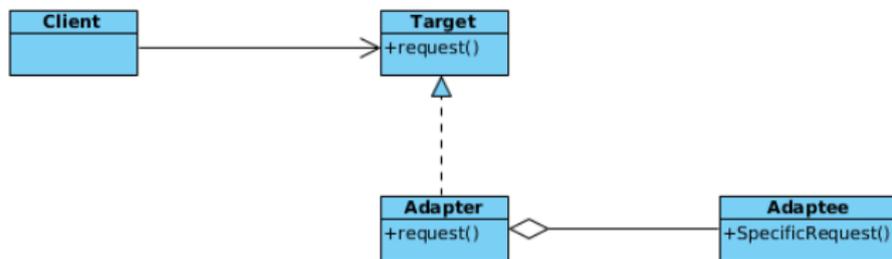
- **Scopo:**
 - gestire interfacce incompatibili
 - fornire un'interfaccia stabile a classi funzionalmente simili o con interfacce diverse
- **Motivazione:** spesso le classi di un sistema vengono strutturate/progettate cercando di offrire un profilo riusabile. Tuttavia, può capitare che queste classi non possano essere effettivamente riusate; per esempio perchè la loro interfaccia offerta non rispecchia esattamente i requisiti (o la segnatura) richiesti in uno specifico dominio
- **Sinonimi:** Wrapper
- **Applicabilità:**
 - si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve
 - si vuole realizzare una classe riusabile che coopera con altre classi anche se scorrelate o impreviste e con una interfaccia eventualmente incompatibile

<https://refactoring.guru/design-patterns/adapter>

Adapter - Struttura



Class



Object

Adapter

- **Conseguenze:**

- adatta Adaptee a Target attraverso la classe concreta Adapter; la classe Adapter non è indicata se si volesse gestire l'adattamento di una classe ma anche tutte le sue sotto-classi
- quanto "lavoro" deve fare la classe Adapter? dipende da quanto sono simili le interfacce Target ed Adaptee.
- l'uso del pattern non è sempre trasparente ai Client

- **Partecipanti:**

- **Target:** definisce l'interfaccia di dominio con il client
- **Client:** coopera con oggetti conformi all'interfaccia Target
- **Adaptee:** definisce l'interfaccia esistente da adattare
- **Adapter:** realizza il meccanismo di adattamento

Adapter

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati:

Bridge

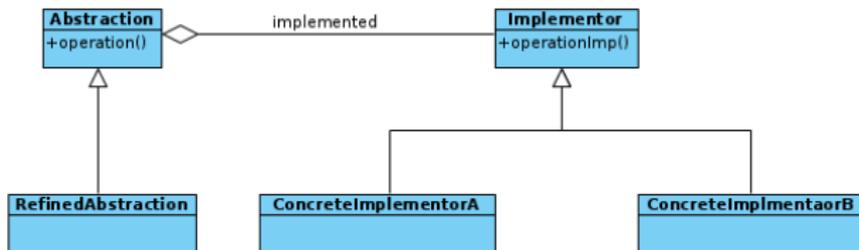
- **Scopo:** Dividere classi o famiglie di classi correlate in due gerarchie, astrazione e implementazione, che possono così essere sviluppate indipendentemente
- **Motivazione:** Situazioni in cui l'implementazione di un possibile concetto astratto può essere molteplice e variare su ulteriori dimensioni dipendenti dallo specifico contesto implementativo. Uso di ereditarietà crea alberi molto complessi e non distingue aspetti concettuali da aspetti più "implementativi".
- **Sinonimi:** Handle, Body

<https://refactoring.guru/design-patterns/bridge>

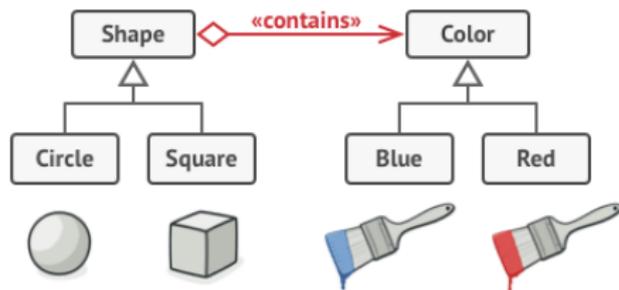
Bridge

- **Applicabilità:**

- evitare binding permanente tra astrazione ed implementazione, avendo possibilità di modificare a run-time
- permettere indipendente estensibilità di astrazione e implementazione
- proliferazione di classi in gerarchie di ereditarietà



Bridge idea



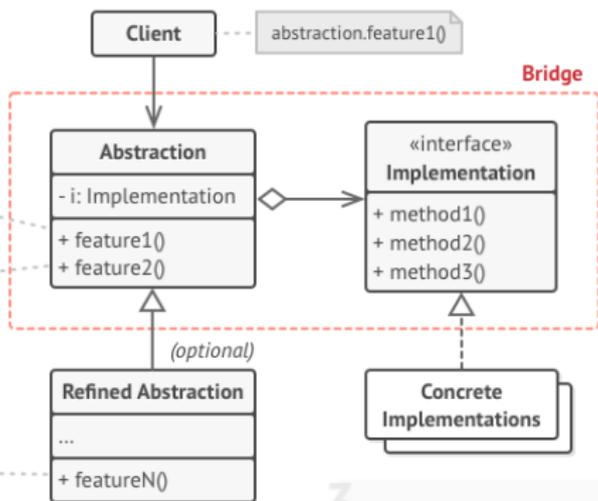
Bridge - Struttura e codice di esempio

1 The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

i.method1()
i.method2()
i.method3()

i.methodN()
i.methodM()

5 Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.



4 **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

3 **Concrete Implementations** contain platform-specific code.

2 The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

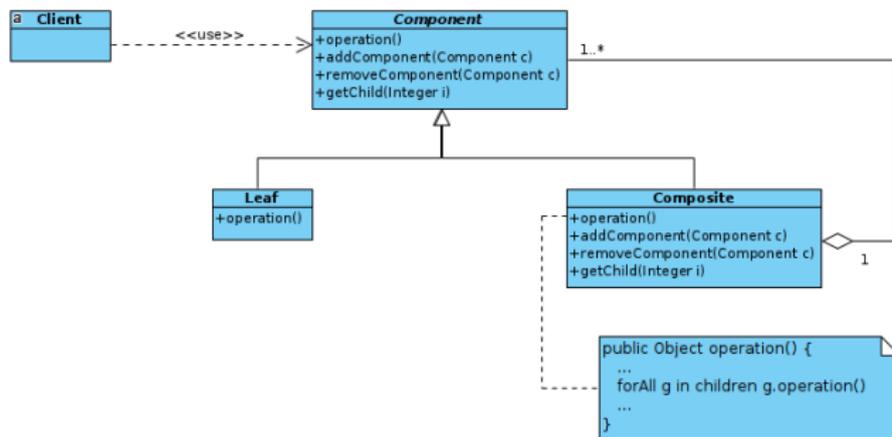
- **Conseguenze:**
 - Disaccoppiamento di aspetti differenti di un oggetto
 - Maggiore flessibilità ed Estendibilità
 - maggiore “hiding” di dettagli implementativi
- **Implementazione:** ...
- **Utilizzi noti:** ...
- **Pattern correlati:** Adapter, Strategy, State, Abstract Factory

Composite

- **Scopo:** comporre oggetti in strutture dati ad albero per rappresentare relazioni del tipo intero-parte. Allo stesso tempo si vorrebbe rendere la struttura trasparente così da interagire con composizioni e singoli oggetti in modo trasparente.
- **Motivazione:** Si vuole evitare di esporre al client informazioni sulla composizione e permettergli di lavorare in modo flessibile indipendentemente dalle differenze del componente/composto
- **Sinonimi:**
- **Applicabilità:** si vuole rappresentare una gerarchia intero-parte e rendere i clienti non dipendenti dalla gerarchia

<https://refactoring.guru/design-patterns/composite>

Composite - Struttura



Composite

- **Conseguenze:**

- indipendenza dei client dai dettagli di strutture complesse
- facilità di gestione delle composizioni
- possibilità di gestire vincoli sui compositi nella struttura

- **Partecipanti:**

- **Component**: rappresenta l'interfaccia che gli oggetti nella composizione e fornisce il comportamento di default per compositi e componenti
- **Leaf**: rappresenta componenti base che dunque hanno un comportamento "primitivo"
- **Composite**: serve a creare la struttura ad albero e immagazzina dunque i riferimenti ai suoi componenti
- **Client**: accede ad oggetti di tipo componente senza aver contezza della struttura degli stessi

Composite

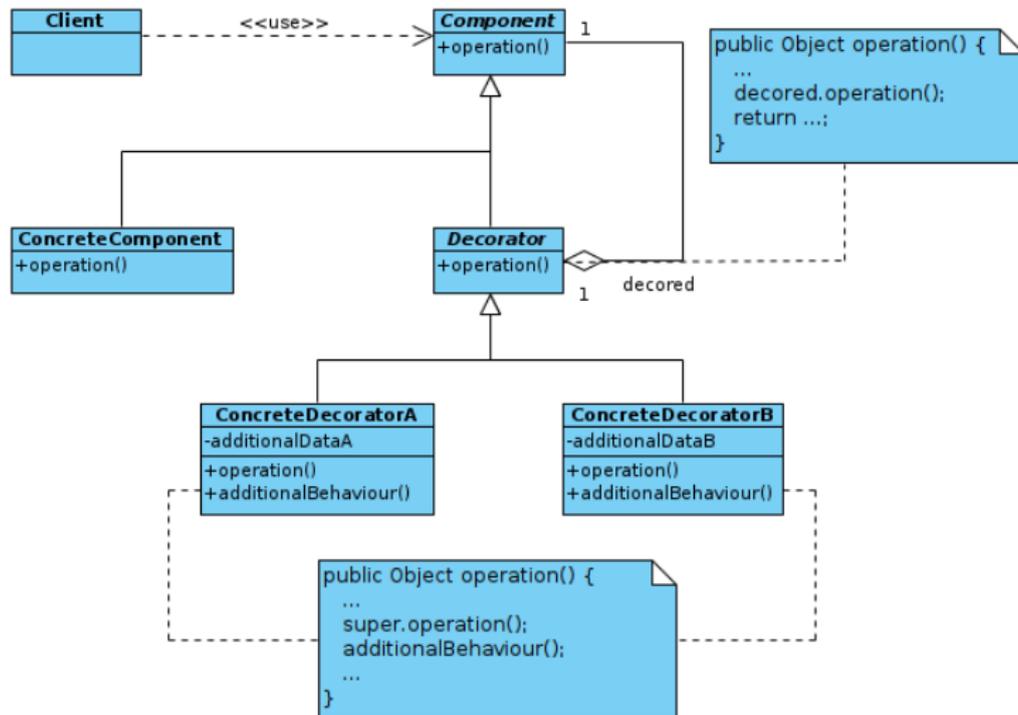
- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati:

Decorator

- **Scopo:** aggiungere dinamicamente responsabilità ad un oggetto. I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità
- **Motivazione:**
 - GUI toolkit dovrebbe consentire di aggiungere proprietà come bordi, scorrimento, ai singoli elementi grafici il modo per aggiungere responsabilità è tramite ereditarietà
 - e.g. estendere una finestra e aggiungere il bordo
 - definire un approccio flessibile che consenta di racchiudere il componente da "decorare" in un altro che abbia la sola responsabilità di aggiungere il bordo/scrollbar/etc oggetto contenitore detto decorator decorator ha un'interfaccia conforme all'oggetto decorato in modo tale da essere trasparente ai vari client decorator trasferisce le richieste al componente decorato effettuando azioni aggiuntive (esempio decorando il bordo) prima o dopo il trasferimento della richiesta essendo trasparente ai client è possibile annidare i decorator consentendo l'aggiunta di un numero illimitato di responsabilità agli oggetti decorati
- **Sinonimi:** Wrapper

<https://refactoring.guru/design-patterns/decorator>

Decorator - Struttura



Decorator

● Applicabilità:

- si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente ed in modo trasparente, senza coinvolgere altri oggetti
- si vuole poter togliere responsabilità agli oggetti
- l'estensione diretta attraverso le definizioni di sottoclassi non è praticabile
 - esplosione di sottoclassi per supportare ogni possibile combinazione

● Conseguenze:

- maggiore flessibilità rispetto all'utilizzo dell'ereditarietà (multipla) statica
 - responsabilità possono essere aggiunte e rimosse in esecuzione semplicemente collegando e scollegando i decoratori agli oggetti decorati
 - in caso di ereditarietà è necessario creare una nuova classe per ogni responsabilità (es. `BorderedScrollableTextView`, `BorderedTextView`)
- consente di evitare di definire classi troppo complesse nella gerarchia

Decorator

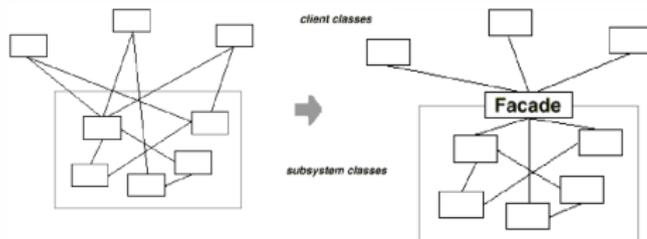
- **Partecipanti:**
 - **Component** (`VisualComponent`): definisce l'interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente
 - **ConcreteComponent** (`TextView`): definisce un oggetto al quale possono essere aggiunte responsabilità ulteriori
 - **Decorator**: mantiene un riferimento ad un oggetto Component e definisce un'interfaccia conforme all'interfaccia di Component
 - **ConcreteDecorator** (`BorderDecorator`, `ScrollDecorator`): aggiunge responsabilità al componente
- **Implementazione:** ...
- **Utilizzi noti:** ...
- **Pattern correlati:**

Façade

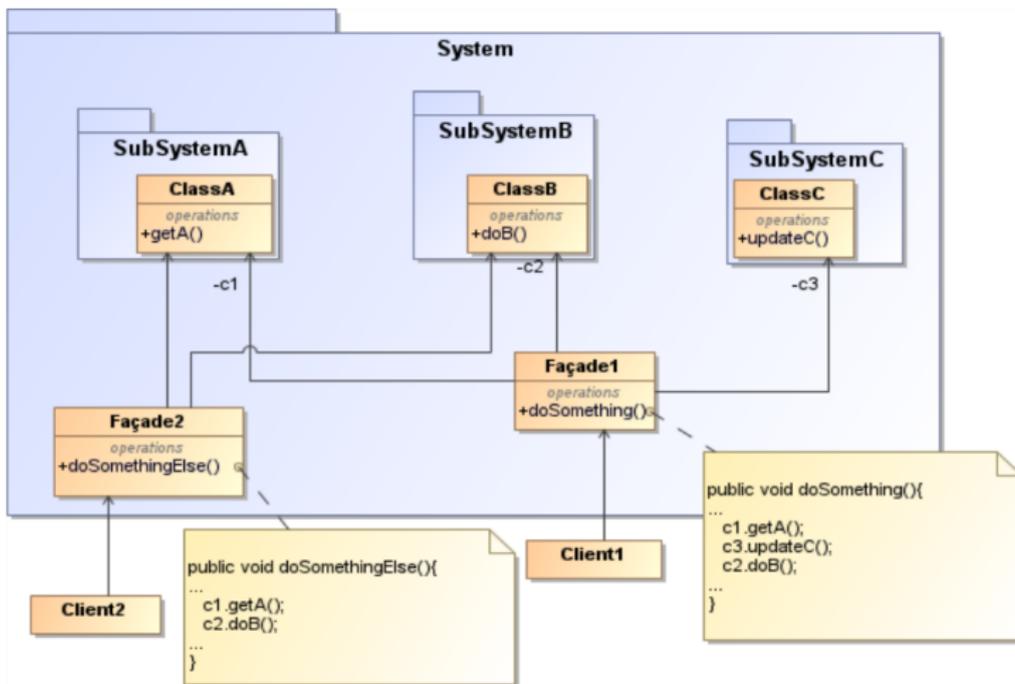
- **Scopo:** fornire una interfaccia unificata ad un insieme di interfacce di un sottosistema
- **Motivazione:**
 - è richiesta un'interfaccia comune ed unificata per un insieme disparato di implementazioni (i.e. classi o interfacce), come se si stesse identificando un sottosistema
 - minimizzare le comunicazioni e le dipendenze tra sottosistemi
 - mascherare l'implementazione di un sottosistema
 - l'implementazione può cambiare nel tempo ma non le funzionalità offerte

<https://refactoring.guru/design-patterns/facade>

idea



Façade - Struttura



Façade

● Applicabilità:

- progettazione architeturale di un sistema
- fornire una interfaccia ad un sistema complesso o che evolve nel tempo
- aumentare il grado di riuso di un sottosistema
- stratificare un sistema identificando i punti di accesso ad ogni sottosistema
- ci sono molte dipendenze tra l'implementazione del sottosistema ed i suoi client
- un client per realizzare una singola operazione logica deve accedere a più classi del sottosistema molto differenti tra loro

● Partecipanti:

- **System**: definisce un sistema del quale si vuole nascondere i dettagli implementativi all'esterno
- **SubSystem** (`SystemA`, `SystemB`, `SystemC`): definisce eventuali sottomoduli del sistema
- **Façade** (`Façade1`, `Façade2`): definisce l'interfaccia comune per l'accesso alle funzionalità del sistema può implementare logiche composizionali per esportare funzionalità di sistema
- **Client** (`Client1`, `Client2`): utilizzatore delle funzionalità del sistema

- **Conseguenze:**

- riduce il numero di oggetti che un client deve gestire
- rende il sistema più riusabile
- riduce il grado di accoppiamento tra un sistema ed i suoi client, mitigando la ripercussione delle modifiche sul sistema anche sui client
- la strutturazione in livelli consente di progettare sistemi indipendenti, con un basso tasso di dipendenze circolari

- **Implementazione:** ...

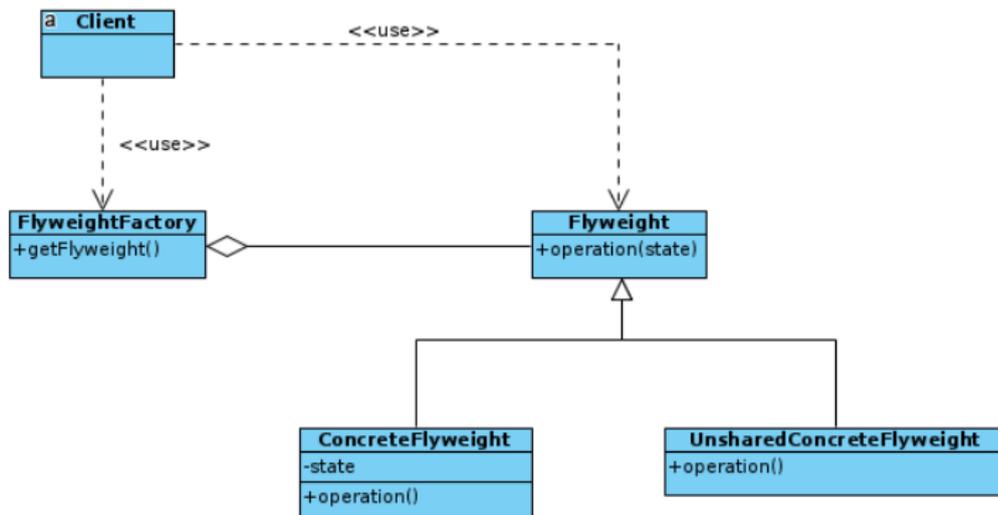
- **Utilizzi noti:** ...

- **Pattern correlati:**

Flyweight

- **Scopo:** ridurre utilizzo di risorse attraverso la condivisione nel contesto di sistemi che prevedono la presenza a run-time di un numero elevato di oggetti
- **Motivazione:** ci possono essere situazioni in cui esistono a run-time un numero molto alto di oggetti che utilizzano risorse limitate ma la loro numerosità crea un'impronta molto alta. Il pattern si applica cercando di creare una condivisione di risorse dunque astruendo la risorsa per collocarla in un oggetto che possa essere condiviso.
- **Sinonimi:**
- **Applicabilità:** Si applichi il pattern quando tutte le condizioni seguenti sono soddisfatte:
 - l'applicazione usa un numero elevato di oggetti
 - l'utilizzo della risorsa è elevato a causa del numero di oggetti
 - buona parte dello stato dell'oggetto può essere "esteriorizzato" (**intrinsic state**)
 - molti gruppi di oggetti possono essere rimpiazzati da pochi oggetti una volta che lo stato è stato esteriorizzato
 - l'applicazione non dipende dall'identità dell'oggetto

Flyweight - Struttura e codice di esempio



Flyweight

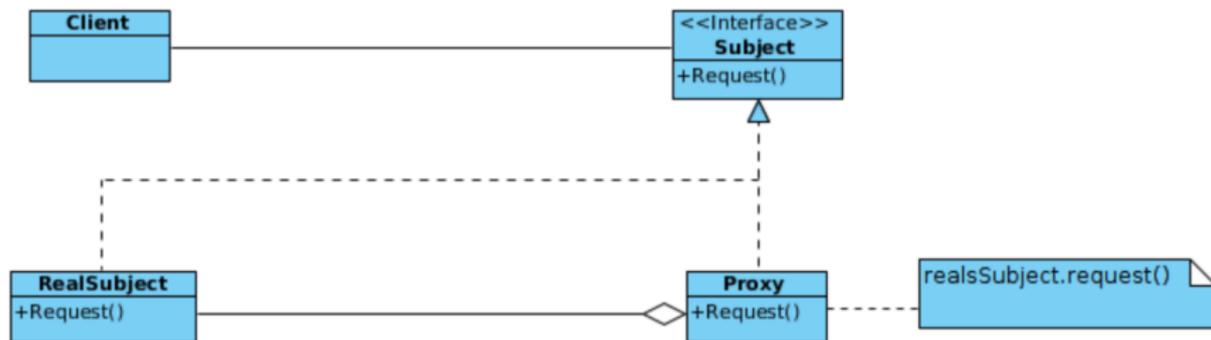
- **Conseguenze:** si potrebbe incorrere in costi di performance dovuti ad esteriorizzazione dello stato
- **Partecipanti:**
 - ***Flyweight***: dichiara un'interfaccia attraverso la quale flyweights possono ricevere messaggi e agire sullo stato esternalizzato
 - ***ConcreteFlyweight***: implementa l'interfaccia flyweight e gestisce lo storage per lo stato condivisibile
 - ***UnsharedConcreteFlyweight***:
 - ***FlyweightFactory***: crea e gestisce gli oggetti flyweight e quando un client richiede un oggetto ne fornisce una copia se l'oggetto non esiste
 - ***Client***: ha riferimenti ai flyweights, può compueare e immagazzinare lo stato esternalizzato dei flyweight
- **Implementazione:**
- **Utilizzi noti:**
- **Pattern correlati:**

Proxy

- **Scopo:** fornire un surrogato o un segnaposto per un altro oggetto al fine di controllare l'accesso all'oggetto stesso. Allo stesso tempo può essere utile per **rimandare la creazione** di un oggetto ad un momento successivo al fine di ridurre uso di risorse.
- **Motivazione:** motivo può essere differire il costo di creazione ed inizializzazione dell'oggetto ad un momento successivo quando questo è effettivamente necessario
- **Applicabilità:** il pattern proxy può essere usato secondo diverse tipologie:
 - **remote proxy:** fornisce rappresentazione locale per un oggetto residente in un diverso spazio di indirizzamento
 - **virtual proxy:** gestisce su richiesta la creazione di oggetti costosi
 - **protection proxy:** controlla l'accesso ad un oggetto. Si rivela utile quando possono esserci diversi diritti di accesso allo stesso oggetto.
 - **smart reference:** sostituisce quello che è un puntatore puro con un puntatore dalle funzionalità aggiuntive quando si accede all'oggetto referenziato

<https://refactoring.guru/design-patterns/proxy>

Proxy - Struttura



- **Conseguenze:**

- introduce un livello di indirectione nell'accesso ad un oggetto. In generale operazioni molto complesse possono portare a degrado.
- disaccoppiano cliente e oggetto stesso ottimizzando l'uso delle risorse rimandando solo al momento necessario l'esecuzione di operazioni potenzialmente costose.

- **Partecipanti:**

- **Proxy**: mantiene un riferimento che consente al proxy di accedere all'oggetto rappresentato. Proxy deve implementare la stessa interfaccia del referenziato al fine di nascondere la propria presenza al cliente
- **Subject**: definisce l'interfaccia comune per l'oggetto referenziato ed il proxy. Il proxy può sempre essere usato dove è richiesto l'oggetto referenziato.
- **RealSubject**: caratterizza l'oggetto referenziato dal proxy

Proxy

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati:

Pattern Comportamentali

Pattern Comportamentali

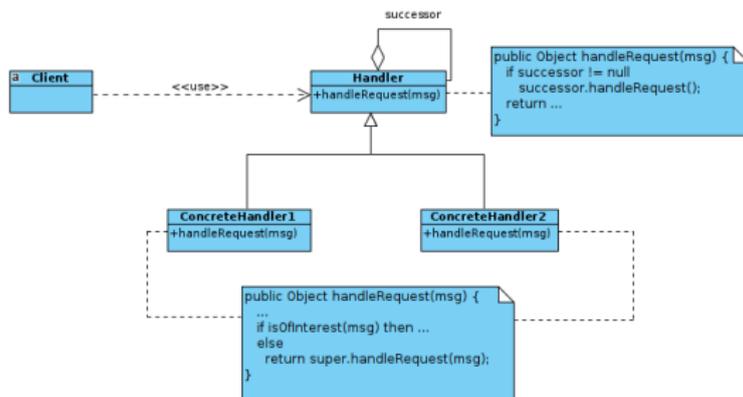
I pattern comportamentali hanno a che fare con gli algoritmi e gli assegnamenti di responsabilità tra gli oggetti. Caratterizzano sistemi di controllo complessi e si focalizzano anche sui pattern di interazione tra i componenti.

Chain of Responsibility

- **Scopo:** Evitare di accoppiare il mittente di una richiesta ed il suo ricevente. Il messaggio viene inoltrato ad una catena di riceventi che possono decidere di gestirlo o inoltrarlo al ricevente successivo.
- **Motivazione:** si vuole evitare di far conoscere al cliente chi deve gestire una certa richiesta. Questo permette di disaccoppiare il cliente e crea più isolamento delle classi. È possibile aggiungere la gestione di certi eventi in maniera semplice aggiungendo un gestore.
- **Applicabilità:**
 - più di un oggetto vorrebbe poter gestire una richiesta
 - non si vuole conoscere il ricevente
 - gli oggetti che possono gestire le richieste possono modificarsi dinamicamente

<https://refactoring.guru/design-patterns/chain-of-responsibility>

Chain of Responsibility - Struttura



● Partecipanti:

- **Handler**: definisce un'interfaccia per la gestione delle richieste. Include il link al successore
- **ConcreteHandler**: decide se gestire la richiesta o se inoltrarla al successivo gestore.
- **Client**: richiede la gestione di un messaggio alla catena

Chain of Responsibility

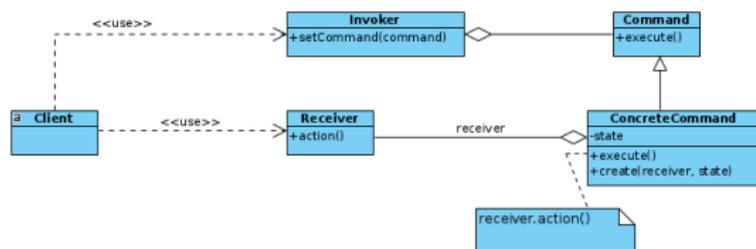
- **Conseguenze:**
 - accoppiamento ridotto
 - flessibilità nell'assegnamento delle responsabilità e nella loro modifica
 - la gestione non è garantita
- **Implementazione:** ...
- **Utilizzi noti:** ...
- **Pattern correlati:**

Command

- **Scopo:** si vuole incapsulare una richiesta in un oggetto così da parametrizzare la gestione della stessa
- **Motivazione:** in alcuni casi si vuole emettere una richiesta senza avere nessuna conoscenza dell'operazione necessaria e di chi possa gestirla.
- **Applicabilità:**
 - parametrizzare un oggetto sulla base di un azione da eseguire.
 - specificare, accodare e gestire le richiesta con tempi definiti dal gestore
 - supportare azioni di "undo"
 - supportare il logging delle azioni effettuate

<https://refactoring.guru/design-patterns/command>

Command - Struttura



● Partecipanti:

- **Command**: dichiara un'interfaccia per l'esecuzione di un'operazione
- **ConcreteCommand**: definisce un collegamento tra il ricevente ed un'azione. Implementa l'operazione di esecuzione invocando la corrispondente operazione nel Receiver
- **Client**: crea un oggetto **ConcreteCommand** ed imposta il Receiver
- **Invoker**: richiede al **Command** di eseguire il task
- **Receiver**: sa come eseguire le operazioni legate ad una richiesta

Command

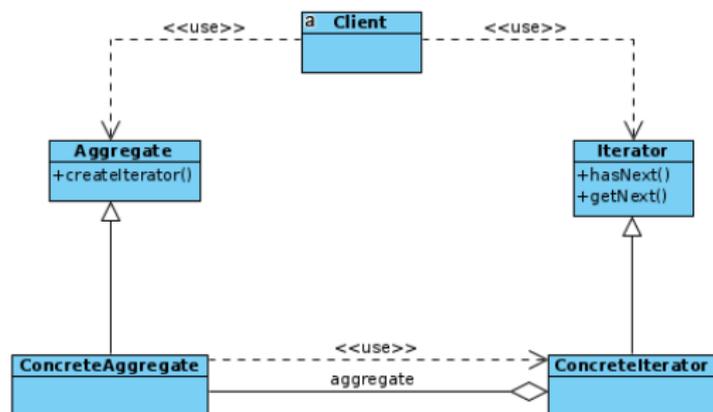
- **Conseguenze:**
 - disaccoppiamento tra oggetto che effettua la richiesta e chi sa come gestirla
 - facilità di estendere i comandi in quanto rappresentati come classi
 - comandi possono diventare compositi
 - facile aggiungere nuovi comandi
- **Implementazione:** ...
- **Utilizzi noti:** ...
- **Pattern correlati:**

Iterator

- **Scopo:** poter accedere ad un aggregato di oggetti sequenzialmente senza dover esporre la struttura di rappresentazione sottostante
- **Motivazione:** poter accedere ad aggregati di oggetti senza considerare la loro struttura sottostante permette di disaccoppiare questo comportamento dal cliente e dunque riduce dipendenza tra le classi e modifiche alla rappresentazione interna diventano non percepibili all'esterno.
- **Applicabilità:**
 - per accedere un aggregato senza esporre rappresentazione interna
 - per avere più strategie di attraversamento trasparenti
 - per fornire un'interfaccia uniforme per la scansione di un aggregato e dunque avere comportamenti polimorfi

`https://refactoring.guru/design-patterns/iterator`

Iterator - Struttura



● Partecipanti:

- **Iterator**: dichiara interfaccia per poter traversare gli elementi dell'aggregato
- **ConcreteIterator**: fornisce gli algoritmi per navigare all'interno dell'aggregato
- **Aggregate**: definisce l'interfaccia in modo da poter generare iteratori compatibili e dunque poterli settare sull'aggregato
- **ConcreteAggregate**: contiene gli oggetti organizzati in una struttura dati e implementano i metodi dell'interfaccia Aggregate
- **Client**: interagisce con la classe aggregante e l'iterator per navigare l'aggregato

Iterator

- **Conseguenze:**

- supporto di varianti nella scansione di aggregati e disaccoppiamento dai dettagli implementativi
- possibilità di avere più scansioni differenti sullo stesso aggregato
- semplificazione e “standardizzazione” della scansione di aggregati
- attenzione a inserimento e rimozione di elementi ad aggregato mentre un iterator sta scandendo la struttura

- **Implementazione:** ...

- **Utilizzi noti:** ...

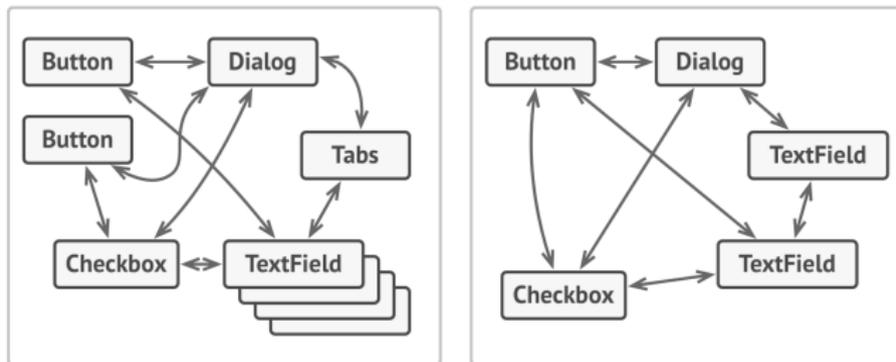
- **Pattern correlati:**

Mediator

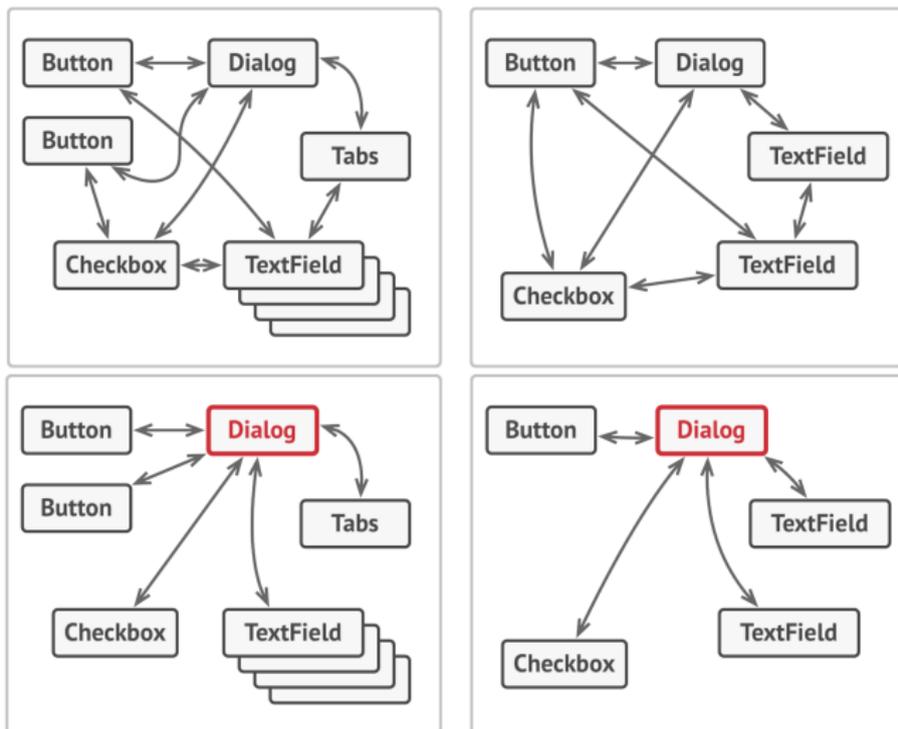
- **Scopo:** ridurre dipendenze caotiche tra oggetti. Il pattern restringe le comunicazioni dirette tra oggetti obbligandoli a parlare tra gli oggetti Mediator.
- **Motivazione:** La distribuzione delle responsabilità può produrre molte interconnessioni tra gli oggetti e nel caso limite ogni oggetto conosce tutti gli altri. A quel punto diventa però anche difficile cambiare il comportamento che è distribuito tra molti oggetti. L'idea è quella dunque di usare un oggetto preposto denominato **mediator**. Tale oggetto è responsabile di controllare e coordinare le interazioni di un gruppo di oggetti. Gli oggetti dunque conoscono soltanto il mediator e dunque si riducono le dipendenze.

<https://refactoring.guru/design-patterns/mediator>

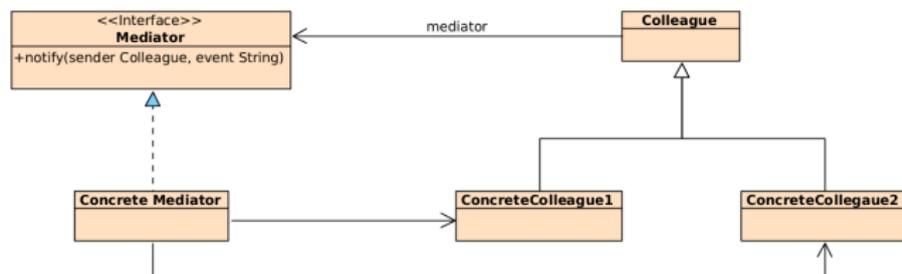
Example



Example



Mediator – Struttura



● Partecipanti:

- **Mediator**: definisce un'interfaccia per comunicare con gli oggetti `Colleague`
- **ConcreteMediator**: implementa il comportamento di coordinamento tra i vari oggetti `Colleague`. Conosce e mantiene i vari oggetti `Colleague`
- **Colleague**: ogni `Colleague` conosce il `Mediator` e comunica solo con esso

Mediator – Struttura

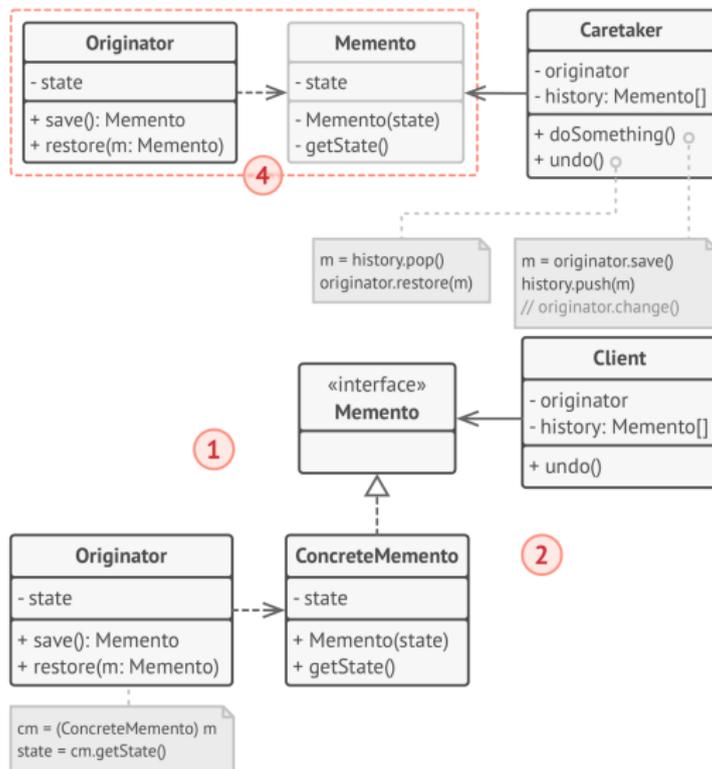
- **Applicabilità:**
 - quando è difficile cambiare una classe perché ha molte dipendenze
 - difficile riusare una classe perché si porta dietro molte dipendenze
 - quando al fine di riusare del comportamento si usa pesantemente ereditarietà
- **Conseguenze:**
 - riduzione di accoppiamento
 - Open/closed favorito tramite introduzione di ulteriori mediator senza cambiare comportamento degli oggetti `Colleague`
 - rischio di “God Object”
- **Implementazione:** ...
- **Utilizzi Noti:** ...
- **Pattern Correlati:** Observer, Command, Façade ...

Memento

- **Scopo:** senza violare l'incapsulamento si vuole catturare e esternalizzare lo stato interno di un oggetto, cosicché l'oggetto possa essere recuperato in un momento successivo
- **Motivazione:** in alcuni casi si vogliono realizzare dei checkpointing per un oggetto in modo ad esempio di poter procedere con azioni di "UNDO".

<https://refactoring.guru/design-patterns/memento>

Memento – Struttura



Memento – Struttura

- **Partecipanti:**

- **Memento:** immagazzina lo stato dell'oggetto `Originator`. La quantità di informazioni salvate è a discrezione dell'oggetto `Originator`. L'accesso alle informazioni dell'oggetto è possibile al solo `Originator` mentre il `Caretaker` ha una visibilità ridotta dell'oggetto `Memento`.
- **Originator:** crea gli snapshot del suo stato e può usare il `Memento` per recuperare lo stato
- **Caretaker:** è responsabile per il salvataggio dei `Memento` e non accede alle informazioni in esso contenute

Memento

- **Applicabilità:**
 - da usare per produrre snapshot di un altro oggetto per recuperare uno stato precedente
 - quando accesso diretto agli attributi o ai metodi set e get violerebbe incapsulamento
 -
- **Conseguenze:**
 - esternalizzazione dello stato da parte di un oggetto che coordina tale attività senza fornire accesso allo stesso
 - uso di memoria va considerato
 - possibilità di “gettare” oggetti “vecchi”
- **Implementazione:** ...
- **Utilizzi Noti:** ...
- **Pattern Correlati:** Command, Prototype ...

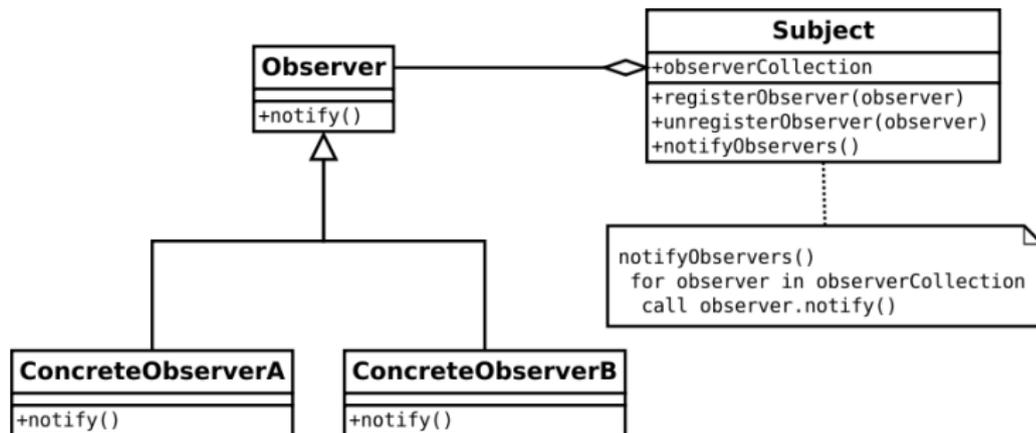
Observer

- **Scopo:** definisce una relazione uno-a-molti tra oggetti cosicché quando l'oggetto (uno) cambia stato tutti gli altri oggetti (molti) coinvolti vengano informati del cambiamento
- **Motivazione:** c'è bisogno di mantenere consistenza tra gli oggetti correlati
- **Applicabilità:** quando una modifica ad un oggetto richiede modifiche ad altri oggetti e non è noto a priori quali saranno tali oggetti

<https://refactoring.guru/design-patterns/observer>

Observer

- Struttura:



Observer

- **Conseguenze:**

- Accoppiamento astratto e minimale tra subject e observer:
subject conoscono lista di observer
- Supporto per comunicazioni broadcast: fornisce meccanismi per implementare comunicazione di un l'evento è da notificare a tutti coloro abbiano manifestato interesse ad un evento. Aggiunta e rimozione di osservatori estremamente semplice.
- Aggiornamenti inattesi: modifiche apportate da un osservatore potrebbero scatenare sequenza costosa di invocazioni senza che questo possa essere facilmente stabilito a priori.

- **Partecipanti:**

- **Subject:** conosce i suoi "osservatori" e fornisce meccanismi per il collegamento e lo scollegamento degli osservatori
- **Observer:** fornisce un'interfaccia per la notifica
- **ConcreteSubject:** contiene lo stato ed invia le notifiche
- **ConcreteObserver:** gestisce il riferimento al soggetto e mantiene l'informazione sullo stato dell'oggetto osservato.

Observer

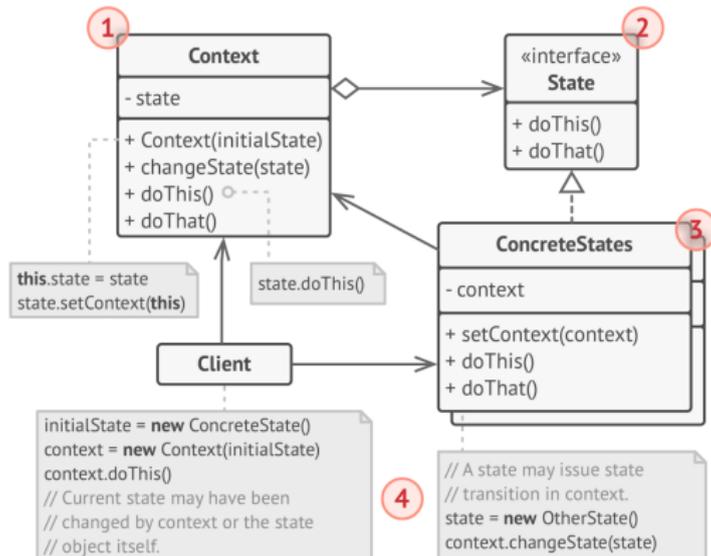
- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati:

State

- **Scopo:** permettere ad un oggetto di modificare il suo comportamento quando il suo stato interno cambia. L'oggetto può apparire come se avesse modificato la sua classe.
- **Motivazione:** In molti contesti è possibile voler avere oggetti che rappresentino comportamenti rappresentabili da macchine a stati. Si pensi ad oggetti che rappresentano connessioni di un protocollo di comunicazione.

<https://refactoring.guru/design-patterns/state>

State – Struttura



- **Partecipanti:**

- **State:** è un'interfaccia che dichiara i metodi il cui comportamento cambia quando lo stato dell'oggetto cambia.
- **ConcreteStates:** per ogni stato dell'oggetto ci sarà un oggetto concreto che conterrà i dettagli implementativi e comportamentali correlati allo specifico stato. In generale potrebbe includere la conoscenza sul cambio di stato.
- **Context:** ha un riferimento ad uno stato che rappresenta lo stato attuale dell'oggetto. Reindirizza le invocazioni relative al comportamento a tale oggetto stato incapsulato.

State

- **Applicabilità:**

- si usa il pattern quando si vuole implementare oggetti che cambiano il comportamento al cambiare del loro stato. Chiaramente se il numero di stati è non minimo.
- può essere utile quando ci si rende conto che la classe include molto comportamento all'interno di case statement

- **Conseguenze:**

- Single responsibility principle permette di organizzare il comportamento in classi separate. Un oggetto per ogni singolo stato.
- open/closed principle è ben supportato potendo facilmente aumentare il comportamento senza dover modificare i client
- semplificazione della gestione dell'oggetto

- **Implementazione:** ...

- **Utilizzi Noti:** ...

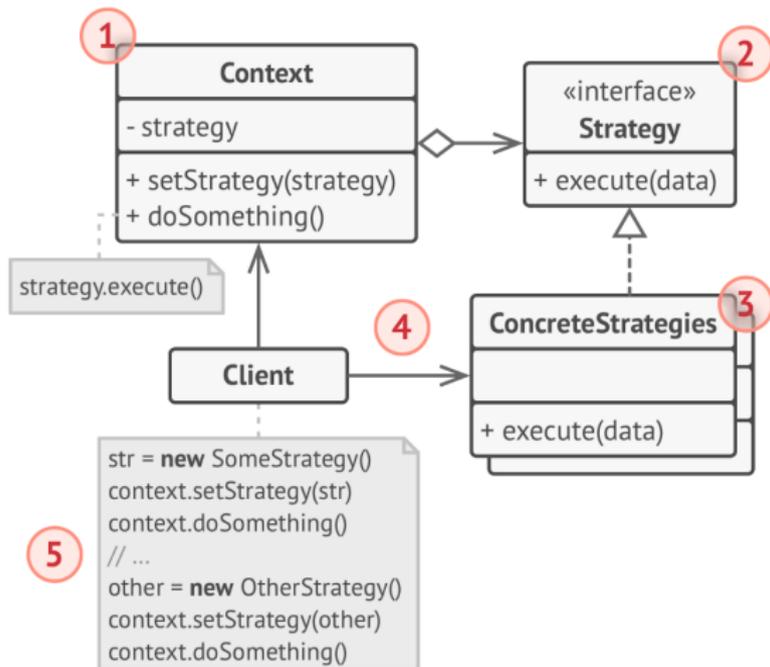
- **Pattern Correlati:** Strategy ...

Strategy

- **Scopo:** permettere di definire una famiglia di algoritmi e renderli intercambiabili. Fare in modo di rendere indipendenti gli algoritmi dai loro utenti.
- **Motivazione:** In molti contesti è necessario avere classi che implementano algoritmi per la manipolazione di dati ma che possano cambiare sulla base di specifici parametri. Si consideri ad esempio algoritmi per la formattazione di un testo o per il calcolo del percorso migliore. Implementare l'algoritmo in modo estremamente generico con dozzine di punti di configurazione e di scelta risulta poco flessibile e manutenibile.

<https://refactoring.guru/design-patterns/strategy>

Strategy – Struttura



Strategy – Struttura

- **Partecipanti:**

- **Strategy:** è un interfaccia che dichiara i metodi che sono rilevanti per la specifica famiglia di algoritmi
- **ConcreteStrategies:** per ogni algoritmo si implementerà un oggetto concreto che conterrà i dettagli implementativi e comportamentali correlati allo specifico algoritmo.
- **Context:** ha un riferimento ad uno specifico algoritmo che può poi essere settato sulla base di specifici parametri.

- **Applicabilità:**

- si usa il pattern quando si vuole avere la possibilità di fornire un servizio sulla base di differenti algoritmi e poterli cambiare anche a run-time
- si può usare quando ci si accorge di avere molte classi simili che differiscono per come raggiungono l'obiettivo
- quando ci si accorge di avere molti comandi switch per dirigere uno specifico comportamento

Strategy

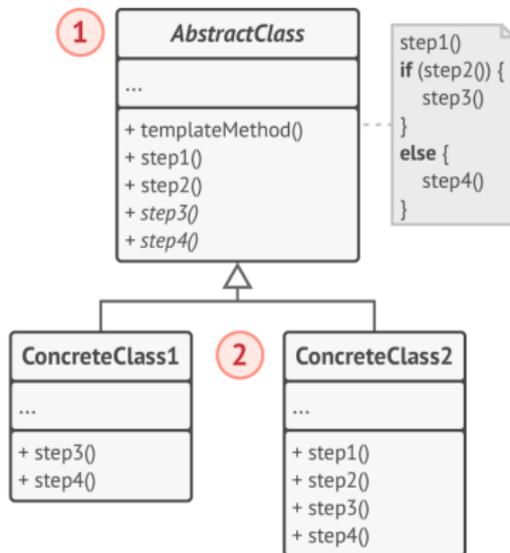
- **Conseguenze:**
 - dinamicità a tempo di esecuzione
 - isolamento del codice che implementa algoritmo dai suoi clienti
 - Open/closed principle possibilità di estendere comportamento senza impattare sui clienti
 - è necessario che i clienti siano però a conoscenza dei parametri che modificano il comportamento
 - generalmente non vale la pena introdurlo per poche varianti
- **Implementazione:** ...
- **Utilizzi Noti:** ...
- **Pattern Correlati:** State ...

Template Method

- **Scopo:** definire alcuni passi di un algoritmo per poi poter dettagliare o ridefinire il comportamento in classi più specifiche
- **Motivazione:** si possono presentare delle situazioni in cui oggetti differenti potrebbero essere manipolati utilizzando algoritmi simili ma che si distinguono per alcuni passi dipendenti dal parametro stesso.

<https://refactoring.guru/design-patterns/template>

Template Method – Struttura



Template Method – Struttura

- **Partecipanti:**
 - **AbstractClass:** è una classe astratta che include la specifica dello scheletro dell'algoritmo ed eventualmente la specifica di alcuni suoi passi. Definisce poi i passi necessari a svolgere il comportamento atteso
 - **ConcreteClasses:** implementano specifici passi dell'algoritmo sulla base delle specifiche necessità e caratteristiche del tipo a cui sono associate

Template Method

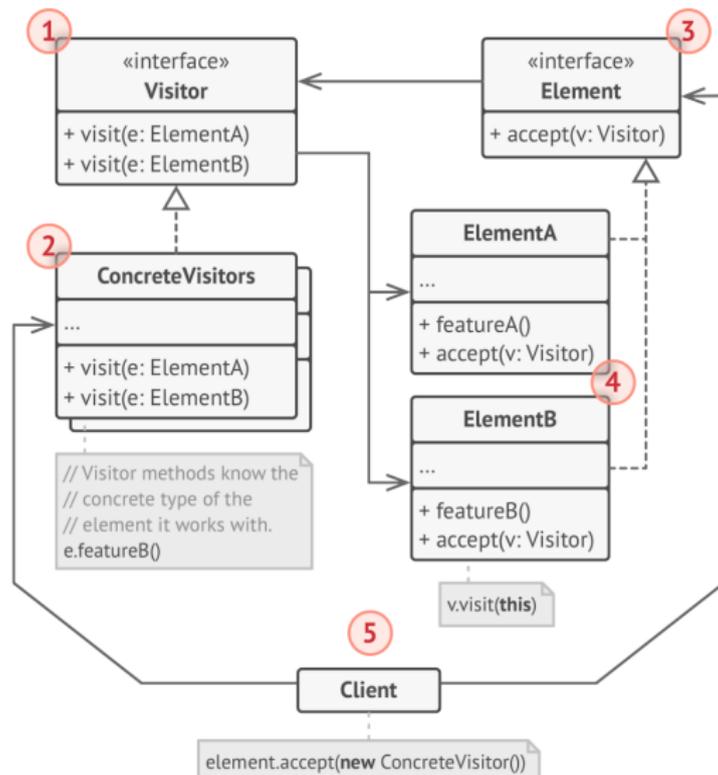
- **Applicabilità:**
 - si usa il pattern quando si vuole far sì che specifici clienti estendano alcuni passi di un algoritmo ma non il processo
 - si può usare quando ci si accorge di avere molte classi simili che differiscono per aspetti minori e piccoli passi. In tal modo si ottiene una maggiore manutenibilità dell'algoritmo principale stesso
- **Conseguenze:**
 - maggiore manutenibilità
 - bisogna fare attenzione a che sottoclassi non violino la sostituibilità cancellando passi o violando le condizioni dei passi intermedi
- **Implementazione:** ...
- **Utilizzi Noti:** ...
- **Pattern Correlati:** Factory method caso particolare di Template method. Strategy si basa su composizione mentre Template method usa ereditarietà, condividono comunque parzialmente le motivazioni ...

Visitor

- **Scopo:** si vogliono separare gli algoritmi dagli oggetti su cui operano
- **Motivazione:** il contesto generico è quello di strutture dati complesse che presentano elementi eterogenei. Si pensi ad un albero i cui nodi possono rappresentare oggetti di differente natura, quale ad esempio un AST, o un grafo rappresentante città ed elementi della stessa. Nella visita di tali strutture si vorrebbero applicare processi differenti nella manipolazione degli stessi.

<https://refactoring.guru/design-patterns/visitor>

Visitor – Struttura



Visitor – Struttura

- **Partecipanti:**

- **Visitor:** è l'interfaccia che dichiara l'algoritmo di visita. L'interfaccia è nota agli elementi visitabili che invocano i metodi in essa definiti
- **ConcreteVisitor:** implementano i comportamenti relativi alla manipolazione degli specifici oggetti. Possono ad esempio immagazzinare informazioni in relazione alla visita e ai nodi incontrati.
- **Element:** accettano i visitor e invocano i metodi di visita
- **ConcreteElement:**

- **Applicabilità:**

- si usi il pattern quando si devono applicare specifiche manipolazioni a tutti gli elementi di una struttura dati complessa
- si può usare quando certi comportamenti hanno significato solo per specifici nodi mentre per altri può aver senso "skippare"

Visitor

- **Conseguenze:**
 - Open/closed principle viene supportato dalla possibilità di modificare/estender il comportamento del visitor senza impattare sul cliente
 - Single responsibility ne beneficia in quanto ci permette di isolare il comportamento in classi specificamente definite allo scopo
 - estensione della struttura dati è complicata e richiede di modificare i visitor
- **Implementazione:** ...
- **Utilizzi Noti:** ...
- **Pattern Correlati:** ...