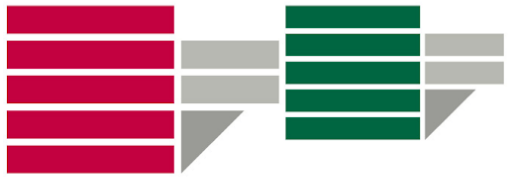
Design Patterns

UNIVERSITÀ DELLA CALABRIA



DIMES - Dipartimento di INGEGNERIA INFORMATICA
MODELLISTICA, ELETTRONICA E SISTEMISTICA

Ing. Ludovica Sacco
DIMES – UNICAL - 87036 Rende(CS) - Italy
Email: l.sacco@dimes.unical.it

Design Patterns

Tabella di riepilogo

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

Classificazione Design Pattern

Un primo criterio di classificazione dei Pattern riguarda lo scopo (*purpose*):

- **creazionali** → riguardano il processo di creazione di oggetti
- **strutturali** → utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti (si basano sui concetti di ereditarietà e polimorfismo)
- **comportamentali** → si occupano di come gli oggetti interagiscono reciprocamente e distribuiscono tra di essi le responsabilità

Un secondo criterio riguarda il raggio di azione (*scope*):

- **classi:** pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sul concetto di ereditarietà e sono quindi s***tatiche***, definite a tempo di compilazione
- **oggetti:** pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più ***dinamiche***

ADAPTER

→ STRUTTURALE

→ BASATO SU CLASSI E OGGETTI

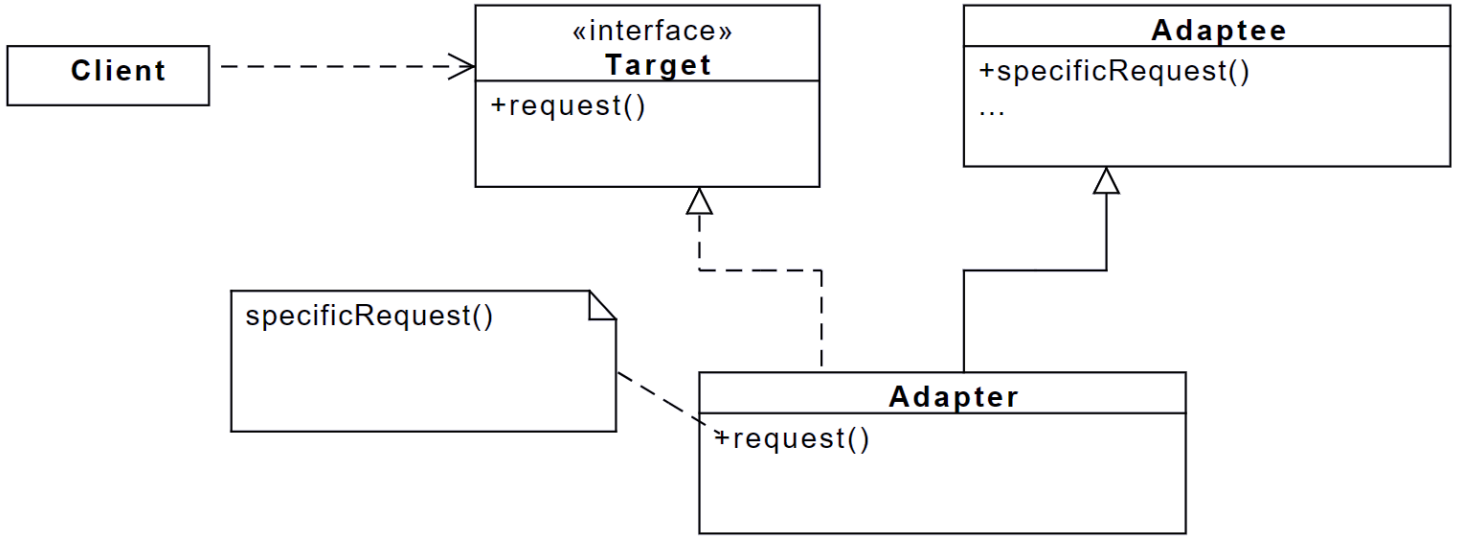
SCOPO

Fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti (es. librerie già esistente da adattare nel nostro progetto)

APPLICABILITÀ

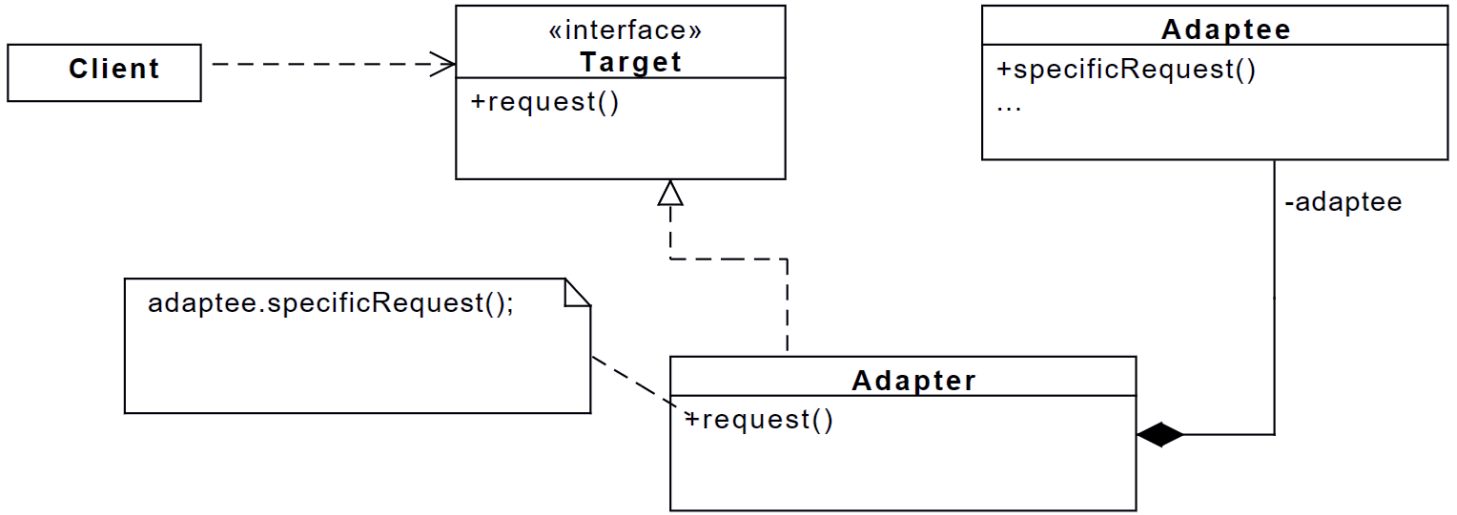
- Utilizzo di una classe esistente che presenti un'interfaccia diversa da quella desiderata;
- Scrittura di una determinata classe senza poter conoscere a priori le altre classi con cui dovrà operare, in particolare senza poter conoscere quale specifica interfaccia sia necessario che la classe debba presentare alle altre.

ADAPTER



CLASS ADAPTER

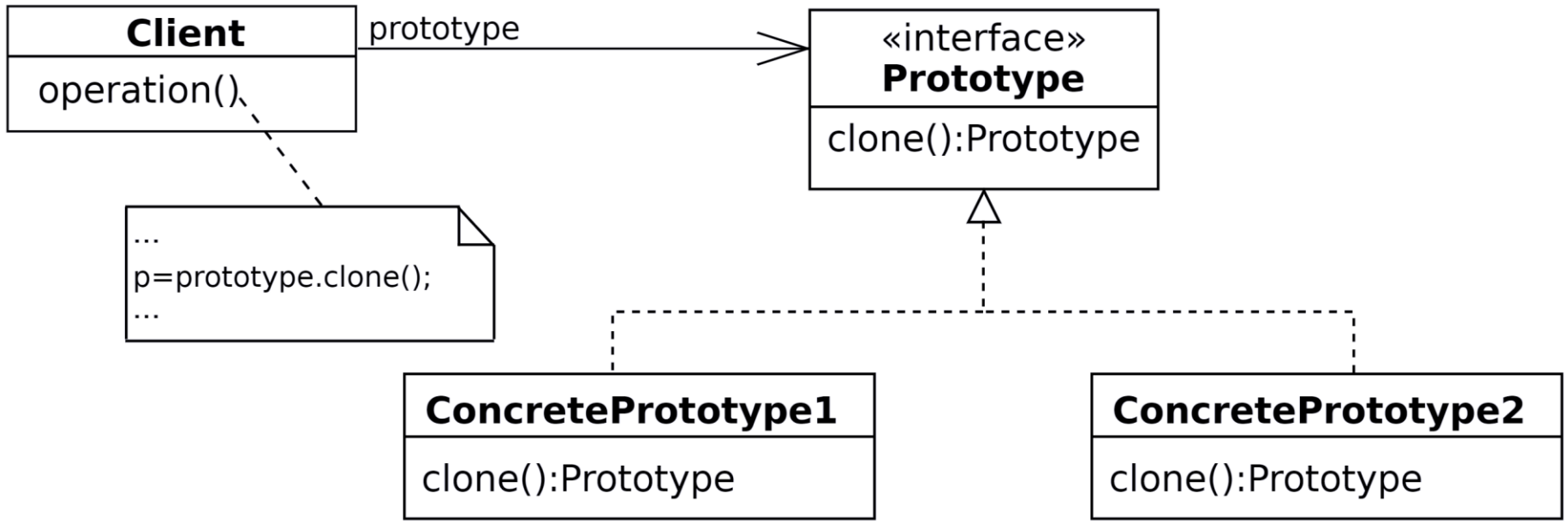
OBJECT ADAPTER



Prototype

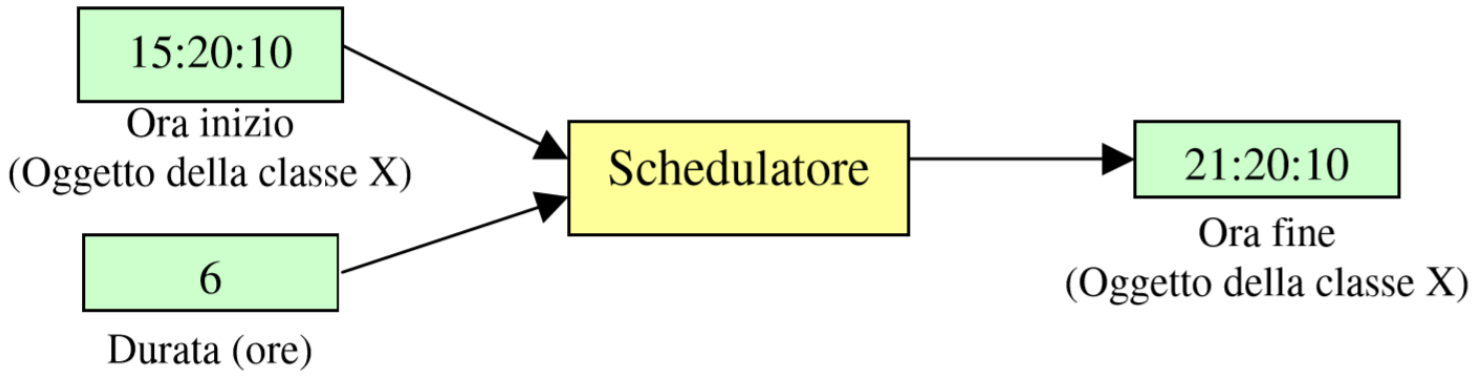
PROTOTYPE
→ CREAZIONALE
→ BASATO SU OGGETTI

- Specifica i tipi di oggetti da creare, utilizzando un'istanza prototipo e crea nuove istanze tramite la copia di questo prototipo.



Prototype

Es. Schedulatore in grado di eseguire operazioni su oggetti che rappresentano istanti di tempo.



Input:
- Ora inizio attività
- Durata

Output
- Ora fine attività = somma della ora di inizio e la durata dell'attività

Soluzione
Clonazione di oggetti utilizzati come prototipi.

Questi oggetti devono implementare una interfaccia che offre un servizio di copia dell'oggetto, che sarà quella di cui il framework avrà conoscenza al momento di dover creare nuovi oggetti.

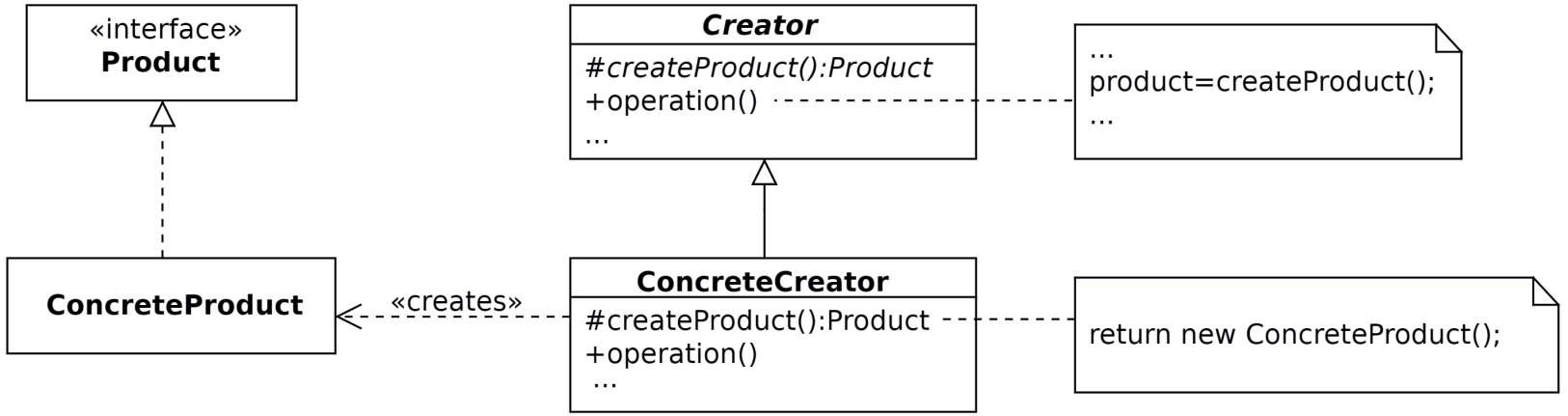
FACTORY METHOD

FACTORY METHOD

→ CREAZIONALE

→ BASATO SU CLASSI

- Definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione del tipo di classe da istanziare



Es. 1 Framework per la manipolazione di elementi cartografici.

Osservazioni:

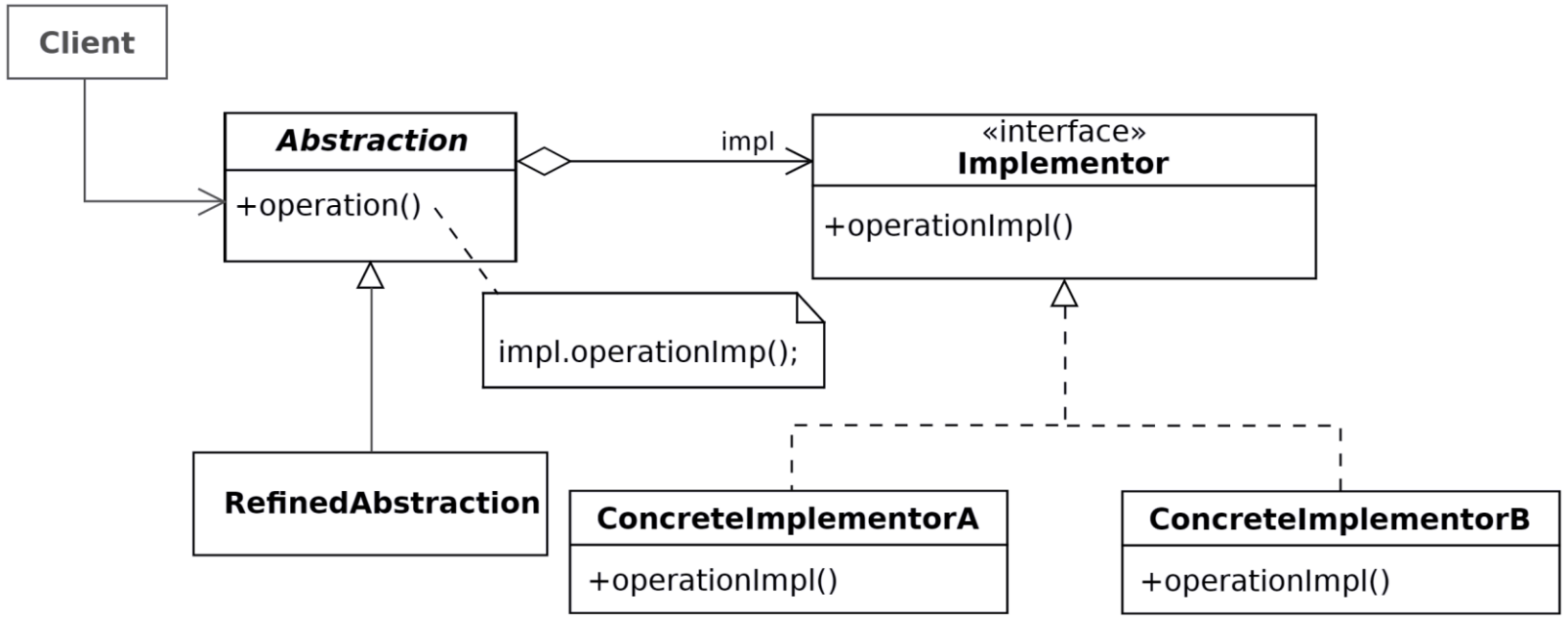
Si vuole rendere noto che il factory method (newElement) dichiara come tipo da restituire al punto di chiamata, sia nel Creator (ElementHandler), sia in ogni ConcreteCreator (PlaceHandler e ConnectorHandler), un oggetto di tipo Product (MapElement), invece dei particolari tipi da produrre (Place e Connector). Questo è dovuto al fatto che le sottoclassi che ridefiniscono un metodo devono esplicitare lo stesso tipo di ritorno di quello indicato nella dichiarazione del metodo nella superclasse.

Es. 2 Gestione contratti telefonici e televisivi, archivio clienti.

BRIDGE

BRIDGE
→ STRUTTURALE
→ BASATO SU OGGETTI
- Separa un'astrazione dalla sua implementazione, in modo che entrambe possano variare indipendentemente.

Il pattern "Bridge" suggerisce la separazione dell'astrazione dall'implementazione in gerarchie diverse, legando oggetti della seconda a quelli della prima tramite un relazione di composizione.



Es. Stampanti «Bianco e Nero» e a «Colori»

CHAIN-OF-RESPONSIBILITY

→ COMPORTAMENTALE
→ BASATO SU OGGETTI

SCOPO
Evita di accoppiare il mittente di una richiesta al suo destinatario dando a più di un oggetto la possibilità di gestire la richiesta.
Permette agli oggetti riceventi di passare la richiesta lungo la catena, fino a quando un oggetto non lo gestisce.

DESCRIZIONE
Il primo oggetto nella catena riceve la richiesta, o la gestisce o la inoltra al candidato successivo sulla catena e così via. L'oggetto che ha effettuato la richiesta non ha una conoscenza esplicita di chi la gestirà, si parla di *implicitReceiver*.

Per inoltrare la richiesta lungo la catena e garantire che i ricevitori rimangano impliciti, ogni oggetto sulla catena condivide un'interfaccia comune per la gestione delle richieste e per l'accesso al successore sulla catena.

CHAIN-OF-RESPONSIBILITY

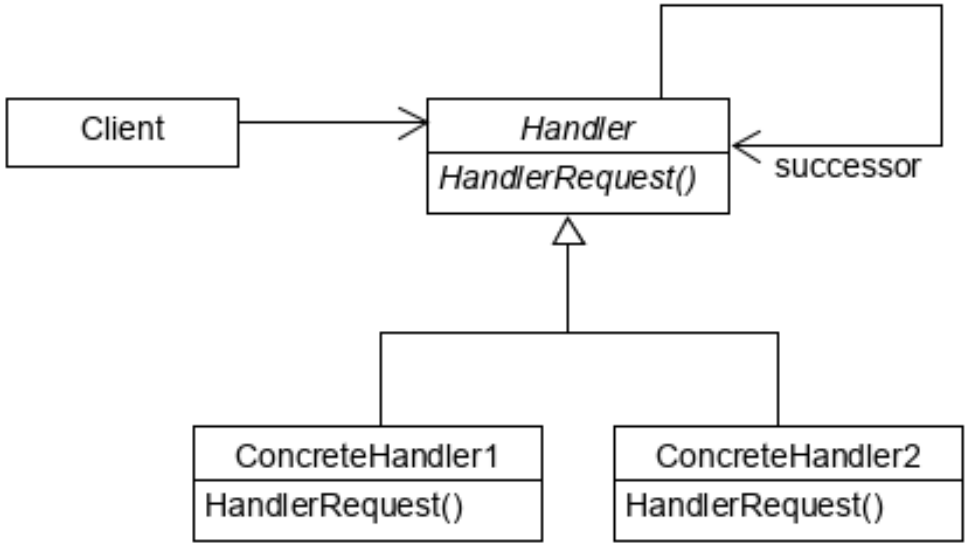
APPLICABILITÀ

Si utilizza il pattern *chain-of-responsibility* quando:

- più di un oggetto può gestire una richiesta e il gestore non è noto a priori.
- si desidera inviare una richiesta a uno di più oggetti senza specificare esplicitamente il destinatario
- l'insieme di oggetti in grado di gestire una richiesta deve essere specificato in modo dinamico.

CHAIN-OF-RESPONSIBILITY

STRUTTURA E PARTECIPANTI



- **Handler**
- definisce un'interfaccia per la gestione delle richieste
- (facoltativo) implementa il successivo collegamento

- **ConcreteHandler**
- gestisce le richieste per le quali è responsabile
- può accedere al suo successore
- se ConcreteHandler è in grado di gestire la richiesta, lo fa; altrimenti inoltra la richiesta al suo successore.

- **Client**
- avvia la richiesta a un oggetto ConcreteHandler sulla catena

CHAIN-OF-RESPONSIBILITY

VANTAGGI/SVANTAGGI

1. ***Riduce l'accoppiamento***

   ↳ Il richiedente deve solo sapere che la propria richiesta verrà gestita.
   Non ha bisogno di sapere chi realmente gestirà la richiesta.

2. ***Flessibilità***

   ↳ La catena della responsabilità può essere modificata senza condizionare il richiedente.

3. ***Risposta non attesa***

   ↳ La richiesta viene presa in carico e propagata nella catena di responsabilità, ma potrebbe non esserci il responsabile con conseguente non gestione della richiesta.

CHAIN-OF-RESPONSIBILITY

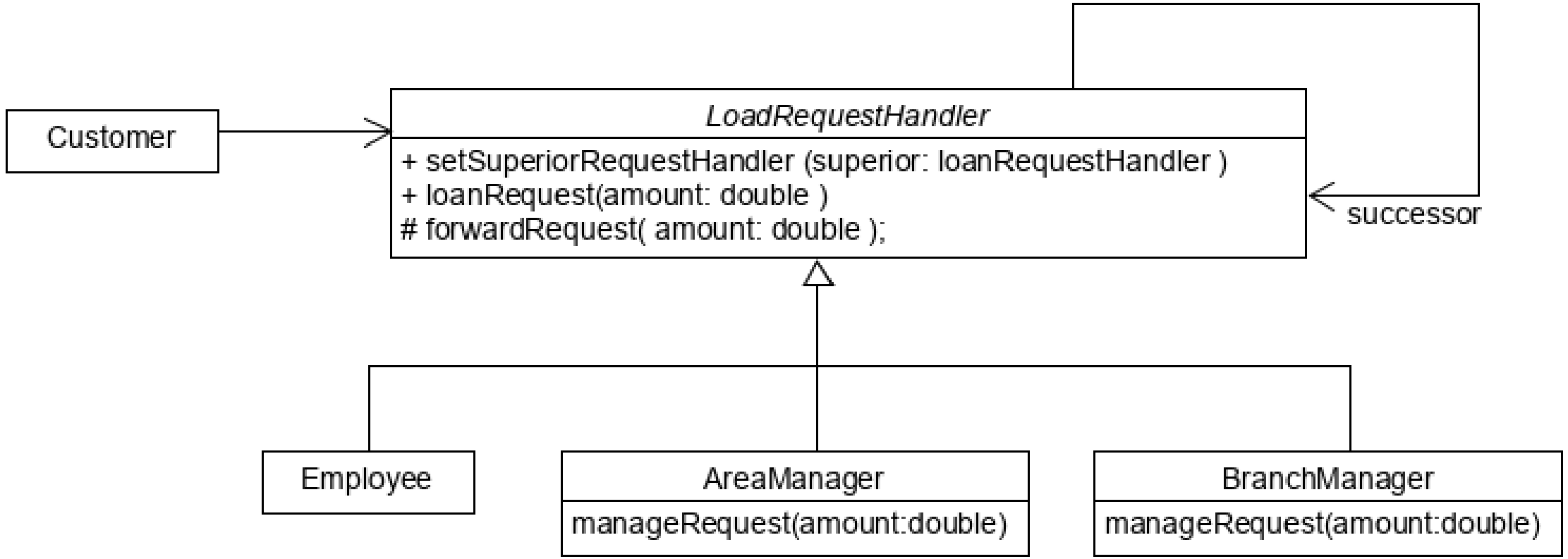
ESEMPIO

Una banca per gestire le richieste di concessione di mutui provenienti dai propri clienti (*clients*), internamente si organizza con tre figure (*Impiegato*, *Direttore filiale*, *Direttore di zona*) aventi responsabilità diverse che dipendono dall'importo da approvare.

Al livello più basso (*Impiegato*) viene consentito l'approvazione di richieste fino a 30.000€. Le richieste che superano questo importo sono gestite dal *Direttore di filiale*, il quale ha un altro importo massimo di 80.000€ da gestire. Richieste che vanno oltre quest'ultimo importo, vanno gestite dal *Direttore di zona*.

Il problema riguarda la definizione di un meccanismo di inoltro delle richieste, senza che il Client conosca la struttura organizzativa interna alla banca.

CHAIN-OF-RESPONSIBILITY



MEDIATOR

→ COMPORTAMENTALE
→ BASATO SU OGGETTI

SCOPO

Definisce un oggetto che incapsula il modo in cui un insieme di oggetti interagisce.
Il mediatore promuove l'accoppiamento libero impedendo agli oggetti di riferirsi esplicitamente tra loro e consente di variare la loro interazione in modo indipendente.

MOTIVAZIONE E DESCRIZIONE

La programmazione orientata agli oggetti incoraggia la distribuzione di comportamento tra gli oggetti, portando a una struttura con molte connessioni tra oggetti (nel peggiore dei casi, ogni oggetto finisce per conoscersi).

Il partizionamento di un sistema in molti oggetti generalmente migliora la riusabilità, ma quando si hanno molte interconnessioni si tende a ridurla di nuovo (molte interconnessioni rendono meno probabile che un oggetto possa funzionare senza il supporto di altri).

MEDIATOR

Inoltre, può essere difficile modificare il comportamento del sistema in modo significativo, poiché il comportamento è distribuito tra molti oggetti.

Di conseguenza, si potrebbe essere costretti a definire molte sottoclassi per personalizzare il comportamento del sistema.

È possibile evitare questi problemi incapsulando il comportamento collettivo in un oggetto ***mediator*** separato. Un mediatore è responsabile del controllo e del coordinamento delle interazioni di un gruppo di oggetti. Funge da intermediario che impedisce agli oggetti nel gruppo di riferirsi in modo esplicito. Gli oggetti conoscono solo il mediatore, riducendo così il numero di interconnessioni.

MEDIATOR

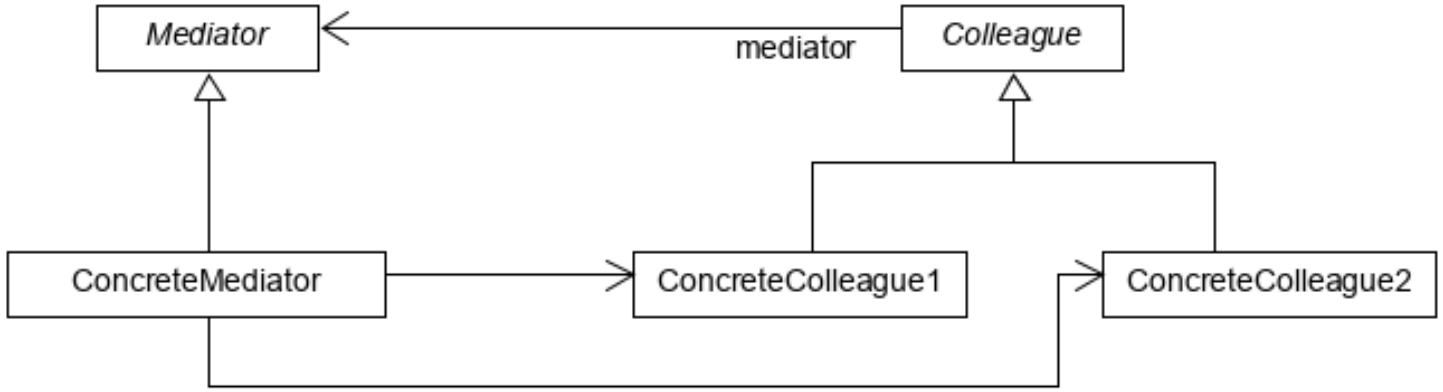
APPLICABILITÀ

Si usa il pattern *mediator* quando:

- un insieme di oggetti comunica in modi ben definiti ma complessi. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere
- riutilizzare un oggetto è difficile perché si riferisce e comunica con molti altri oggetti.
- un comportamento distribuito tra più classi dovrebbe essere personalizzabile senza molte sottoclassi.

MEDIATOR

STRUTTURA E PARTECIPANTI



- **Mediator**
- Definisce una interfaccia comune per la gestione della comunicazioni tra *Colleagues*;
- **ConcreteMediator**
- Mantiene la lista dei *Colleagues* e implementa il comportamento cooperativo tramite la coordinazione dei *Colleagues*;
- **Colleague**
- Entità che possiede un riferimento al *Mediator*;
- Definisce un metodo di notifica di eventi al *Mediator*
- **ConcreteCollegue:**
- Comunicano gli eventi al Mediator invece di interagire con i *Collegues*

MEDIATOR

VANTAGGI/SVANTAGGI

\+ ***1. Limita la sottoclasse***

↳ Un mediatore localizza comportamenti che altrimenti verrebbero distribuiti tra più oggetti.

\+ ***2. Disaccoppia i colleghi***

↳ Si possono riutilizzare le classi *Colleague* e *Mediator* in modo indipendente.

\+ ***3. Semplifica i protocolli degli oggetti***

↳ Un mediatore sostituisce le interazioni molti-a-molti con interazioni uno-a-molti tra il mediatore e i suoi colleghi.

\+ ***4. Astrae il modo in cui gli oggetti cooperano***

↳ Rende la mediazione un concetto indipendente e la incapsula in un oggetto, consente di concentrarsi su come gli oggetti interagiscono, invece che sul loro comportamento individuale

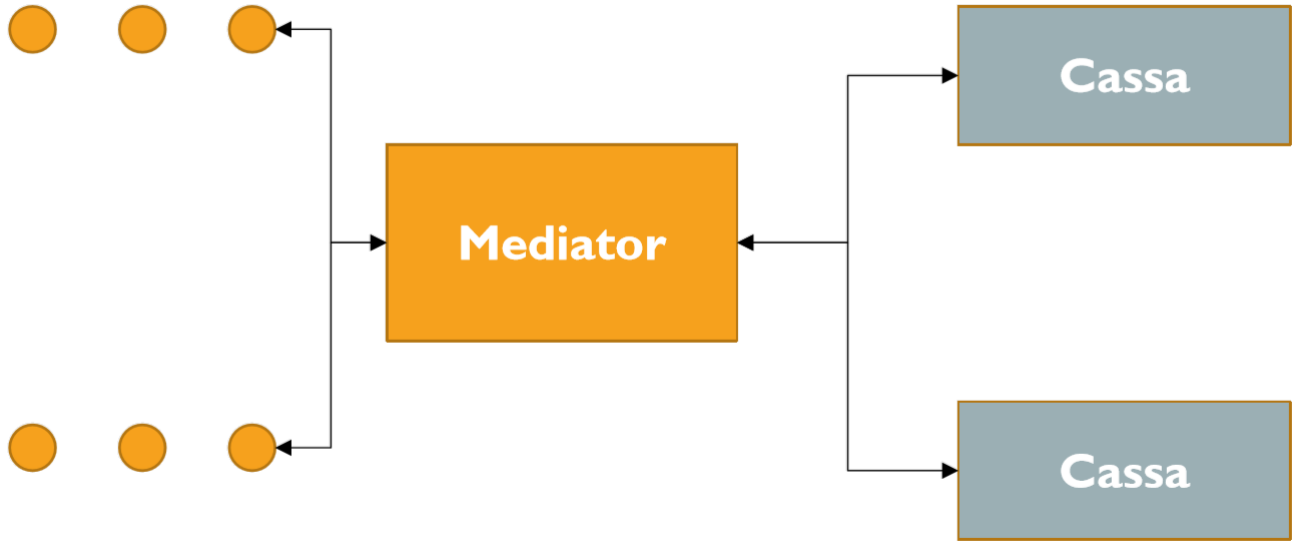
\- ***5. Centralizza il controllo***

↳ Un *Mediator* incapsula i protocolli e può diventare più complesso di qualsiasi singolo *Colleague*. Ciò può rendere il *Mediator* stesso monolitico e difficile da mantenere.

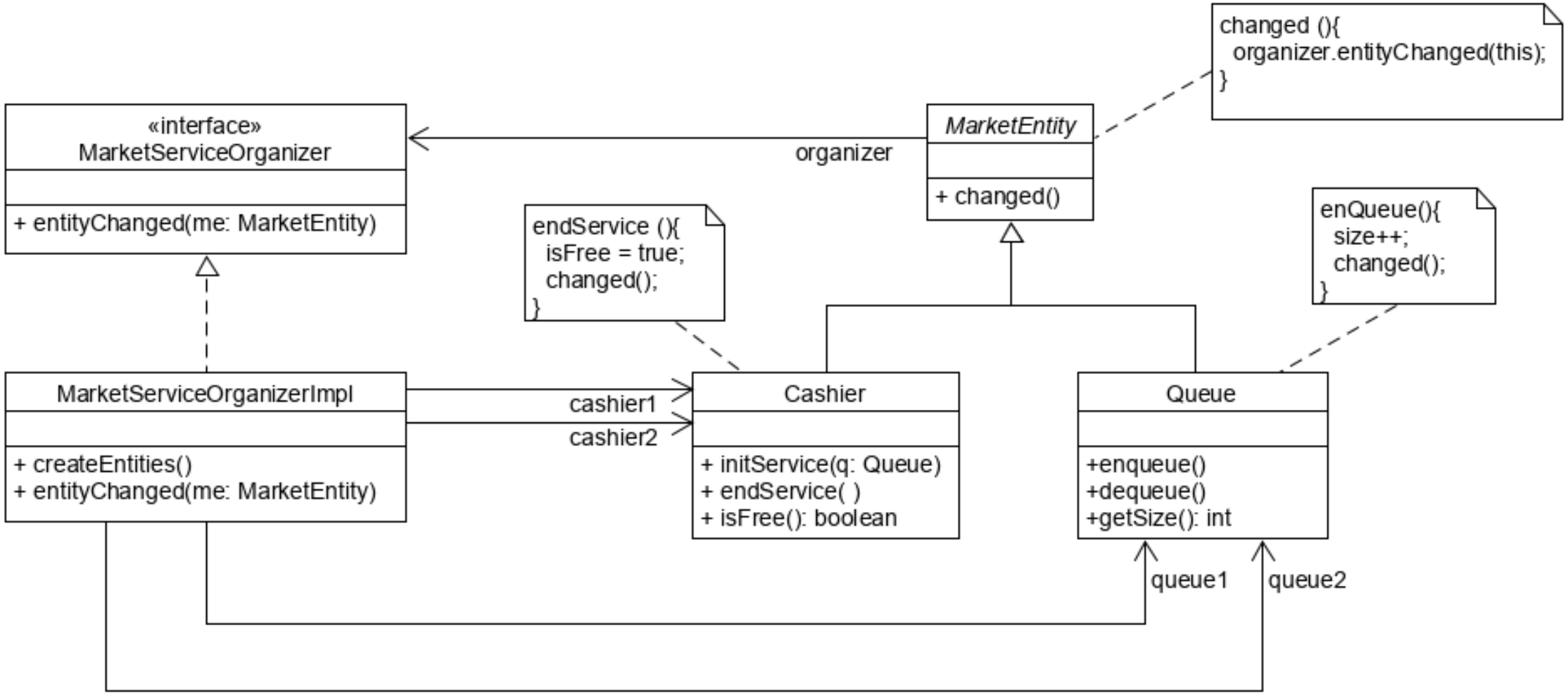
MEDIATOR

ESEMPIO

Un piccolo supermercato è organizzato con due cassiere che servono i clienti in coda per pagare la propria spesa.



MEDIATOR


changed (){
organizer.entityChanged(this);
}
«interface»
MarketServiceOrganizer
+ entityChanged(me: MarketEntity)
organizer
MarketEntity
+ changed()
endService (){
isFree = true;
changed();
}
enQueue(){
size++;
changed();
}
MarketServiceOrganizerImpl
+ createEntities()
+ entityChanged(me: MarketEntity)
cashier1
cashier2
Cashier
+ initService(q: Queue)
+ endService( )
+ isFree(): boolean
Queue
+enqueue()
+dequeue()
+getSize(): int
queue1
queue2

FAÇADE

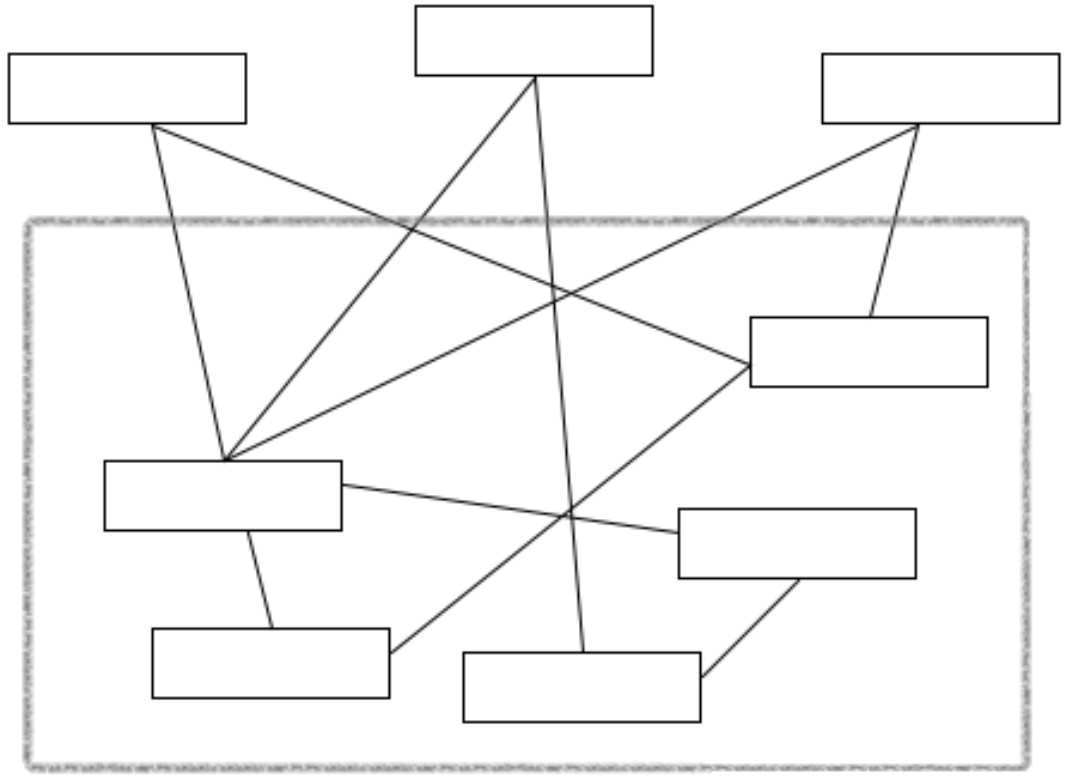
→ STRUTTURALE
→ BASATO SU OGGETTI

SCOPO
Fornire un'interfaccia unificata a un set di interfacce in un sottosistema.
Façade definisce un'interfaccia di livello superiore che semplifica l'utilizzo del sottosistema.

MOTIVAZIONE E DESCRIZIONE
Strutturare un sistema in sottosistemi aiuta a ridurre la complessità.
Un obiettivo di progettazione comune è ridurre al minimo la comunicazione e le dipendenze tra i sottosistemi.

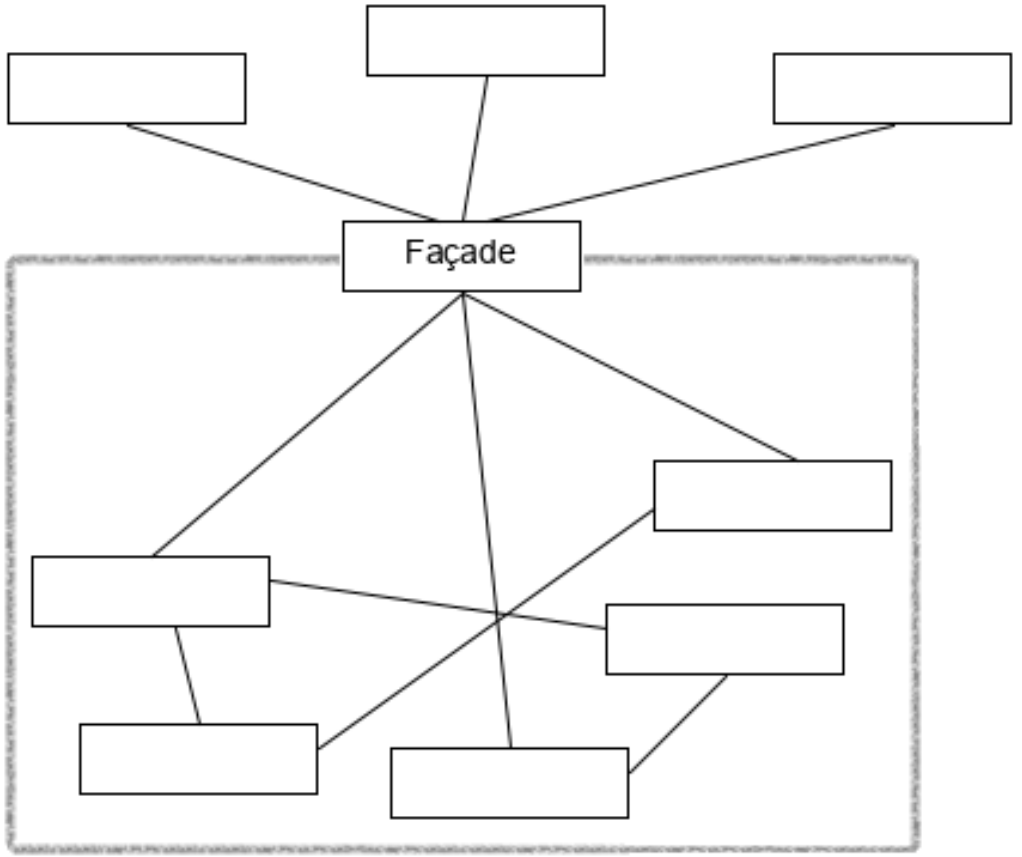
Un modo per raggiungere questo obiettivo è introdurre un oggetto «facciata» che fornisce un'unica interfaccia semplificata alle strutture più generali di un sottosistema. Alcune applicazioni specializzate potrebbero dover accedere direttamente a queste classi. Ma la maggior parte dei client di un compilatore in genere non si preoccupa di dettagli come l'analisi e la generazione di codice, vogliono semplicemente compilare del codice.

FAÇADE



client classes

subsystem classes



FAÇADE

APPLICABILITÀ

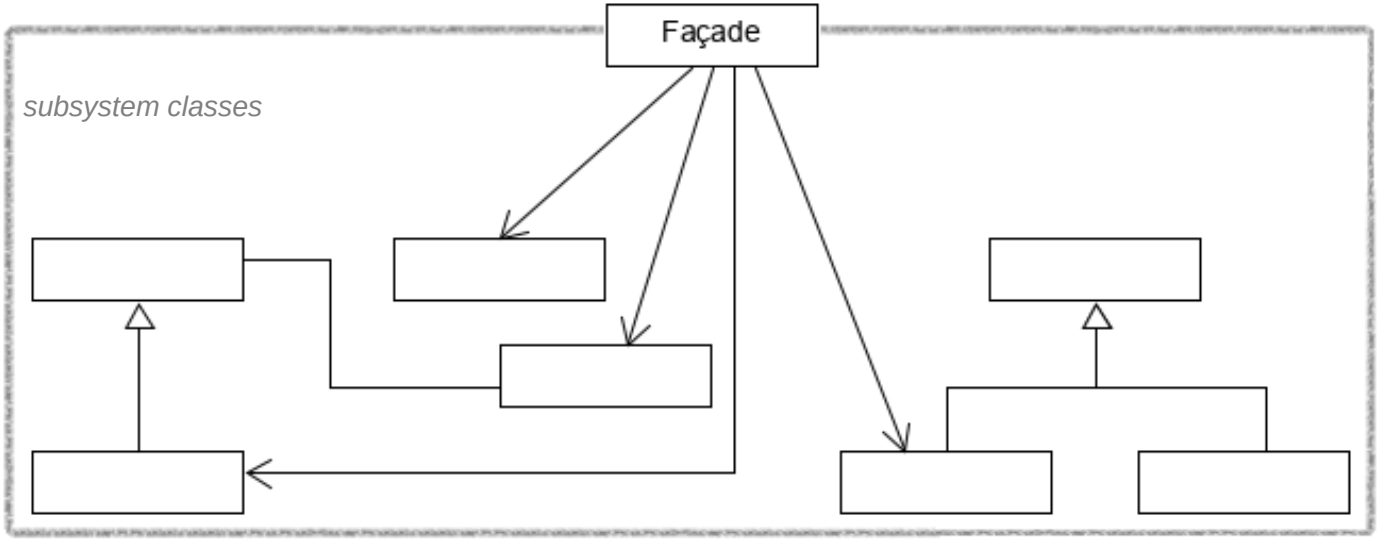
Si usa il pattern *façade* quando:

- Si desidera fornire una semplice interfaccia a un sistema complesso. I sottosistemi spesso diventano più complessi man mano che si evolvono. Ciò rende il sottosistema più riutilizzabile e più facile da personalizzare, ma difficile da utilizzare per i client che non hanno bisogno di personalizzarlo.

  Una facciata può fornire una semplice visualizzazione predefinita del sottosistema, abbastanza buona per la maggior parte dei client. Solo i clienti che necessitano di maggiore personalizzazione dovranno guardare oltre la facciata.
- Ci sono molte dipendenze tra i client e le classi di implementazione di un'astrazione. Introdurre una facciata per separare il sottosistema dai client e da altri sottosistemi, promuovendo così **l'indipendenza** e la **portabilità** del sottosistema.
- Si desidera stratificare i sottosistemi. Utilizzare una facciata per definire un punto di ingresso per ciascun livello di sottosistema.

FAÇADE

STRUTTURA E PARTECIPANTI



- **Façade**
- Ha conoscenza delle funzionalità di ogni classe del sottosistema.
- Delega agli appropriati oggetti del sottosistema ogni richiesta pervenuta dall'esterno.

- **Subsystem classes**
- Implementano le funzionalità del sottosistema
- Gestiscono le attività assegnate dal *Façade*
- Non hanno riferimenti verso il *Façade*.

FAÇADE

VANTAGGI/SVANTAGGI:

1. **Riduce il numero di associazioni**
   - disaccoppiando il Client dal Sistema è possibile ridurre il numero di associazioni tra i due attori.

2. **Agevola il cambiamento**
   - il basso accoppiamento rende possibile modificare il Sistema senza dover notificare il Client.

3. **Consente l'uso diretto del Sistema**
   - il Client può, se necessario, utilizzare direttamente il Sistema.

ESEMPIO

Motore di un auto composto da molti componenti di cui il cliente non ha sempre bisogno di essere a conoscenza.

MEMENTO

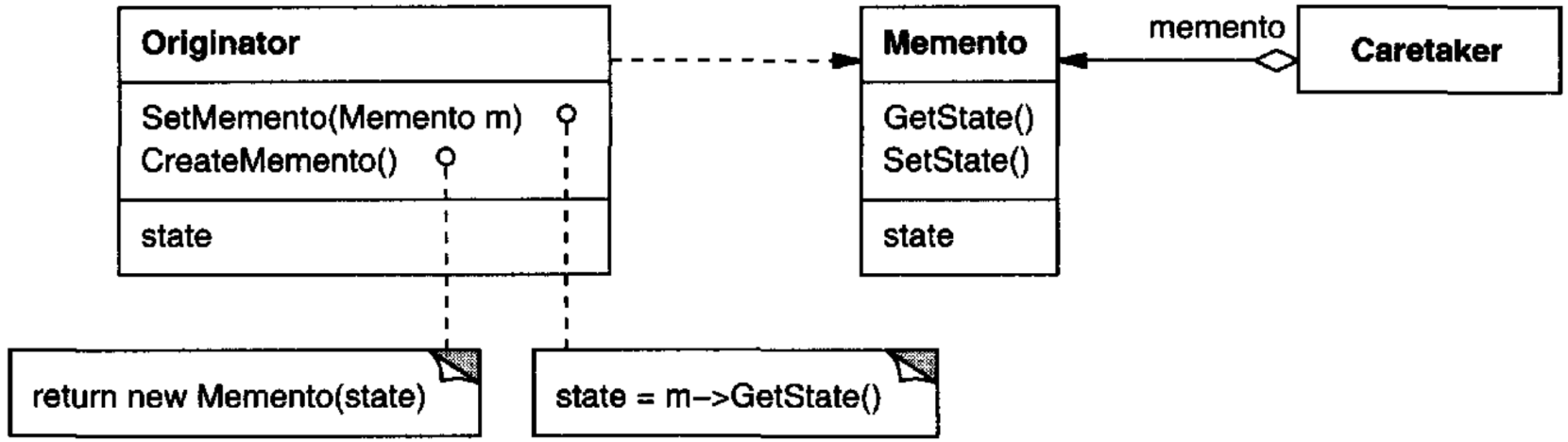
MEMENTO

→ COMPORTAMENTALE

→ BASATO SU OGGETTI

- Senza violare l'incapsulamento di un oggetto, cattura e esternalizza il suo stato interno, così da permettere all'oggetto di essere ripristinato al suo stato in un momento successivo.

Lo stato dell'oggetto che si vuole salvare (Originator) è immagazzinato in un oggetto (Memento) che è portato fuori e custodito esternamente da un altro oggetto (Caretaker). Quando si vuole restituire uno stato memorizzato dell'Originator, è invocato un suo metodo di ripristino che agisce su un Memento che contiene quel particolare stato. Inoltre, il Memento si occupa di proteggere l'accesso allo stato dell'Originator, che può avvenire solo da parte di quest'ultimo.



MEMENTO

- **Memento**
- memorizza lo stato interno dell'oggetto *Originator*
- protegge lo stato interno *dell'Originator* dall'accesso di oggetti diversi da esso. I *Memento* hanno effettivamente due interfacce. Il *CareTaker* ha un contatto stretto con il *Memento,* ne vede l'interfaccia 'stretta' e può solo passare il memento ad altri oggetti. *Originator*, al contrario, vede un'interfaccia 'ampia', che consente di accedere a tutti i dati necessari per ripristinarsi al suo stato precedente (idealmente, solo l'originator che ha prodotto il memento sarebbe autorizzato ad accedere allo stato interno del ricordo).

- **Originator**
- crea un memento contenente un'istantanea del suo stato interno corrente
- utilizza il memento per ripristinare il suo stato interno.

- **CareTaker**
- è responsabile della "custodia" del memento
- non utilizza mai o esamina il contenuto di un memento.

MEMENTO

VANTAGGI/SVANTAGGI

1. **Preservare i confini dell'incapsulamento**
   - ↳ Memento evita di esporre informazioni che dovrebbero essere gestite solo da un mittente ma che devono essere conservate al di fuori del mittente.
2. **Semplifica Originator**
   - ↳ In altri tipi di design che preservano l'incapsulamento, l'Originator mantiene le versioni dello stato interno richieste dai client, avendo quindi tutto l'onere della gestione dello storage.
3. **L'uso dei memento potrebbe essere costoso**
   - ↳ Il Memento potrebbe subire un notevole sovraccarico se Originator ha grandi quantità di informazioni da archiviare nel memento o se i client creano e restituiscono memento al creatore abbastanza spesso.
4. **Definizione di interfacce strette e ampie**
   - ↳ In alcuni linguaggi può essere difficile garantire che solo l'originator possa accedere allo stato memorizzato.
5. **Costi nascosti nella " cura " dei ricordi**
   - ↳ Un CareTaker è responsabile dell'eliminazione dei ricordi a cui tiene. Tuttavia, il CareTaker non ha idea di quanto stato ci sia nel ricordo, quindi potrebbe sostenere costi di archiviazione elevati.

MEMENTO

ESEMPIO

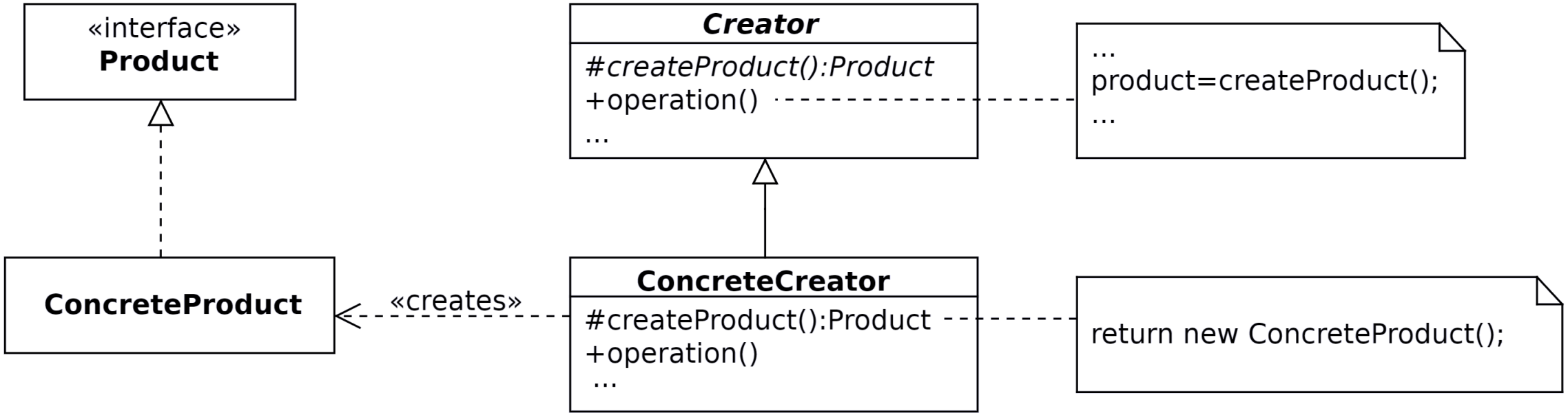
Accumulatore al cui interno si svolgono operazioni di:
- somma
- sottrazione
- moltiplicazione
- divisione

Inoltre, si vuole poter fare un UNDO di ogni operazione

Per esempio, se è stata fatta una somma si vuole tornare indietro con una sottrazione, ovvero con l'operazione opposta.

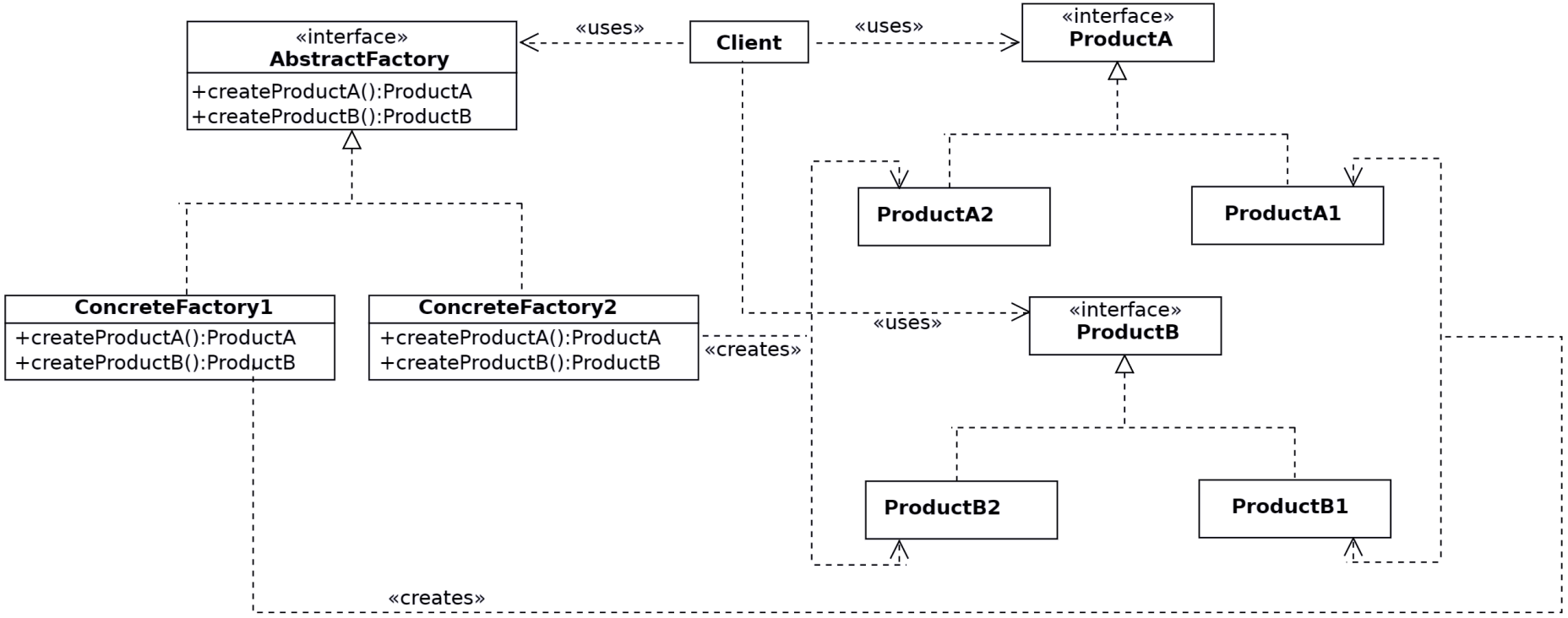
FACTORY METHOD

FACTORY METHOD
→ CREAZIONALE
→ BASATO SU CLASSI
- Definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione del tipo di classe da istanziare



ABSTRACT FACTORY

ABSTRACT FACTORY
→ CREAZIONALE
→ BASATO SU OGGETTI
- Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete



FACTORY METHOD vs ABSTRACT FACTORY

Factory Method	Abstract Factory
È un metodo.	È un'interfaccia per creare oggetti correlati. Abstract Factory è implementato spesso (non per forza) attraverso Factory Method.
Gli oggetti vengono creati attraverso l'ereditarietà	Gli oggetti vengono creati attraverso la composizione
Crea un solo tipo di oggetto	Crea famiglie di oggetti
Per creare nuovi oggetti dello stesso tipo è sufficiente creare una sottoclasse del Creator	Per creare nuovi oggetti della stessa famiglia è necessario ridefinire l'interfaccia di AbstractFactory Per creare nuove famiglie di oggetti è necessario creare una sottoclasse di AbstractFactory

STRATEGY

→ COMPORTAMENTALE
→ BASATO SU OGGETTI

- Definisce una famiglia di algoritmi incapsulati che possono essere scambiati per ottenere comportamenti specifici. Questo permette di modificare gli algoritmi in modo indipendente dai clienti che li usano.

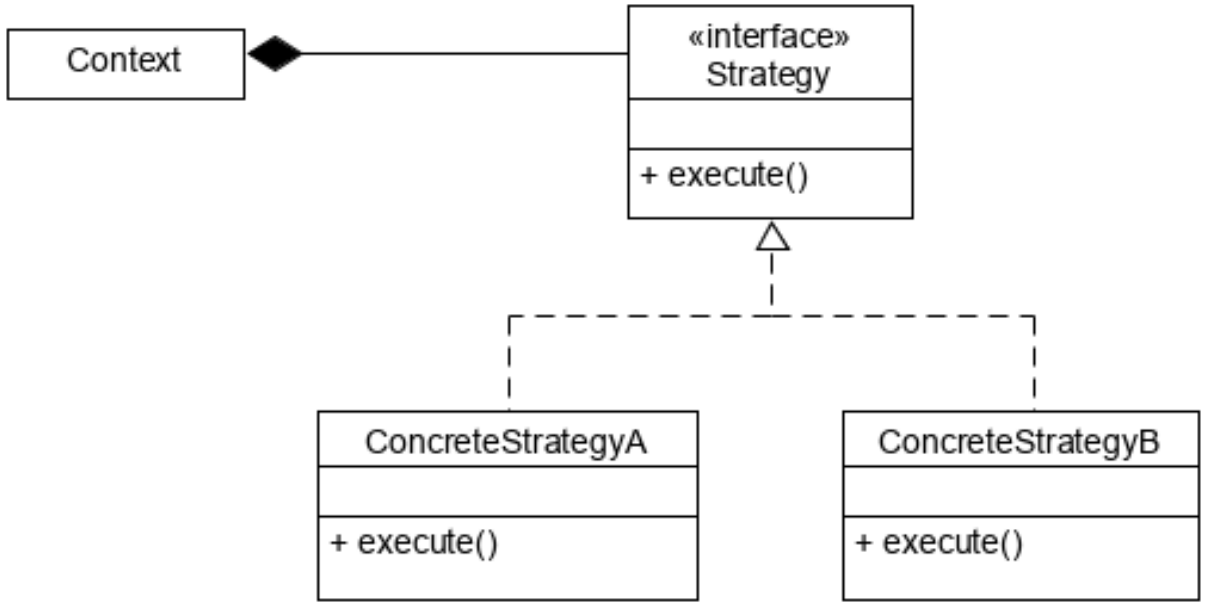
APPLICABILITÀ

Si usa il pattern strategy quando:

- molte classi correlate differiscono solo nel loro comportamento
- sono necessarie diverse varianti di un algoritmo (compromessi spazio/tempo)
- un algoritmo utilizza dati che i clienti non dovrebbero conoscere (ad es. evitare di esporre strutture di dati complesse)
- una classe definisce molti comportamenti e questi compaiono come più istruzioni condizionali.

STRATEGY

STRUTTURA E PARTECIPANTI



- **Strategy**

Dichiara una interfaccia comune per tutti gli algoritmi supportati. Il *Context* utilizza questa interfaccia per invocare gli algoritmi definiti in ogni *ConcreteStrategy.*

- **ConcreteStrategy**

Sono le implementazione degli algoritmi che usano l'interfaccia *Strategy.*

- **Context**

- È configurato con un oggetto *ConcreteStrategy* e mantiene un riferimento verso esso.
- Può specificare un'interfaccia che consenta alle Strategy accedere ai propri dati.

STRATEGY

VANTAGGI E SVANTAGGI

1. **Famiglie di algoritmi correlati**

   Le gerarchie di classi Strategy definiscono una famiglia di algoritmi o comportamenti da riutilizzare per i contesti.

2. **Alternativa all'uso di sottoclassi**.

   L'ereditarietà offre un modo alternativo per supportare una varietà di algoritmi o comportamenti, ma lo fa in modo statico, rendendo il contesto più difficile da comprendere, mantenere ed estendere. Incapsulare l'algoritmo in classi Strategy separate consente di variare l'algoritmo indipendentemente dal suo contesto, facilitando il passaggio, la comprensione e l'estensione.

3. **Le strategie eliminano le istruzioni condizionali**.

   Quando comportamenti diversi sono raggruppati in una classe, è necessario utilizzare istruzioni condizionali per selezionare il comportamento giusto. Con l'incapsulamento del comportamento nelle classi di strategie si può ovviare a ciò.

STRATEGY

4. Scelta di implementazioni

Le strategie possono fornire diverse implementazioni dello stesso comportamento. Il cliente può scegliere tra strategie con diversi compromessi di tempo e spazio.

5. I clienti devono essere consapevoli delle diverse Strategie

Il cliente deve capire in che modo le Strategie differiscono prima di poter selezionare quella appropriata. Quindi potrebbe trovarsi davanti a problemi di implementazione.

6. Overhead di comunicazione tra *Strategy* e *Context*

L'interfaccia Strategy è condivisa da tutte le classi ConcreteStrategy, quindi è probabile che alcune ConcreteStrategy non utilizzino tutti i parametri che ricevono dall'interfaccia (potrebbero anche non usarne proprio). Quindi ci potrebbero essere casi in cui il Context crea e inizializza parametri che non verranno mai usati.

7. Aumento del numero di oggetti

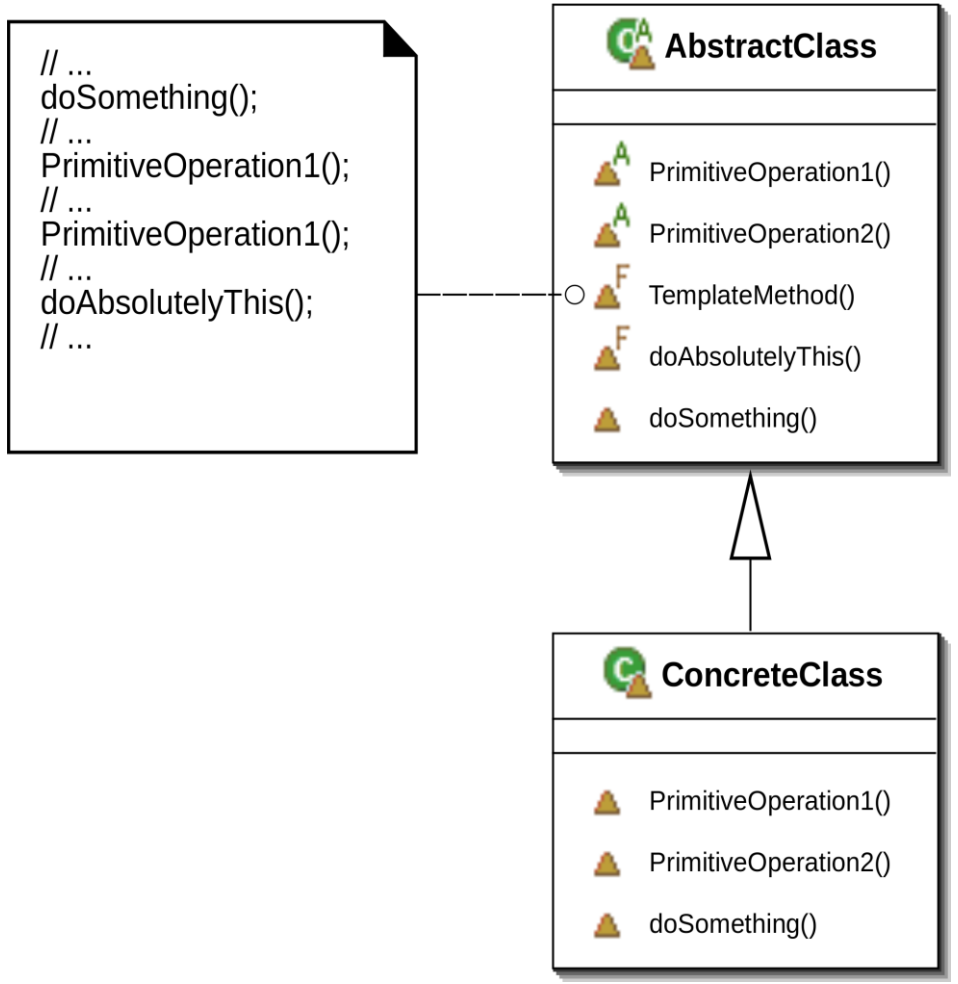
Le strategie aumentano il numero di oggetti in un'applicazione.

TEMPLATE METHOD

→ COMPORTAMENTALE
→ BASATO SU CLASSI

- definisce la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono.

Normalmente sono le sottoclassi a chiamare i metodi delle classi genitrici; in questo pattern è il *metodo template*, appartenente alla classe genitrice, a chiamare i metodi specifici ridefiniti nelle sottoclassi.



TEMPLATE METHOD

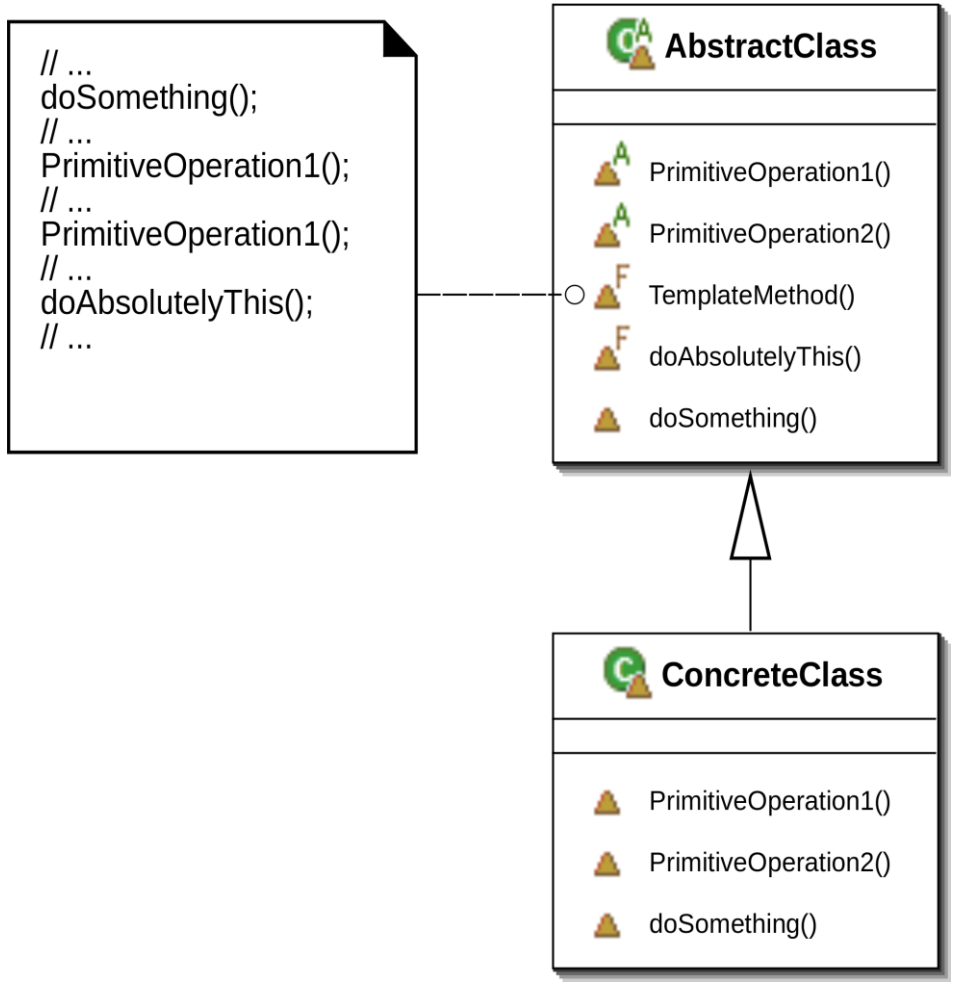
STRUTTURA

AbstractClass

Definisce le operazioni primitive astratte che le classi concrete sottostanti andranno a sovraccaricare e implementa il metodo template che rappresenta lo scheletro dell'algoritmo.

ConcreteClass

Deve implementare le operazioni primitive astratte che eredita dalla superclasse, specificando così il comportamento per i passi variabili dell'algoritmo. Può scegliere di sovrascrivere o meno l'implementazione dei metodi hook. Per tutto il resto del codice si affida all'implementazione contenuta nella AbstractClass.



TEMPLATE METHOD

Il **metodo template** di *AbstractClass* può contenere del codice e anche delle chiamate ad altri metodi:

- *operazioni primitive*, scritte in modo da dover essere ridefinite nelle sottoclassi. Rappresentano la parte variabile e personalizzabile dell'algoritmo. In Java sono metodi astratti (dichiarati con la parola chiave abstract). Nel diagramma sono **PrimitiveOperation1()** e **PrimitiveOperation2()**.
- *metodi hook*, ovvero metodi che possono essere ridefiniti a piacere oppure ereditati e utilizzati così come sono, a discrezione della sottoclasse. Rappresentano passi opzionali dell'algoritmo oppure implementazioni basilari che se necessario possono essere ridefinite nelle sottoclassi. Nel diagramma è il metodo **doSomething()** che la classe concreta sceglie di ridefinire.
- *metodi non sovrascrivibili* che vincolano le implementazioni delle sottoclassi ad utilizzare l'implementazione definita dalla classe astratta. Nel diagramma il metodo **doAbsolutelyThis()** è dichiarato con la parola chiave final del linguaggio Java e quindi è protetto dal meccanismo dell'overriding.

STRATEGY vs TEMPLATE METHOD

Strategy	**Template method**
Encapsulate algorithm → by composition	Encapsulate algorithm → by inheritance
More flexible (thanks to the use of different encapsulated strategies)	More efficient (thanks to shared code in super class)
“Strategy says , I don’t depend on anyone , I can do the entire algorithm my self”	“Template says, I do not keep any duplicate code so I use important technique code reuse and much efficient.”
client can change their algorithm at runtime simply by using different strategy object.	Abstract methods are implemented by sub-classes. To prevent the subclass from changing the algorithm the Template Method has to be declared *final*