

Studio di casi basati su design pattern



*Prof. Paolo Ciancarini
Corso di Ingegneria del Software
CdL Informatica
Università di Bologna*

Obiettivi della lezione

- Alcuni casi di studio
 - Editor grafico
 - Videogioco
 - Sistema legacy
- La teoria dei design pattern
- I 23 pattern della “Gang of Four” (GoF)
- Relazioni tra i design pattern

Caso di studio 1: Editor grafico

- Progettare un editor di documenti
- Useremo alcuni pattern importanti
- Esempio tratto dal libro di Gamma e altri: vedi Riferimenti alla fine di questa presentazione

Caso di studio: Requisiti dell'editor

- Editor grafico di documenti
- Un documento è una sequenza di pagine contenenti testo e grafica
- Interazione basata su pulsanti, menu, scrollbar
- Finestre multiple sullo stesso documento
- Operazioni sui testi: sillabazione, controllo vocaboli
- Multipiattaforma

Editor: interfaccia

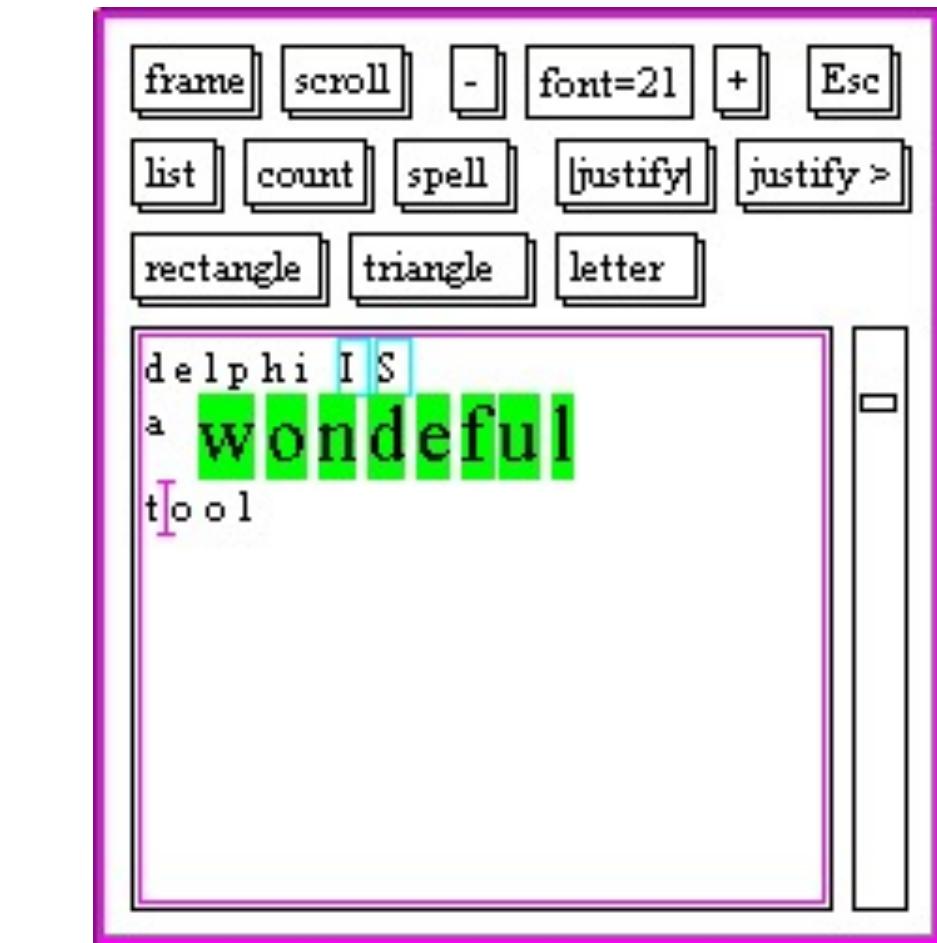
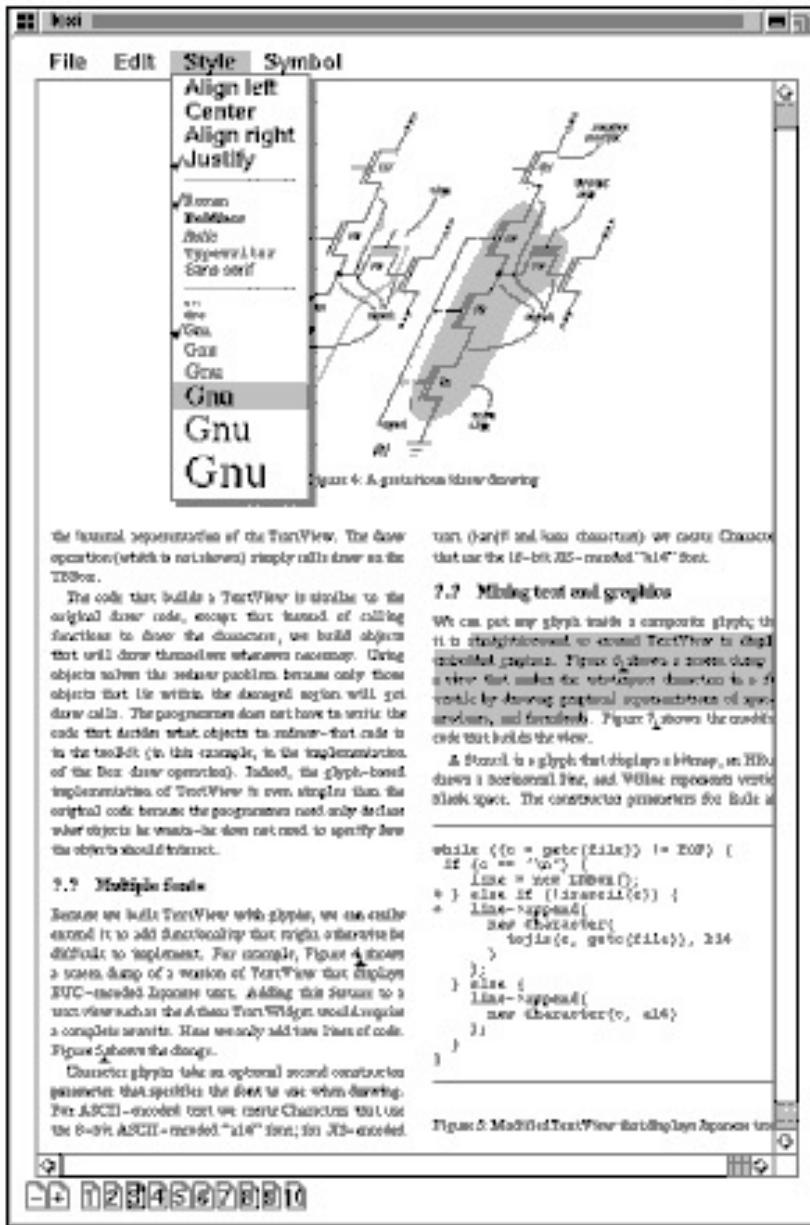
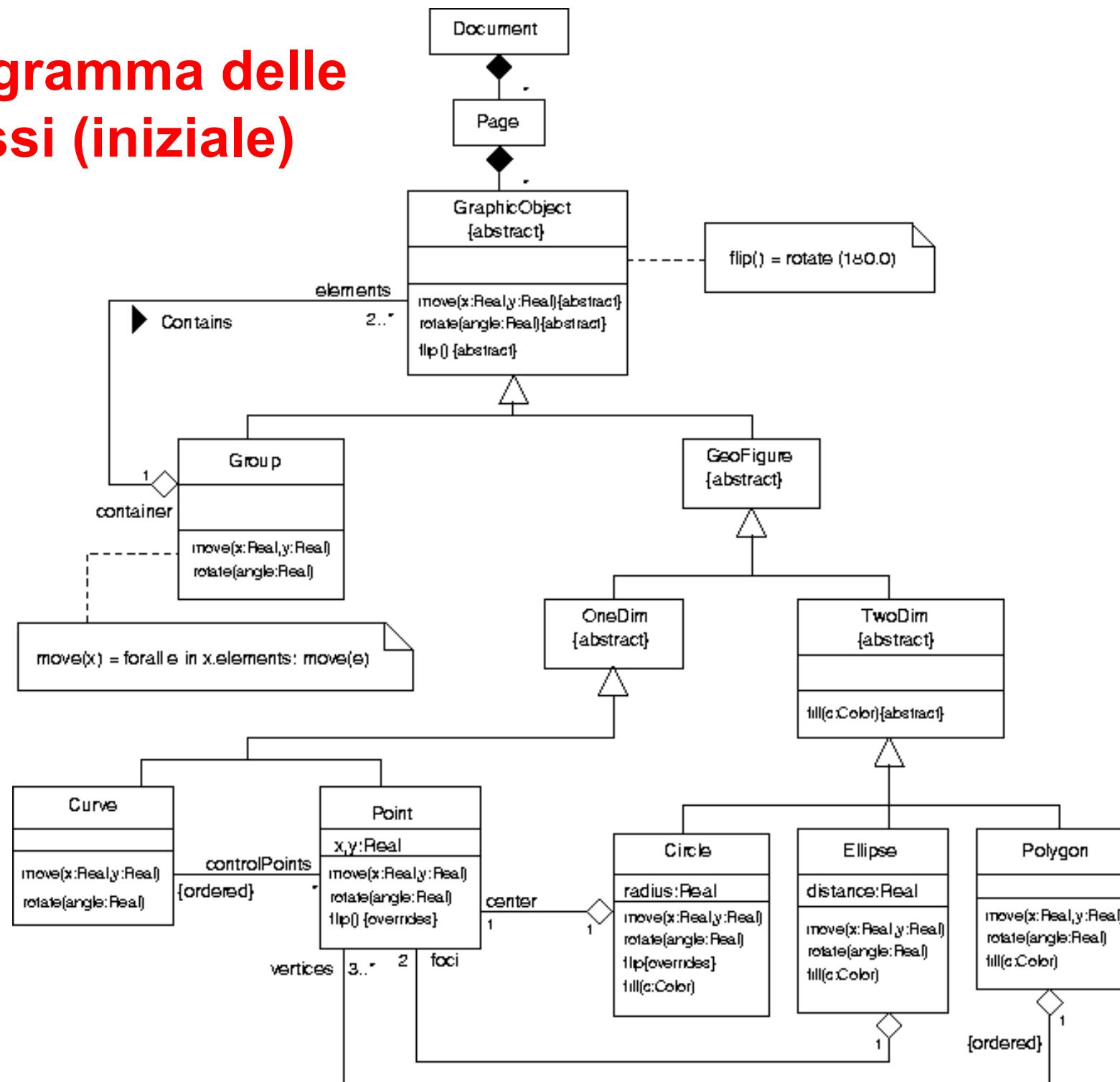
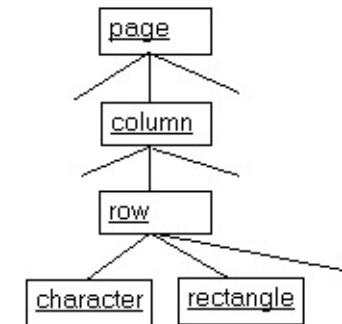
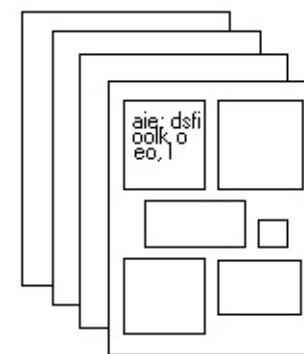


Diagramma delle classi (iniziale)

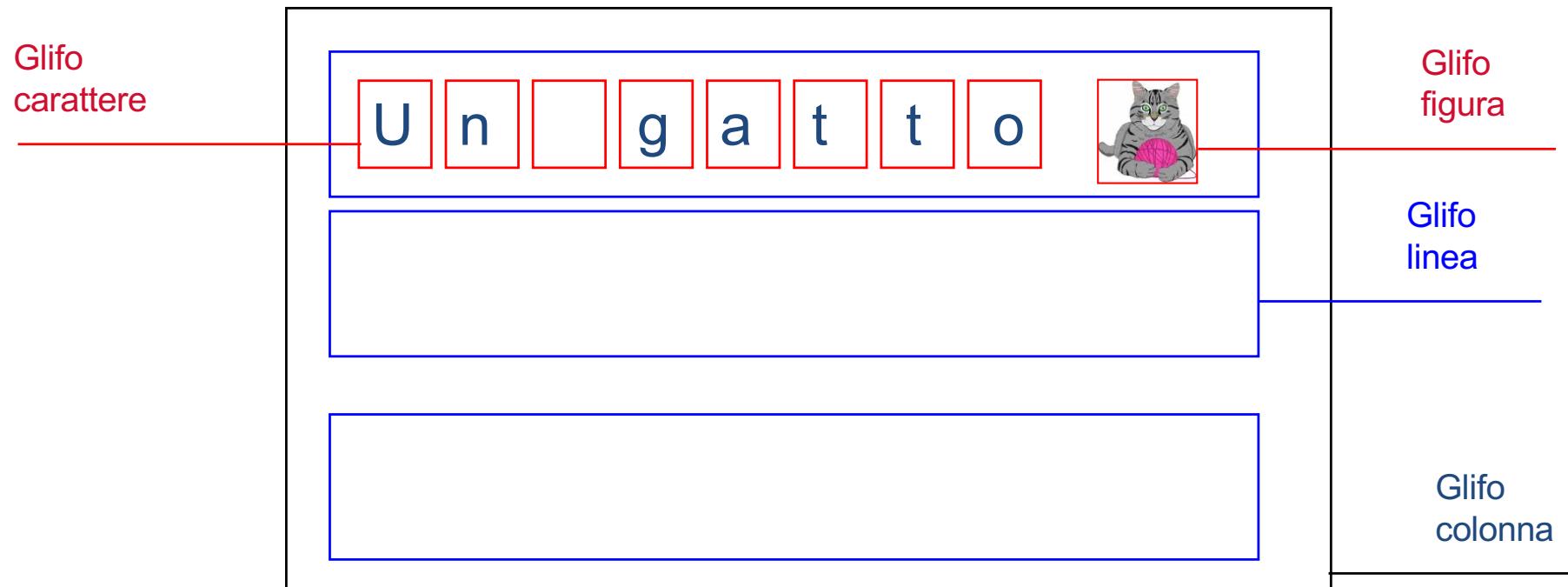


Struttura di un documento

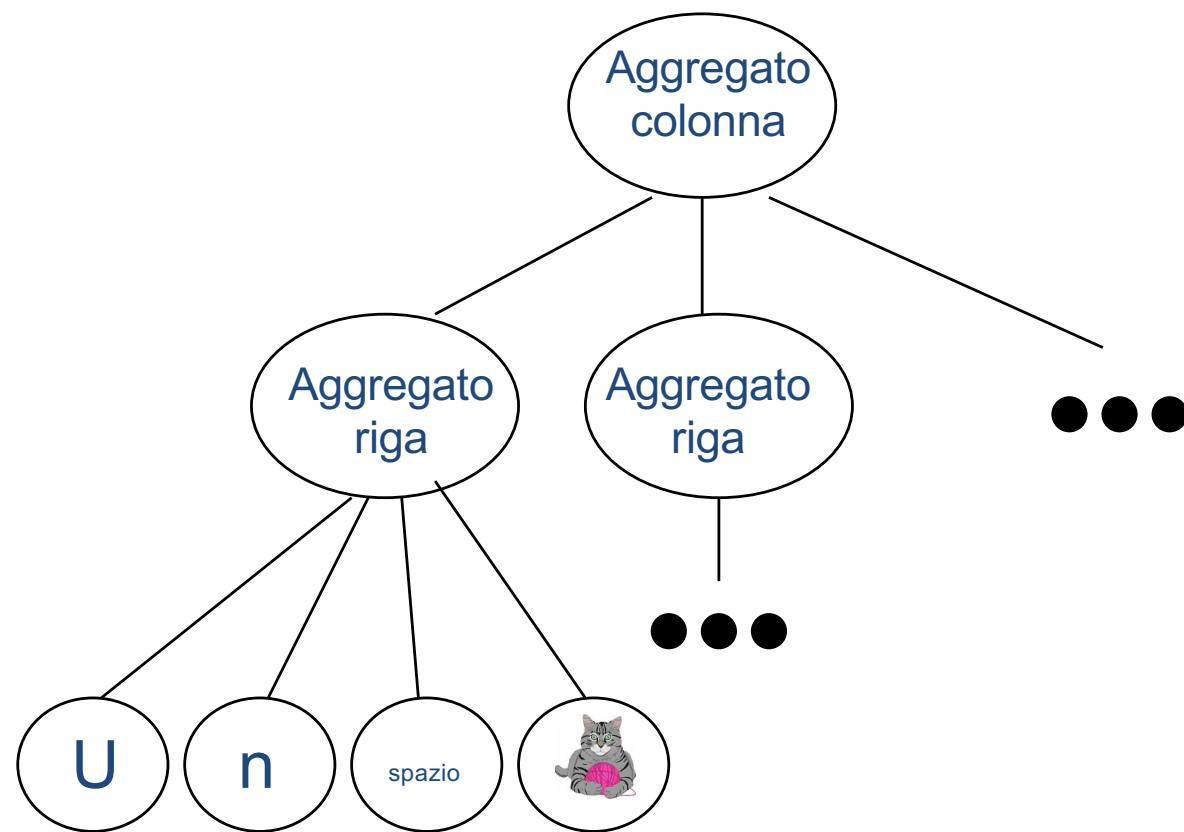
- Un documento ha struttura logica
 - Documento: sequenza di pagine
 - Pagina: sequenza di colonne
 - Colonna: sequenza di linee
 - Linea: sequenza di glifi
 - Glifo: elemento primitivo: carattere, figura, rettangolo, cerchio, ecc.



Composizione gerarchica



Struttura logica degli aggregati



Alternative progettuali

1. Classi diverse per gli elementi primitivi: carattere, linea, colonna, pagina; elementi grafici quali quadrato, cerchio, ecc.
2. Una sola classe astratta per la nozione di glifo, che rappresenta un elemento grafico generico con *unica interfaccia* e che potremo specializzare in tanti modi diversi

Codice Java

```
Class Glyph{
    List children = new LinkedList();
    Int ox,oy,width,height;

    Void draw(){
        For (g:children) g.draw();
    }

    Void insert(Glyph g){
        children.add(g);
    }

    Boolean intersects(int x, int y){
        Return (x>= ox) && (x<ox+width)
        && (y>= oy) && (y<oy+height);
    }
}
```

```
class Character extends Glyph {
    char c;

    public Character(char c) {
        this.c = c;
    }

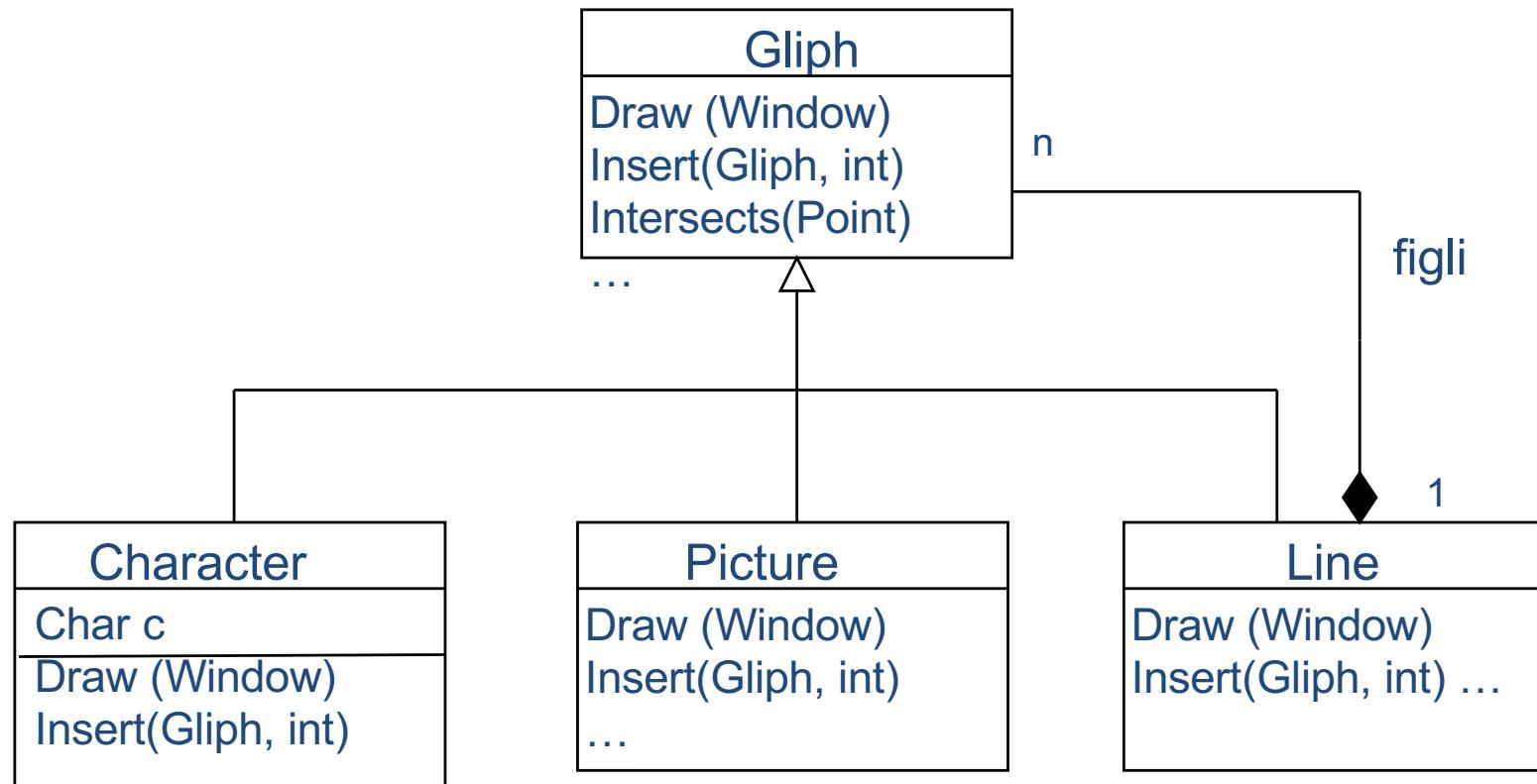
    void draw() {
        ...
    }
}
```

Codice Java

```
class Picture extends Glyph
{
    File pictureFile;
    public Picture(File
        pictureFile) {
        this.pictureFile =
            pictureFile;
    }
    void draw() {
        // draw picture
    }
}
```

```
class Line extends
    Glyph {
    char c;
    public Line() {}
    // inherits draw, ...
}
```

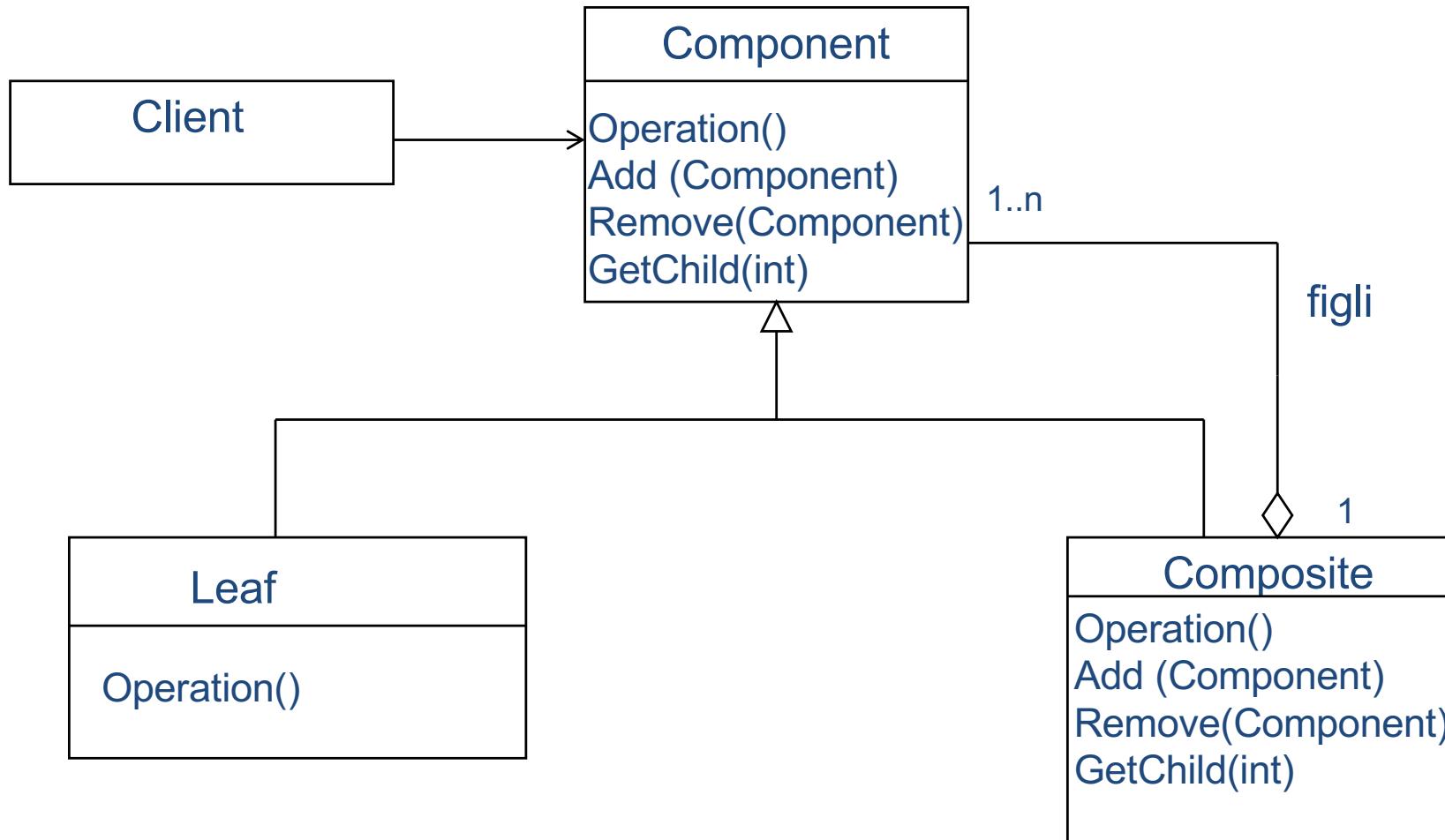
Diagramma delle classi



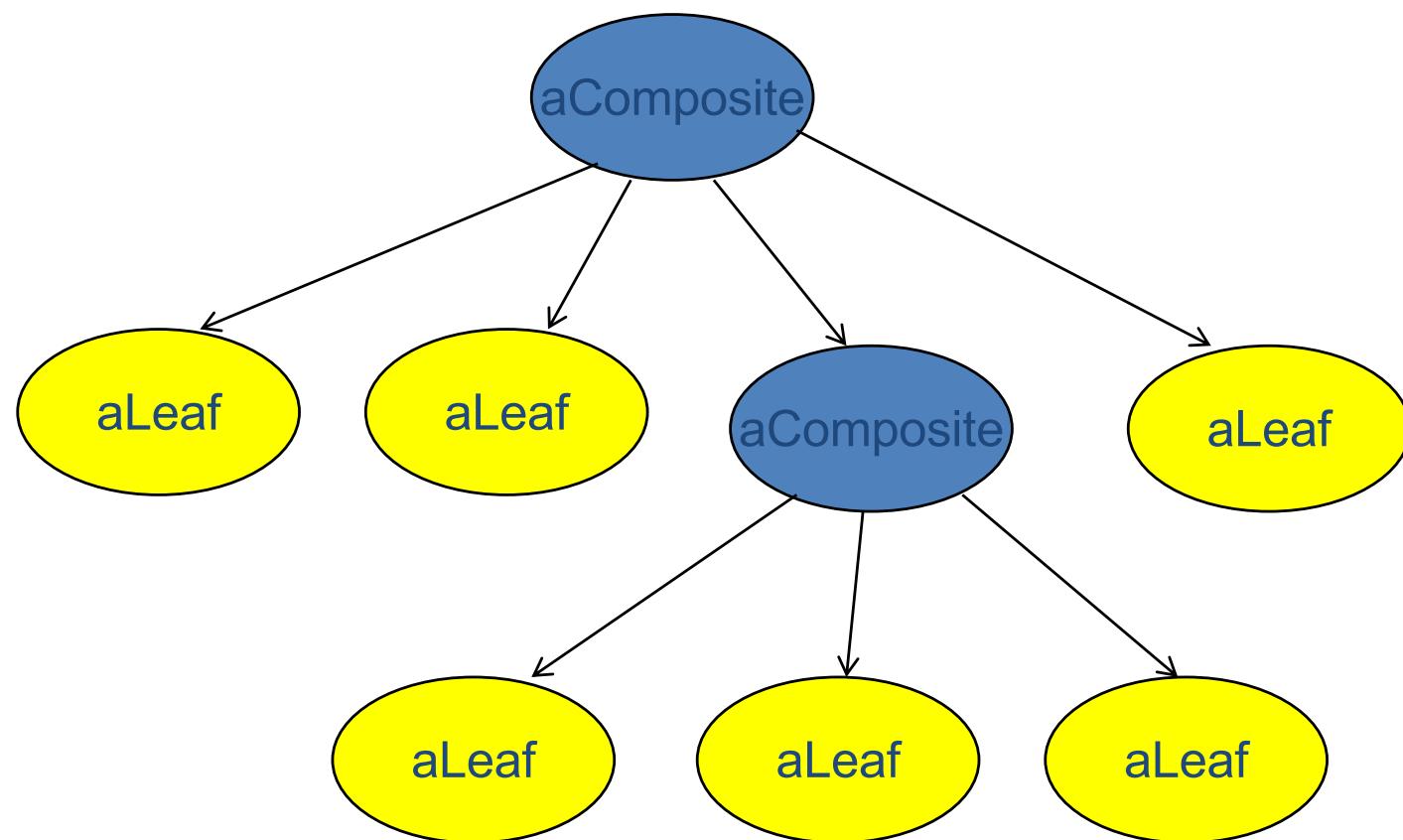
Pattern Composite

- Il diagramma segue il pattern “**Composite**”
 - Anche noto come “struttura ad albero”, “composizione ricorsiva”, “struttura induttiva”, ecc.
- Si applica a qualsiasi struttura gerarchica
 - Foglie e nodi hanno la stessa funzionalità
 - Implementa la stessa interfaccia per tutti gli elementi contenuti

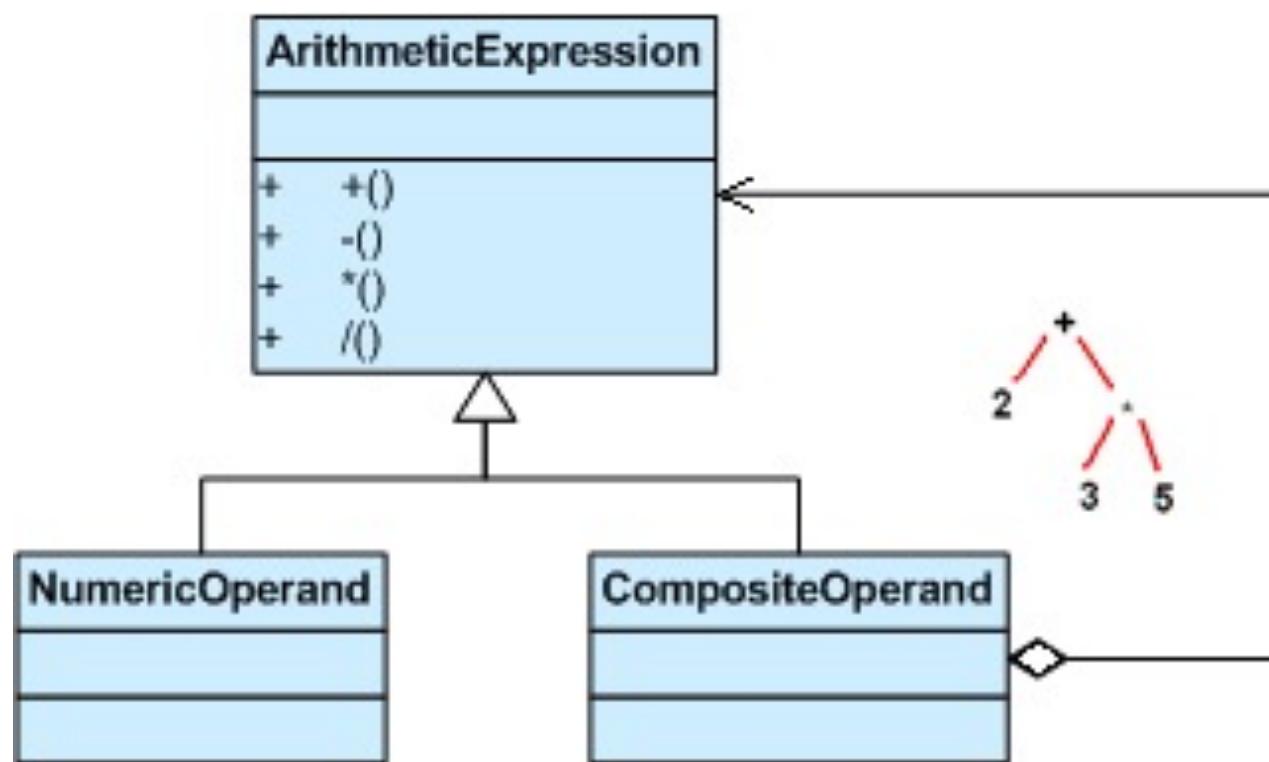
Pattern Composite: struttura



Oggetti “Composite”

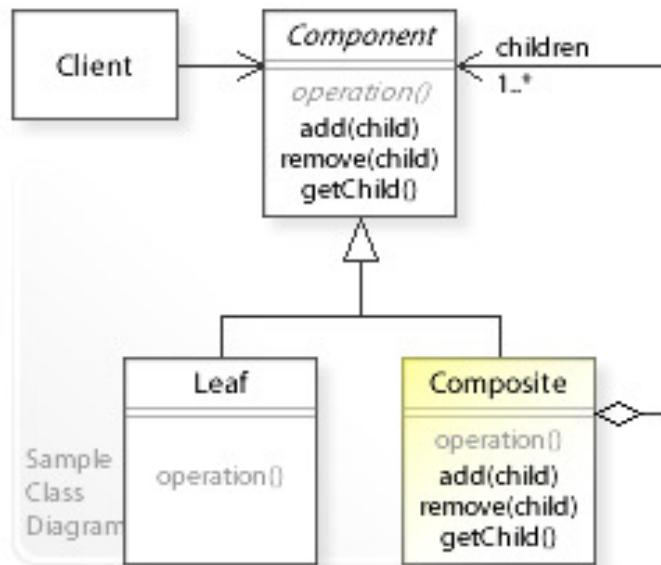


Pattern Composite: Esempio

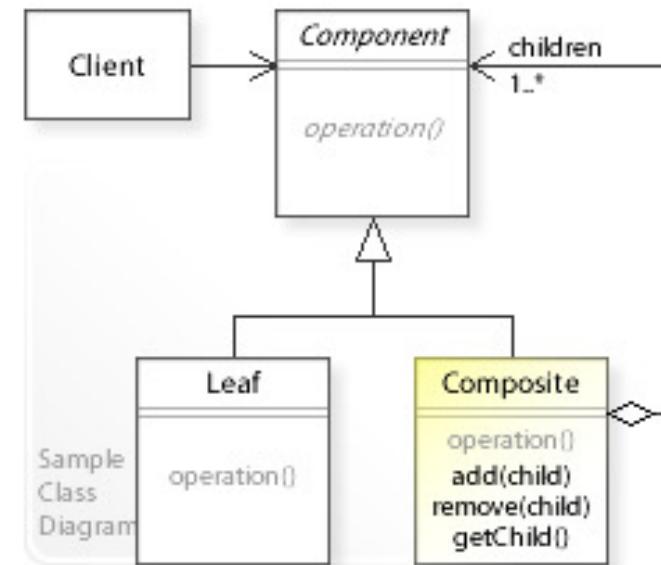


Composite e la Legge di Demetra

Design for Uniformity



Design for Type Safety

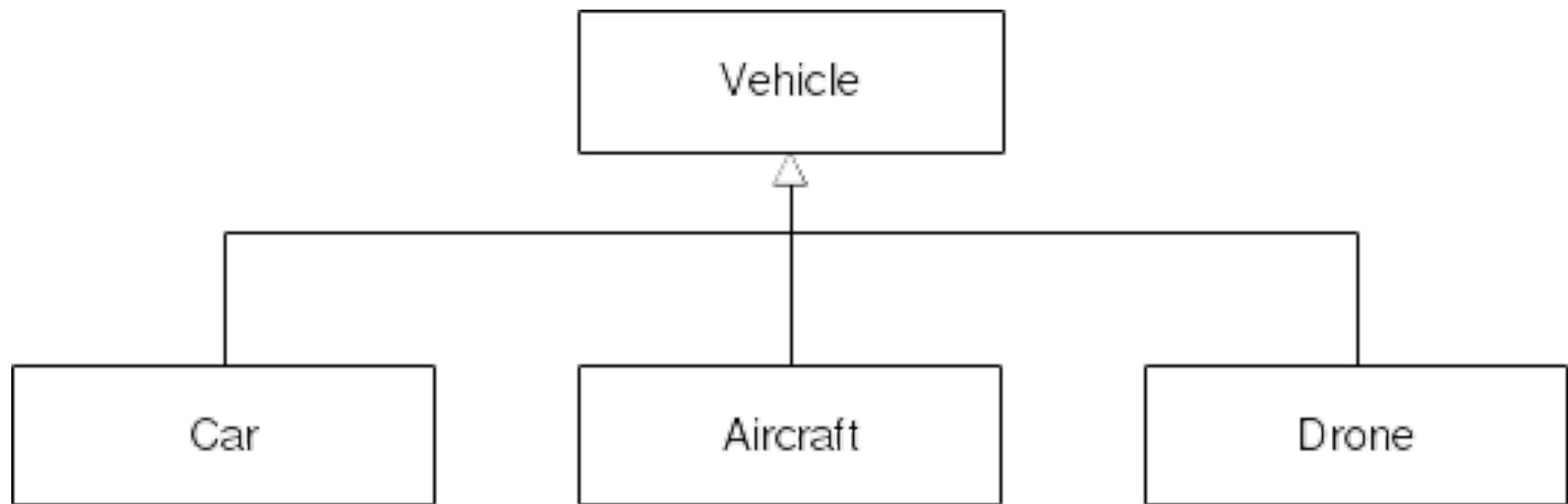


Viola Legge di Demetra!

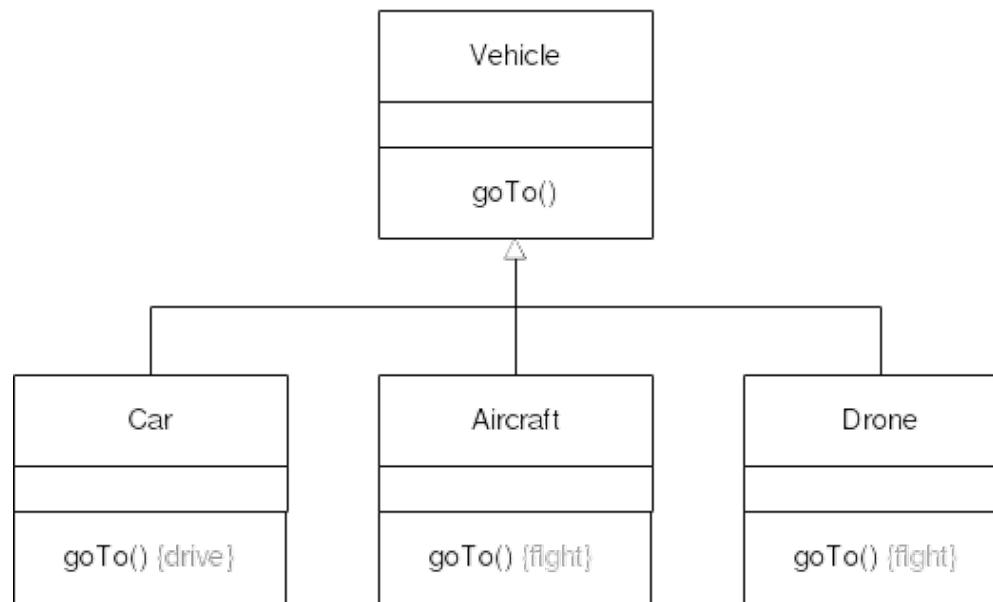
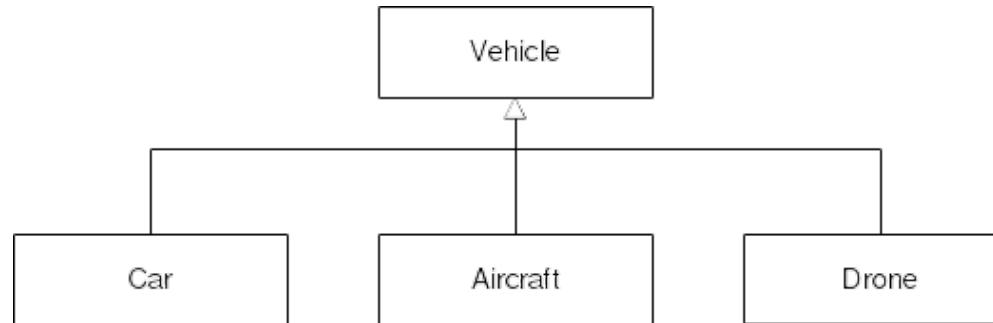
Composition over inheritance

- La composizione (e relativa delegation) è uno dei meccanismi ricorrenti che ritroviamo in molti pattern
- I due metodi più comuni per riutilizzare delle funzionalità nei sistemi object oriented sono l'ereditarietà e la composizione
- Approccio white box (composizione) vs black box (ereditarietà)
- Preferire la composizione rispetto all'ereditarietà aiuta a mantenere le classi encapsulate e focalizzate su di un unico aspetto
- La delegazione permette di rendere la composizione tanto potente quanto l'ereditarietà ("ha un" al posto di "è un")

Qual è il problema con l'ereditarietà



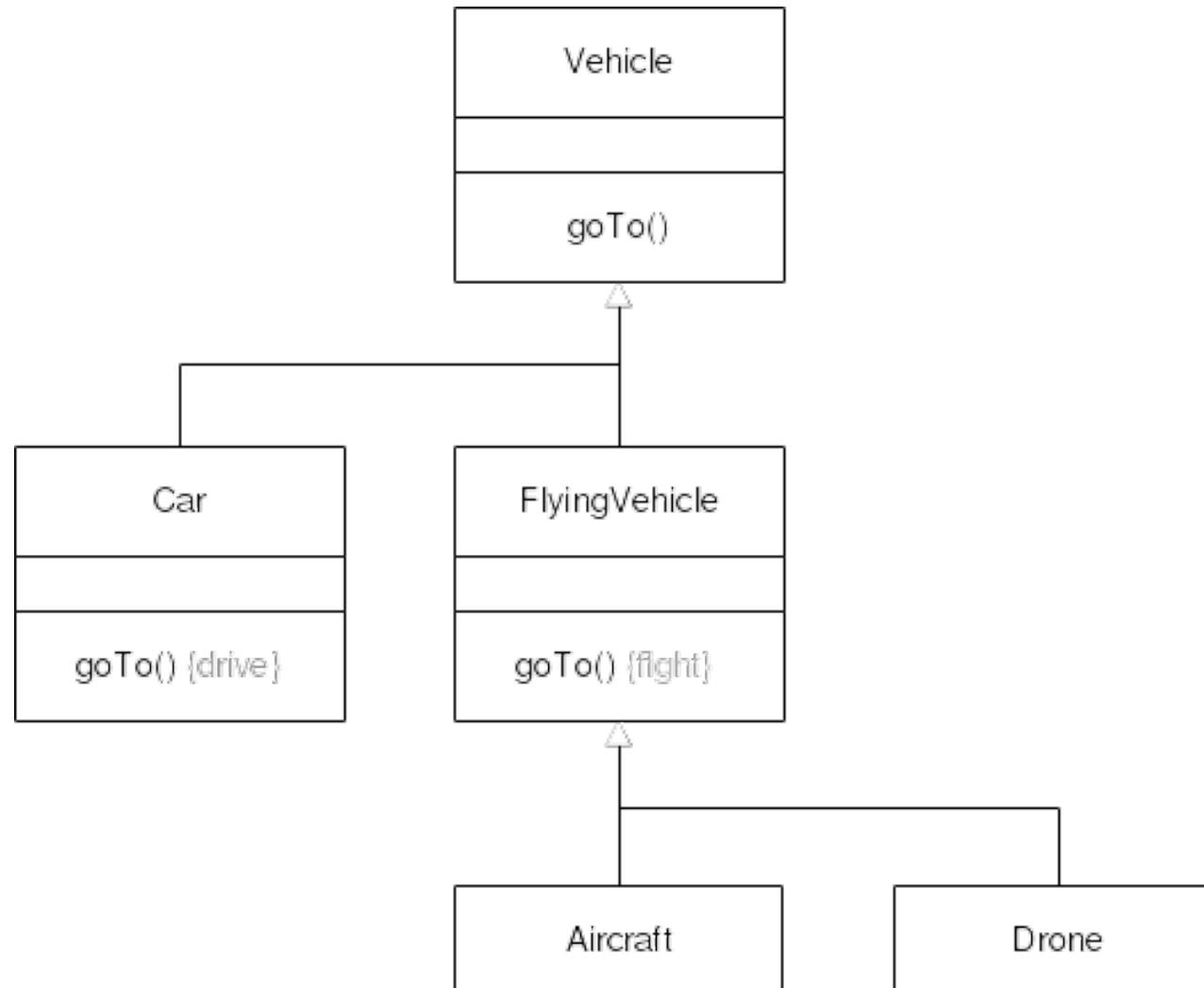
Qual è il problema con l'ereditarietà



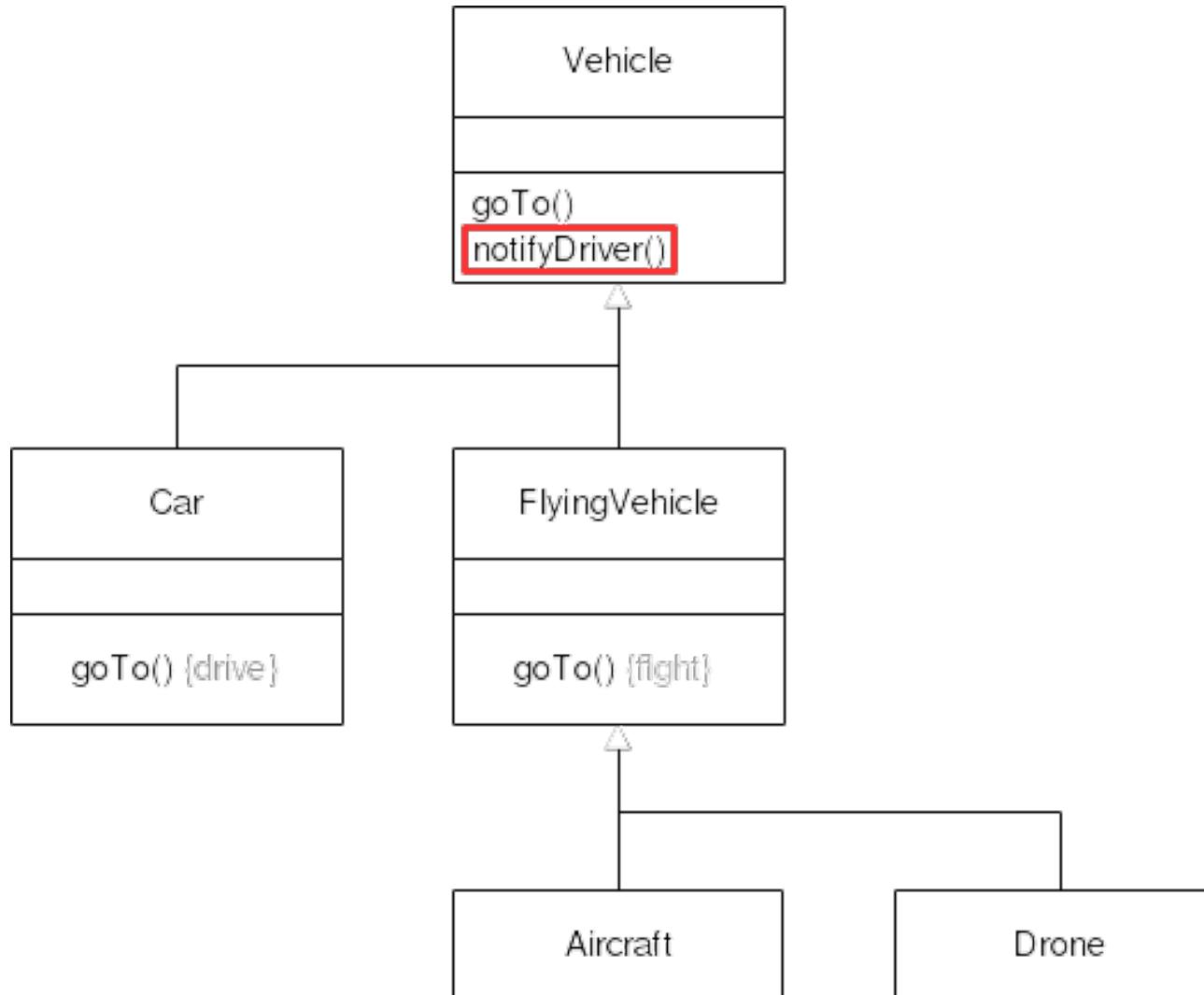
Qual è il problema con l'ereditarietà

- goTo in Aircraft e in Drone esprimono lo stesso comportamento
- Si tratta di un caso di "needless repetition" (un tipico *design smell*)
- Il copia incolla non dovrebbe essere usato mai quando si scrive del codice
- Quando lo stesso codice appare più volte, anche in forme leggermente diverse, lo sviluppatore non si è accorto di una astrazione

Qual è il problema con l'ereditarietà



Qual è il problema con l'ereditarietà



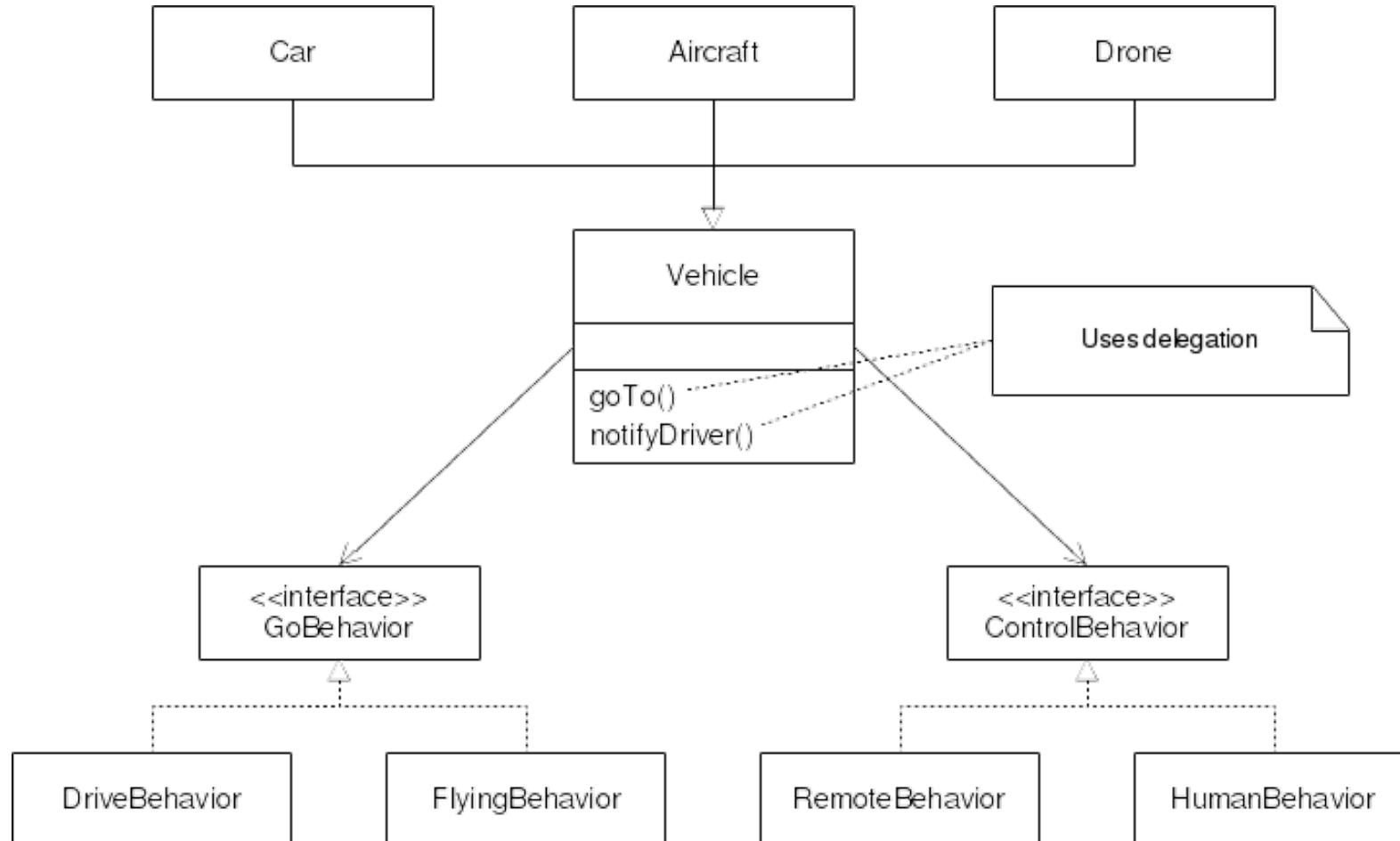
Qual è il problema con l'ereditarietà

- Voglio aggiungere una funzionalità per permettere di spedire dei messaggi che il veicolo dovrà mostrare al guidatore/pilota
- Questa funzionalità dovrebbe essere la stessa nel caso di Car e Aircraft ma non nel caso di Drone (visto che il pilota non è nel veicolo)
- Dovrei creare una gerarchia di ereditarietà che "litiga" con quella che ho creato per gestire goTo.

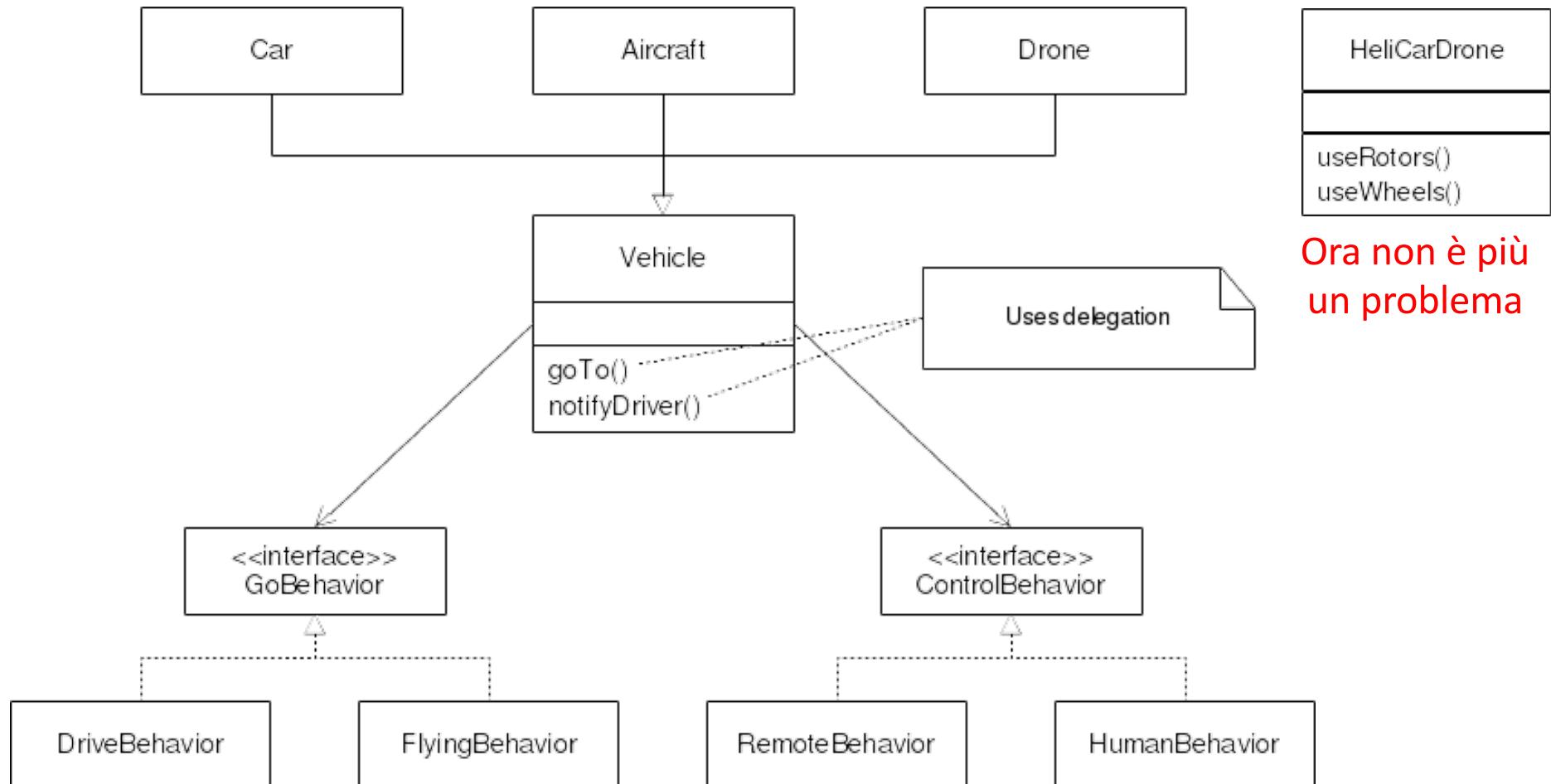
Qual è il problema con l'ereditarietà

- Cosa c'è di sbagliato nell'uso dell'ereditarietà in questo caso?
 - Non stiamo rispettando OCP
 - Non stiamo rispettando la Legge di Demetra (PV)
- Soluzione
 - Le superclassi non sono le sole astrazioni possibili
 - Usare la composizione: più flessibile (e si può cambiare in esecuzione)
 - Questa maggiore flessibilità è alla base della *delegation*

Qual è il problema con l'ereditarietà



Qual è il problema con l'ereditarietà



Problema: aggiungere elementi grafici

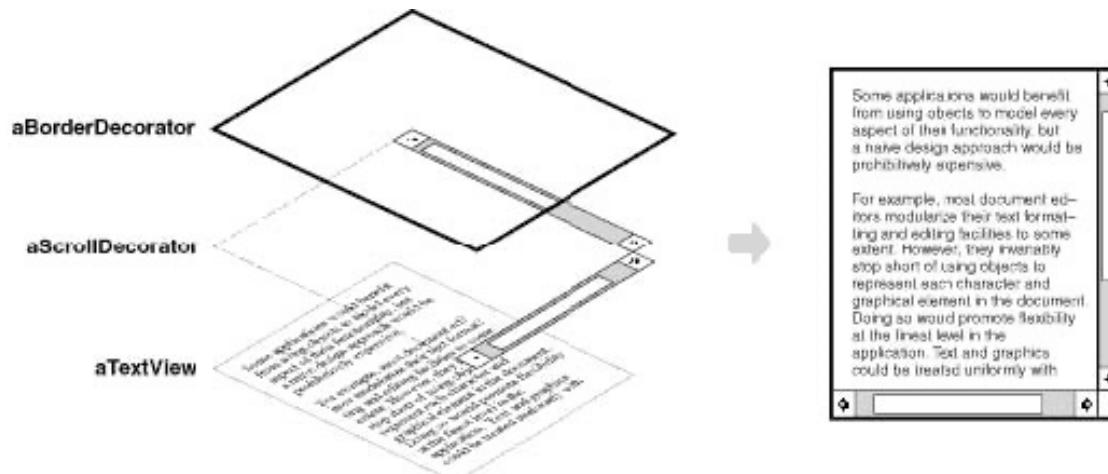
- Vogliamo decorare alcuni elementi dell'interfaccia utente
 - Bordi, Scrollbar, Bottoni, ecc.
- Come si incorporano nella struttura dell'editor?

Soluzione?

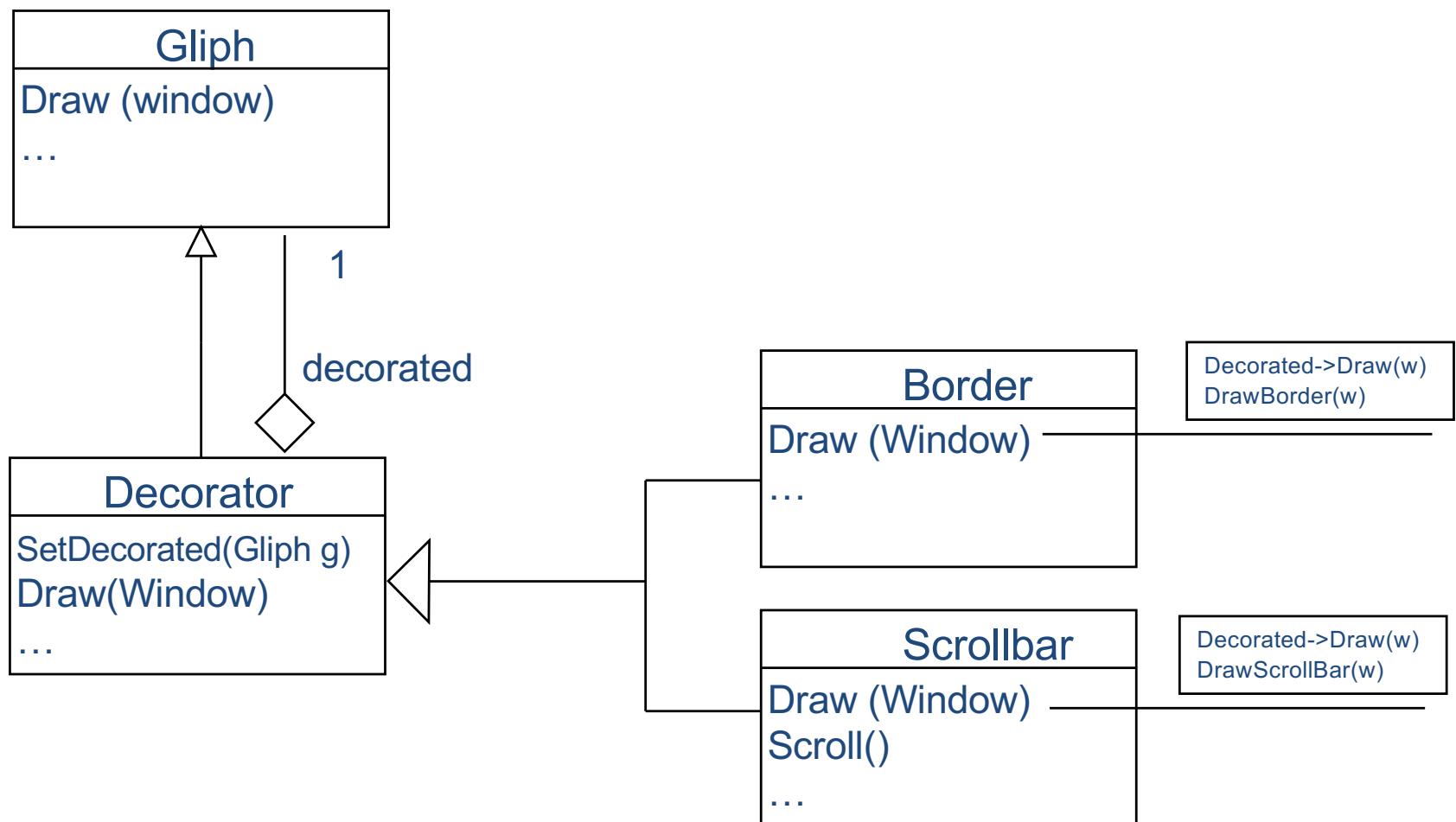
- Usiamo l'ereditarietà per estendere il comportamento
 - Sottoclassi di Gliph: Gliph con bordo, Gliph con scroll, Gliph bottone, Gliph con bordo e scroll, ecc.
 - Con n decoratori avremo 2^n combinazioni
- Esplosione di classi!
(la gerarchia di ereditarietà è statica)

Altra soluzione

- Vogliamo che le decorazioni (bordi, scrollbar, bottoni, menù, ...) possano essere combinate indipendentemente e a piacere
 - `X = new Scrollbar(new Border(new Character))`
- Definiamo una classe astratta Decorator
 - che estende Gliph
 - ha un membro decorato di tipo Gliph
 - Border, Scrollbar, Button, Menu estendono Gliph



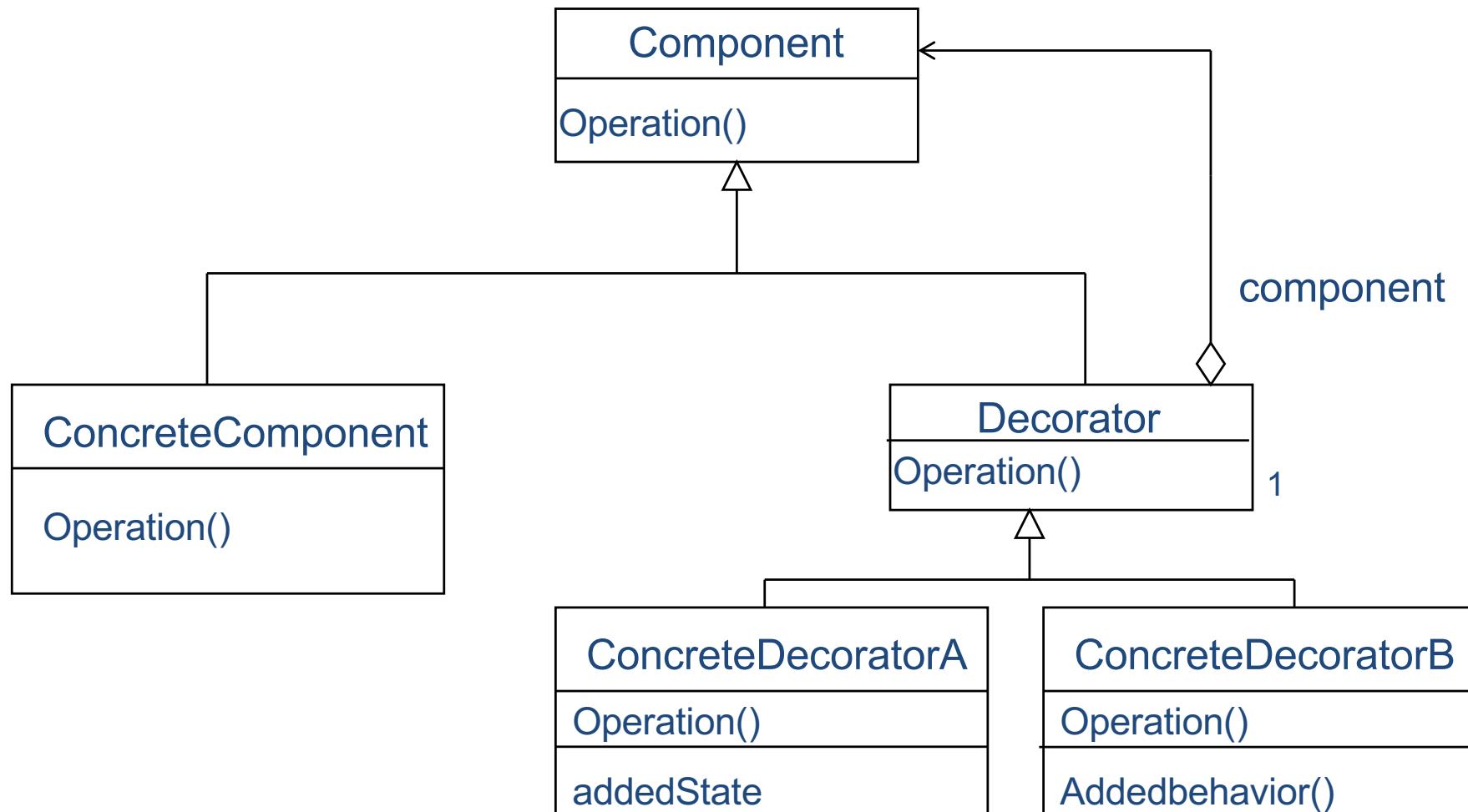
Diagramma



Pattern Decorator

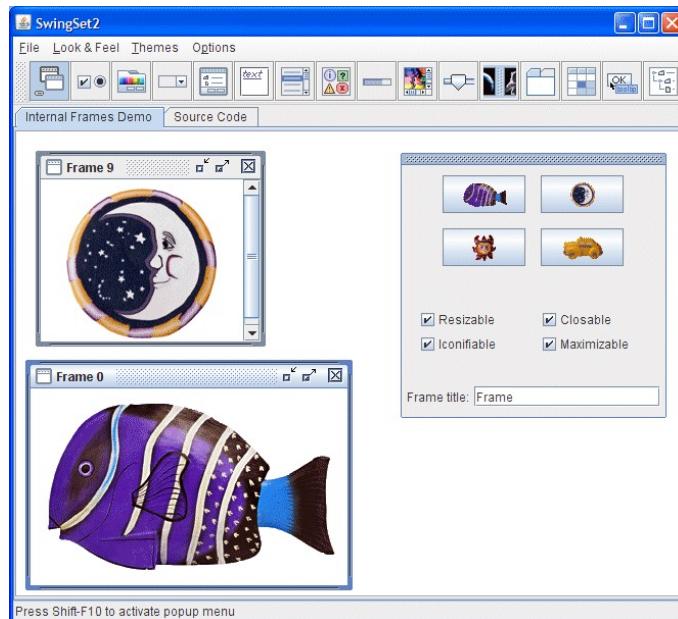
- Questo è il pattern “**Decorator**”
- **Intento:** aggiunge dinamicamente responsabilità ad un oggetto
- Le decorazioni si possono scegliere a run-time
- Di solito estende una struttura “**Composite**”

Pattern Decorator: struttura



Problema: diversi standard di interfaccia GUI

- Esistono diversi standard “look-and-feel” per le interfacce grafiche (GUI: Graphic User Interface)
- Differiscono per come visualizzano menù, bottoni, scrollbar, ecc.
- L'editor li deve supportare tutti



Platform	Look and Feel
Solaris, Linux with GTK+ 2.2 or later	GTK+
Other Solaris, Linux	Motif
IBM UNIX	IBM*
HP UX	HP*
Classic Windows	Windows
Windows XP	Windows XP
Windows Vista	Windows Vista
Macintosh	Macintosh*

Soluzioni?

```
Menu m = new  
MacosMenu;
```

oppure

```
Menu m = new  
WindowsMenu;
```

```
Menu m;  
if (style == Macos  
then m = new MacosMenu  
else if (style ==...)  
then ...
```

Creazione di un oggetto astratto

- Incapsulare la parte variabile di una classe
- Nel nostro caso cambia la creazione degli oggetti in funzione dell'interfaccia corrente
- Definiamo una classe GUIFactory
 - Inseriamo nella classe un metodo per ciascun oggetto dipendente da GUI
 - Avremo un oggetto GUIFactory per ciascun tipo di GUI

Diagramma: abstract factory

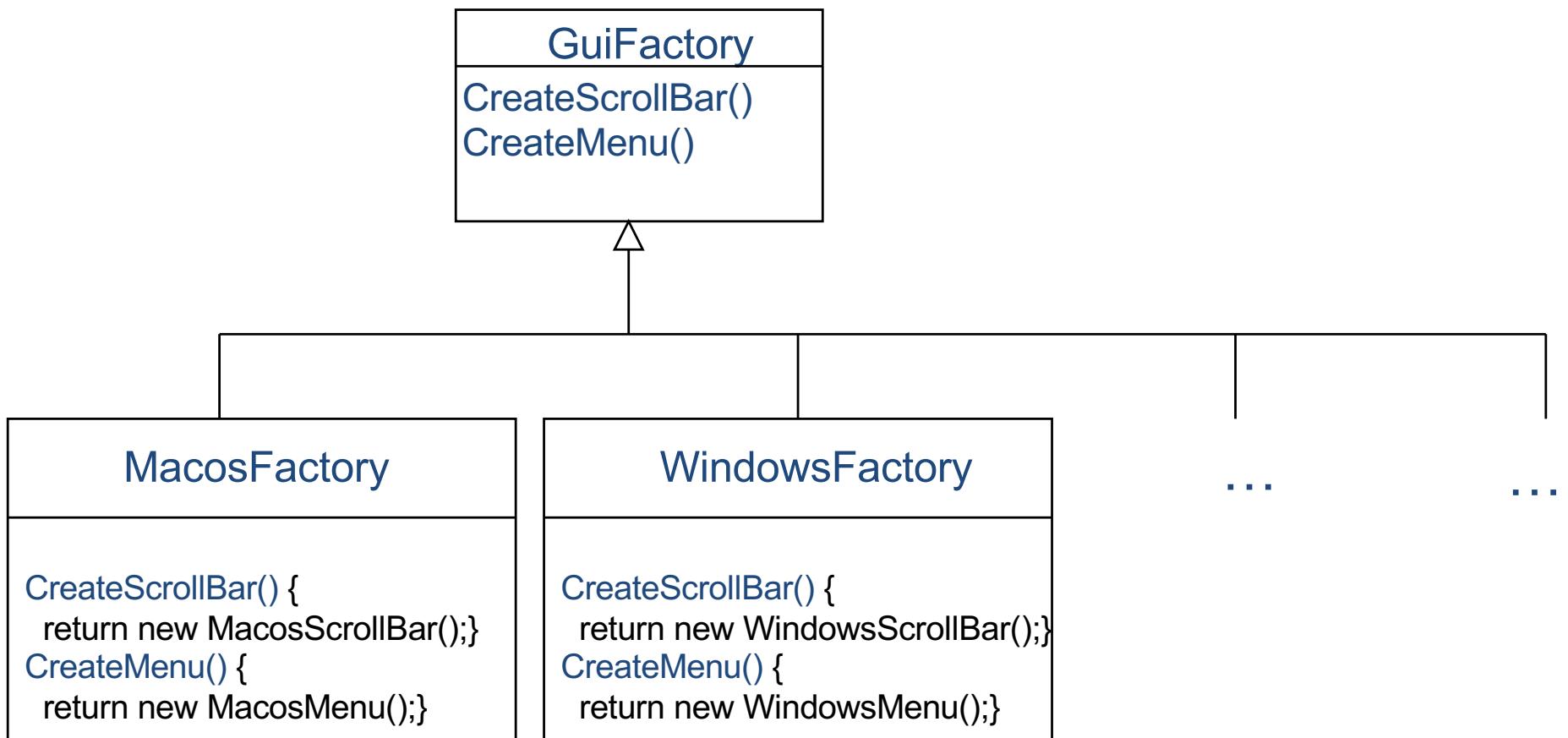
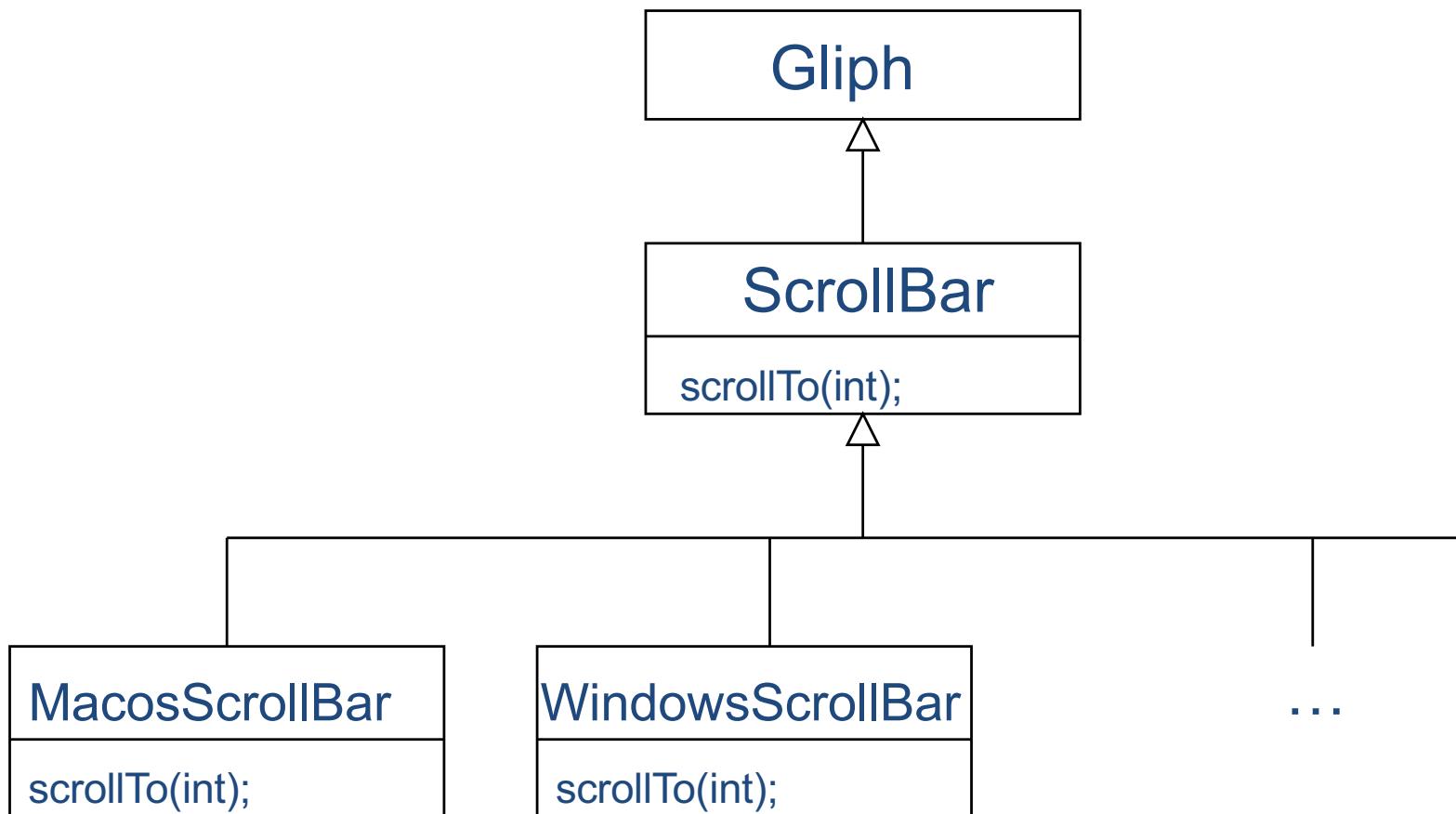


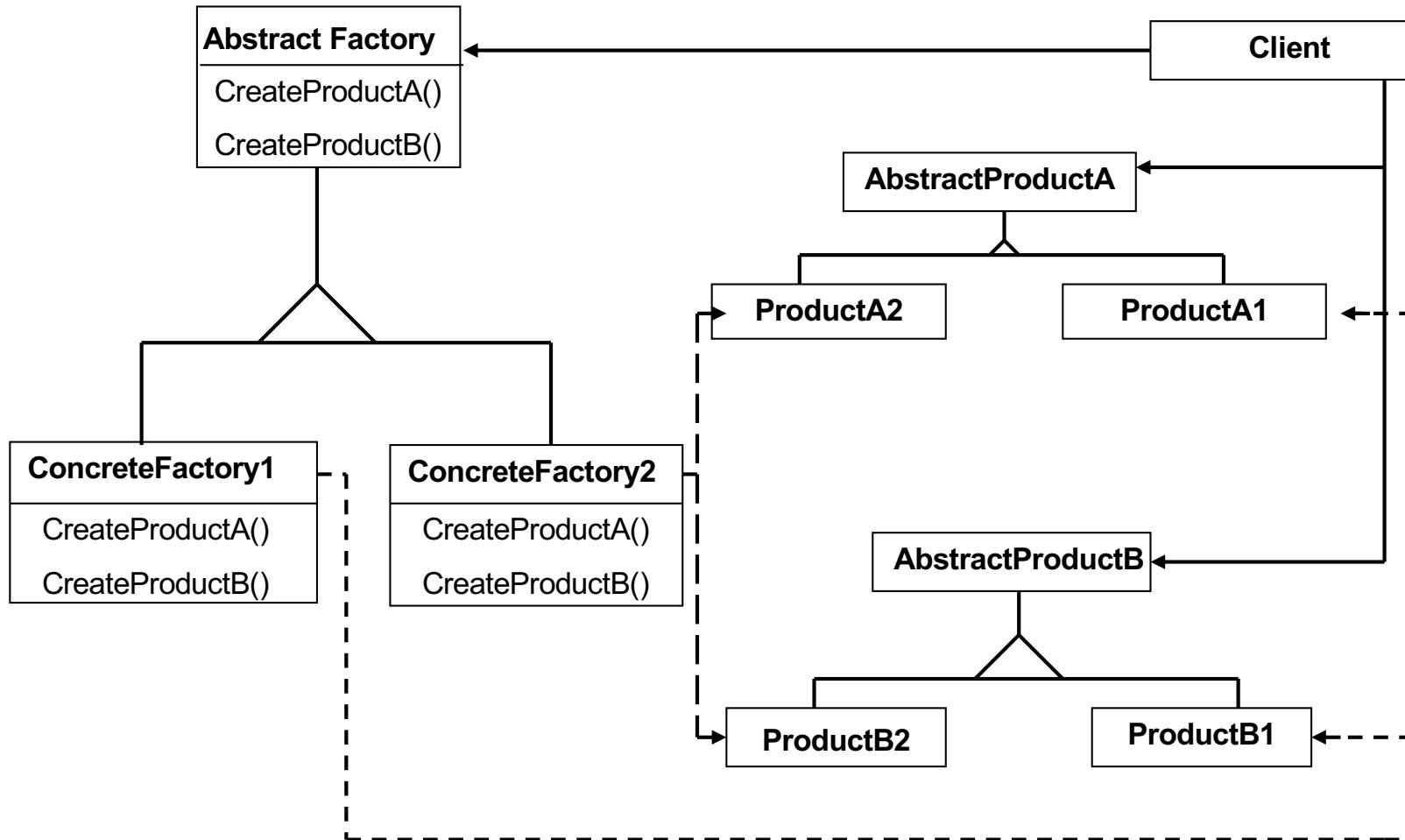
Diagramma: abstract products



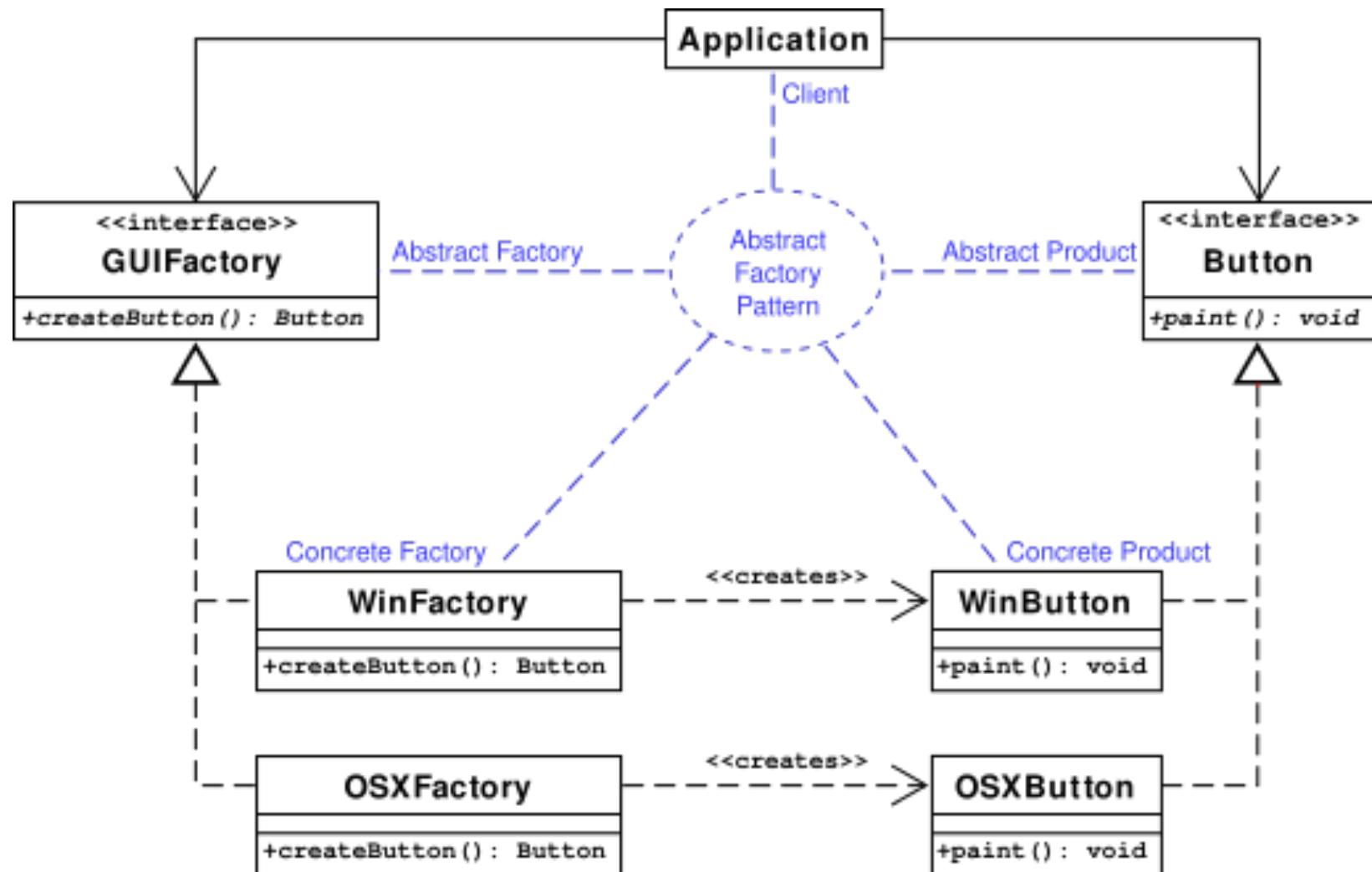
Pattern Abstract Factory: intento

- Questo diagramma segue il pattern “**Abstract Factory**”
- **Intento:** definire una classe che
 - Astrae la creazione di una famiglia di oggetti;
 - Istanze diverse costituiscono implementazioni diverse di membri di tale famiglia
- NB1: la “factory” corrente è una variabile globale
- NB2: la “factory” si può cambiare a runtime

Abstract Factory: Struttura del pattern



Abstract Factory



Problema: sistemi a finestre alternativi

- L'editor deve girare su diversi sistemi a finestre
- Gli standard dei sistemi a finestre sono molto diversi
 - Interfacce complesse
 - Diversi modelli di operazioni sulle finestre
 - Diverse funzionalità

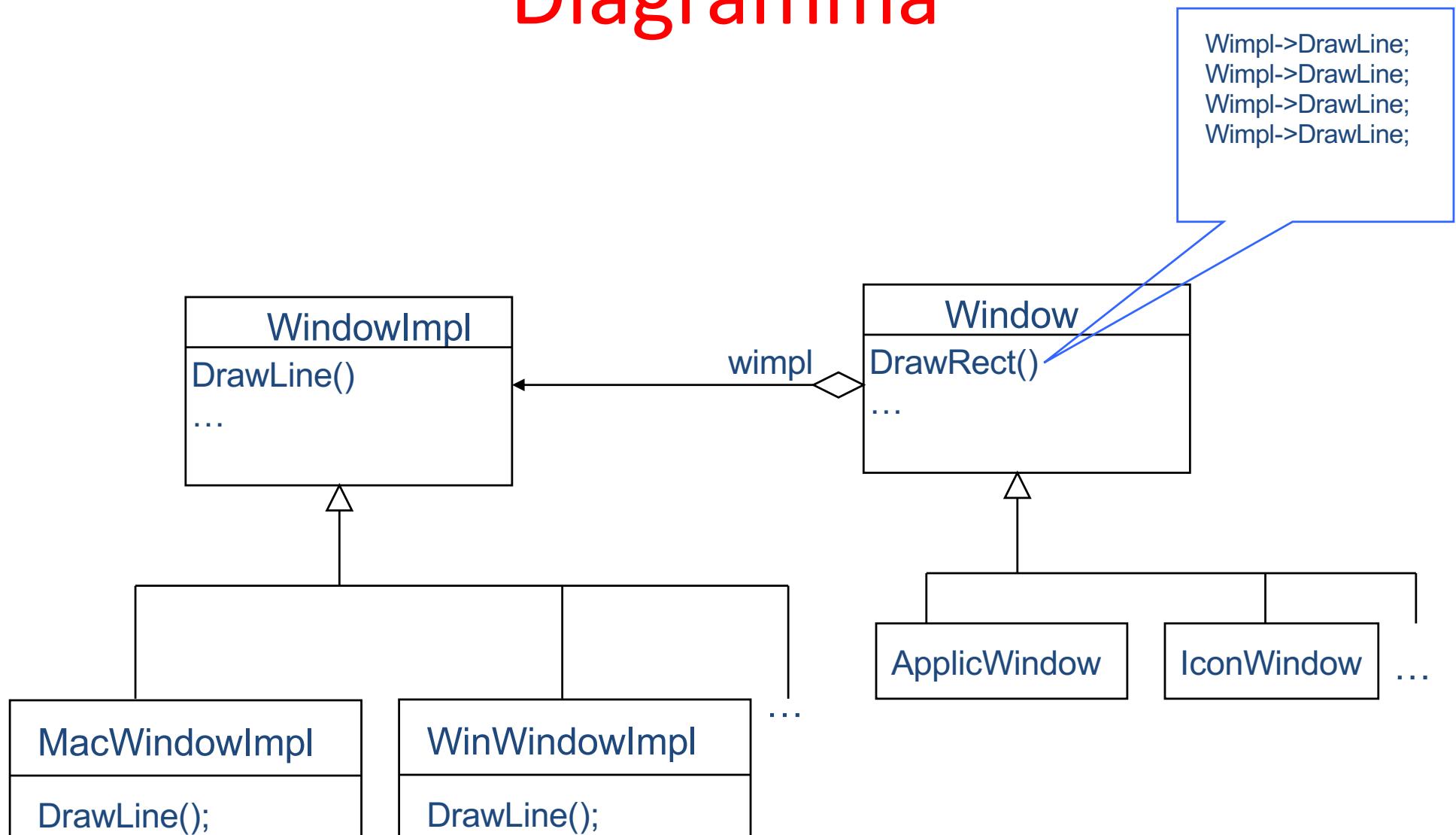
Una soluzione?

- Concentrarsi sulle operazioni presenti in tutti i modelli a finestre
- Definire un'Abstract Factory per encapsulare le varianti, creando “oggetti finestra” per il sistema corrente
- Problema: questa intersezione può non essere sufficientemente completa

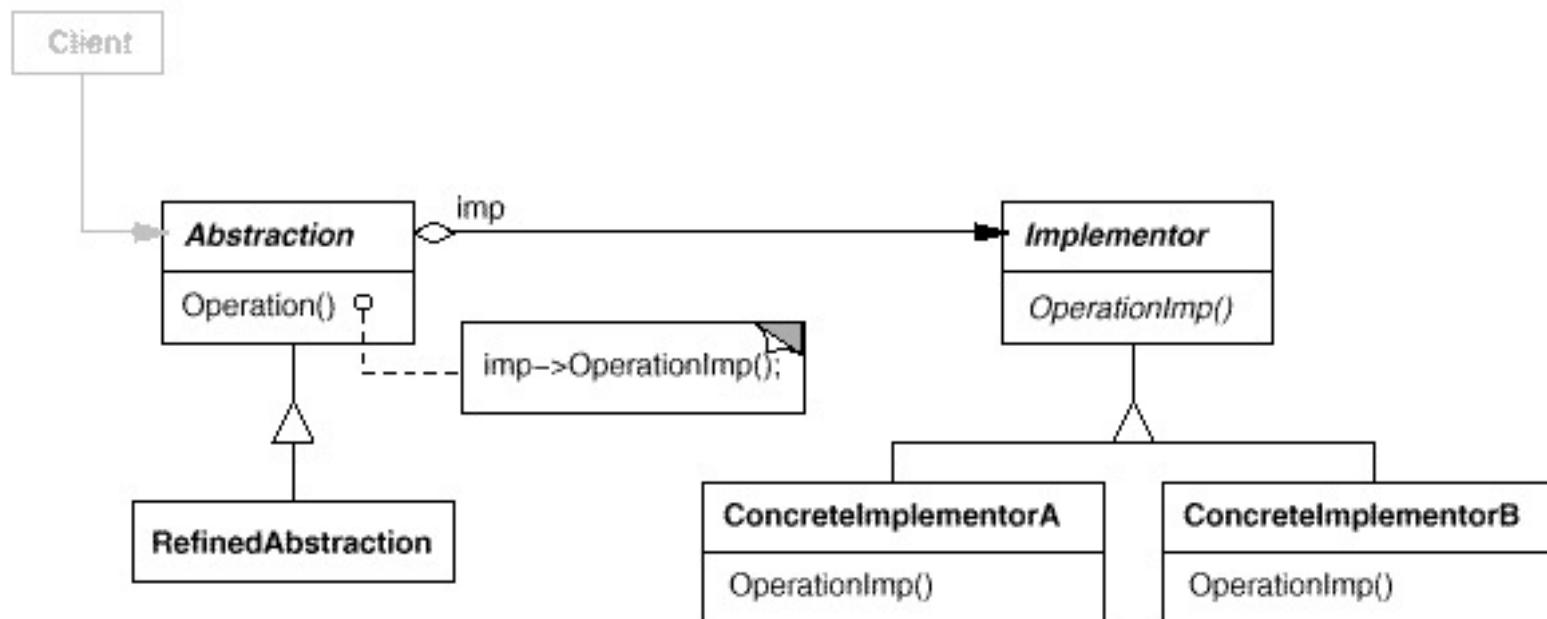
Soluzione

- Definiamo un nostro modello gerarchico a finestre
 - Ci mettiamo tutte le operazioni necessarie
 - Vantaggio: il modello è specializzabile sulla nostra applicazione
- Poi usiamo il modello per definire una gerarchia “parallela” (WindowImpl)
 - WindowImpl astrae (nasconde) i diversi sistemi a finestre concreti
 - WindowImpl contiene tutte le funzioni che ci servono

Diagramma



Pattern Bridge: struttura

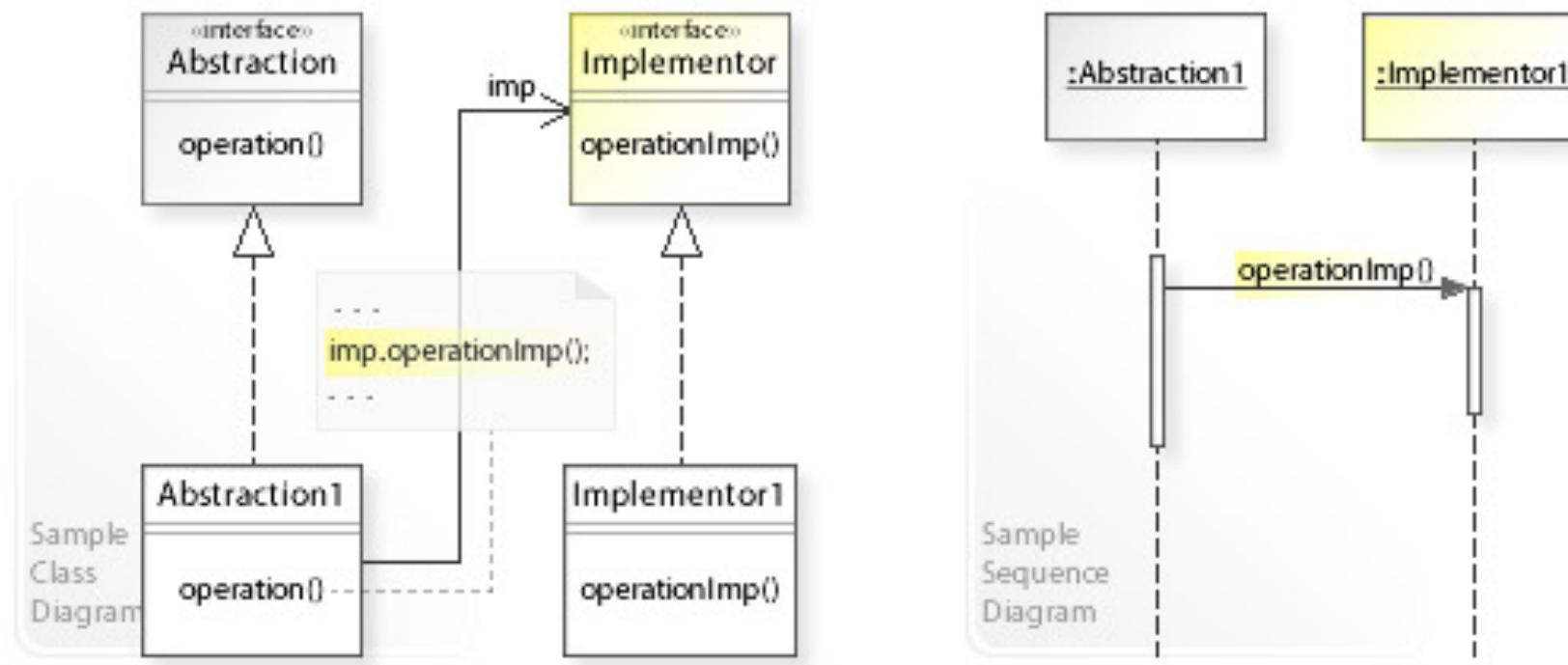


Pattern Bridge: motivazione

- Questo è il pattern “Bridge” (ponte)
- **Motivazione:** definire due gerarchie che possono evolvere in modo indipendente finché viene conservato il “ponte”:
 - Gerarchia logica
 - Legata all'applicazione
 - Gerarchia implementativa
 - Interfaccia col mondo esterno
 - Classe radice astratta con implementazioni multiple

Pattern Bridge: Intento

Intento: separa l'interfaccia di una classe dalla sua implementazione



Problema: controllo lessicale (Spell checking)

- Il controllo lessicale richiede l'analisi dell'intero documento
- Ogni glifo va visitato in sequenza
- L'informazione utile può essere distribuita sull'intero documento
- L'analisi lessicale è utile per altre analisi, es.: grammaticale

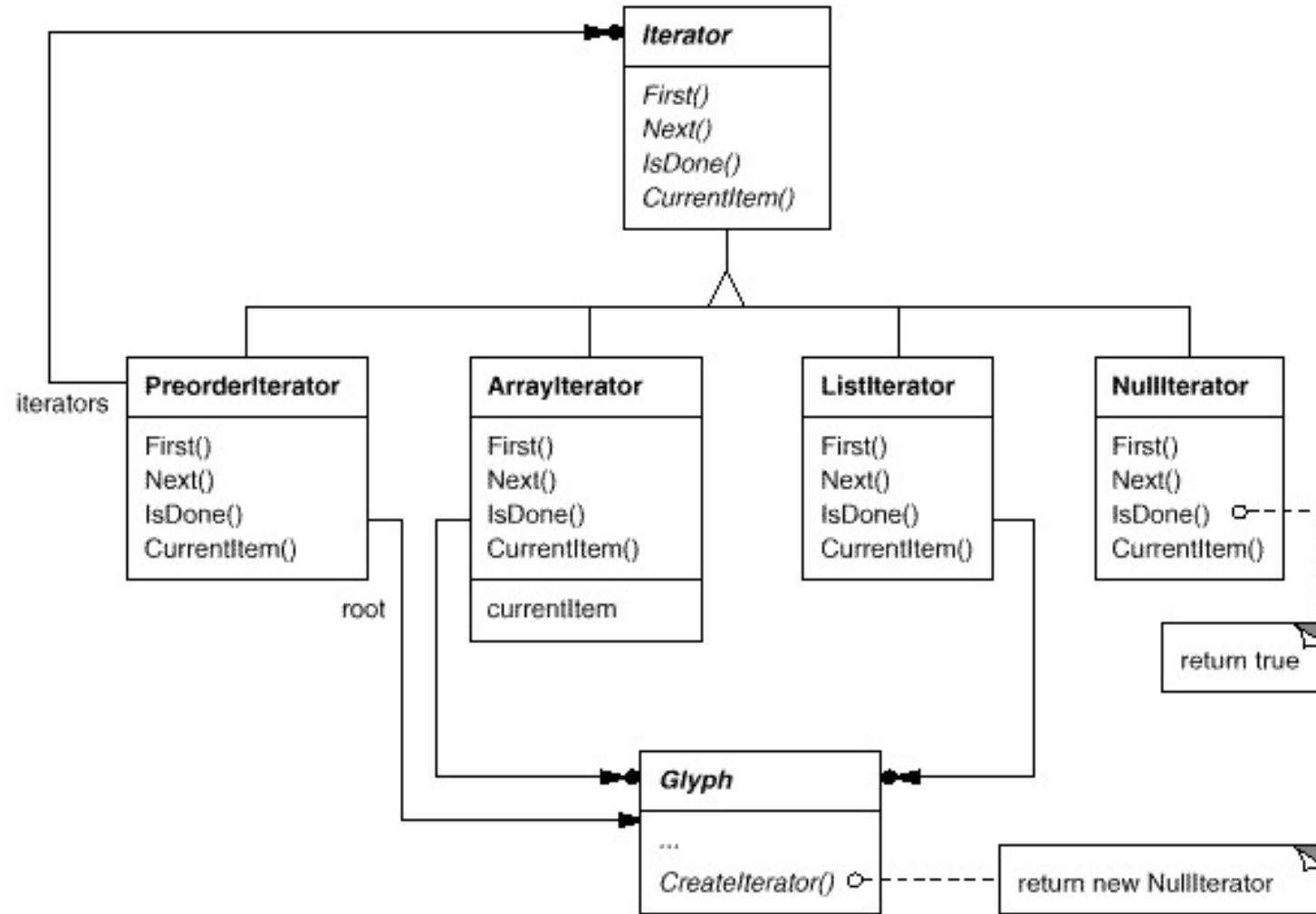
Soluzione?

- Usare un comando iteratore

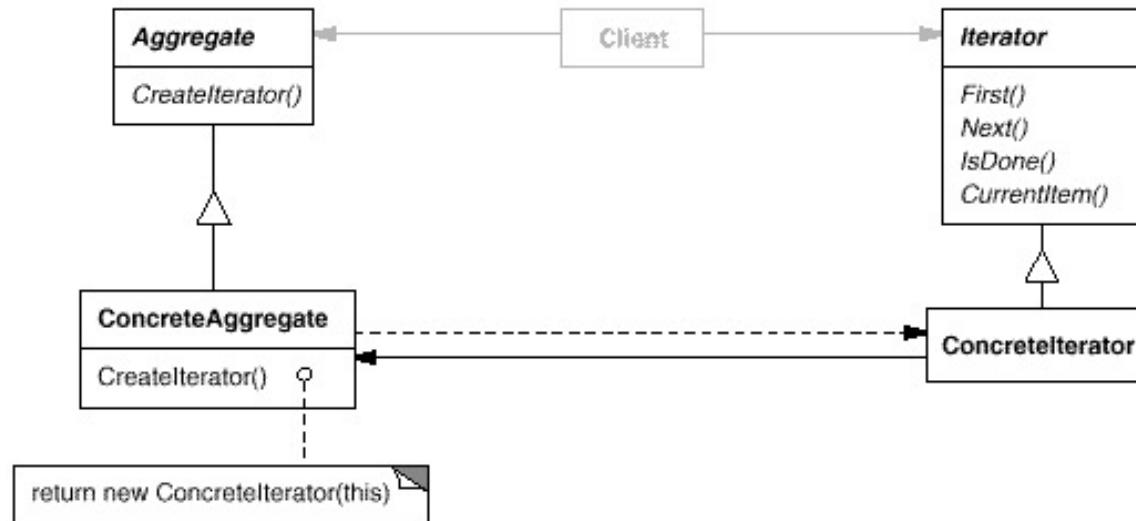
```
Iterator i= CreateIterator(glyph)  
for (i=i.first(); !(i.isdone()); i=i.next())  
{ usare il glifo i.current()...; }
```

- Nasconde ai clienti la struttura del contenitore
- funziona senza bisogno di conoscere il tipo degli elementi su cui si ripete l'operazione

Pattern Iterator: esempio



Pattern iterator: struttura



Il *pattern iterator* definisce due gerarchie di classi: una per i contenitori-aggregati e una per gli iteratori.

Gli aggregati possono essere specializzati per tipo di elemento contenuto o per tipo di struttura (es. lista, vettore, ecc.)

Le classi di iteratori sono specializzate per tipo di contenitore (iteratore concreto) e per tipo di navigazione attraverso la sequenza di elementi (iteratori specializzati).

Pattern Iterator: partecipanti

- **Iterator:** definisce un'interfaccia per attraversare l'insieme degli elementi di un contenitore e accedere ai singoli elementi.
- **ConcretIterator:** implementa l'interfaccia Iterator tenendo traccia della posizione corrente nel contenitore e calcolando qual è l'elemento successivo nella sequenza di attraversamento.
- **Aggregate:** definisce un'interfaccia per creare un oggetto Iterator.
- **ConcreteAggregate:** implementa l'interfaccia di creazione dell'Iterator e ritorna un'istanza appropriata di ConcretIterator.

Iterator: conseguenze

- Semplifica l'interfaccia del contenitore, rimuovendo la parte che permette di navigare attraverso gli elementi contenuti.
- Isola i dettagli di implementazione e la struttura del contenitore, e nasconderli ai programmi che accedono ai dati, che utilizzano soltanto le interfacce del contenitore e dell'iteratore.
- Accede agli elementi di un contenitore attraverso un'interfaccia uniforme e indipendente dal tipo di contenitore (iterazione polimorfica).
- Può attraversare contemporaneamente lo stesso aggregato di elementi con più processi indipendenti. Ciascun processo può usare un iteratore differente, che tiene traccia della propria posizione nella sequenza.
- Attraversa lo stesso aggregato di elementi con modalità diverse. Più iteratori specializzati, ciascuno eventualmente fornito di interfaccia specifica, possono calcolare in modo diverso l'elemento successivo, e quindi definiscono modi diversi di attraversare l'insieme degli elementi.

Problema: con Iterator complichiamo Gliph

- L'interfaccia di Gliph si arricchisce (ovvero si complica) per ogni nuova analisi quale lo spell checking
- Sarebbe bene invece encapsulare l'analisi in una classe specifica, separando l'algoritmo dalla struttura dati cui è applicato
- Soluzione: invece di Iterator, usare Visitor

Pattern Visitor

Il pattern **Visitor** è un iteratore generico

- permette di attraversare un contenitore complesso (es.: *composite*)
- **Intento:** separa un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa
- I visitatori oltre ad attraversare il contenitore applicano azioni dipendenti dal tipo degli elementi, che possono essere ridefinite (*overriding*) per diverse modalità di attraversamento

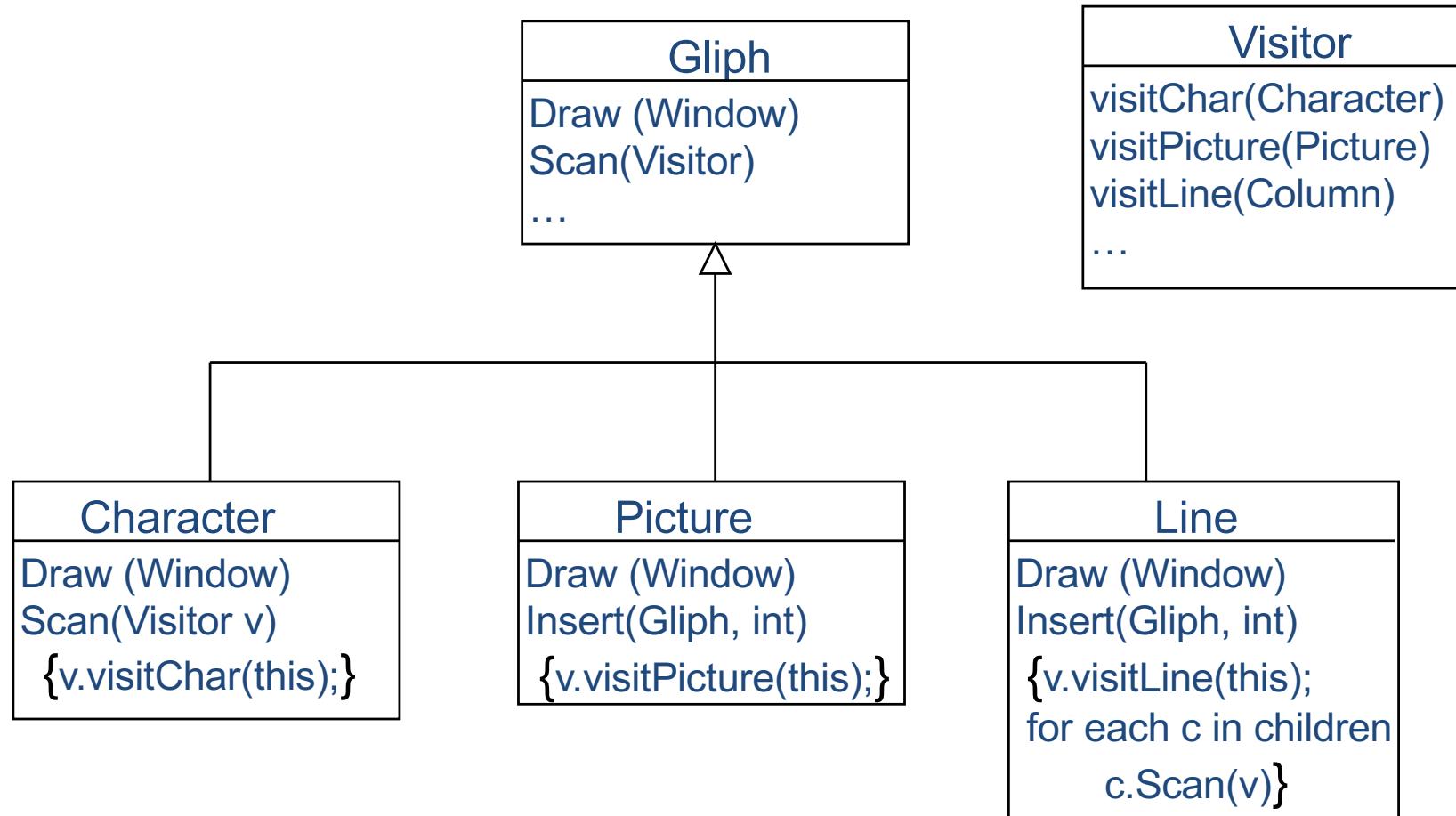
Visitor vs Iterator

la differenza è abbastanza netta:

- il pattern **Iterator** consente di accedere sequenzialmente ad una struttura dati indipendentemente dalla sua rappresentazione interna.
- il pattern **Visitor** consente invece di eseguire una operazione su elementi di una struttura dati; operazione che può cambiare senza dover cambiare anche l'oggetto su cui opera, e che può essere diversa per ogni tipo di oggetto.

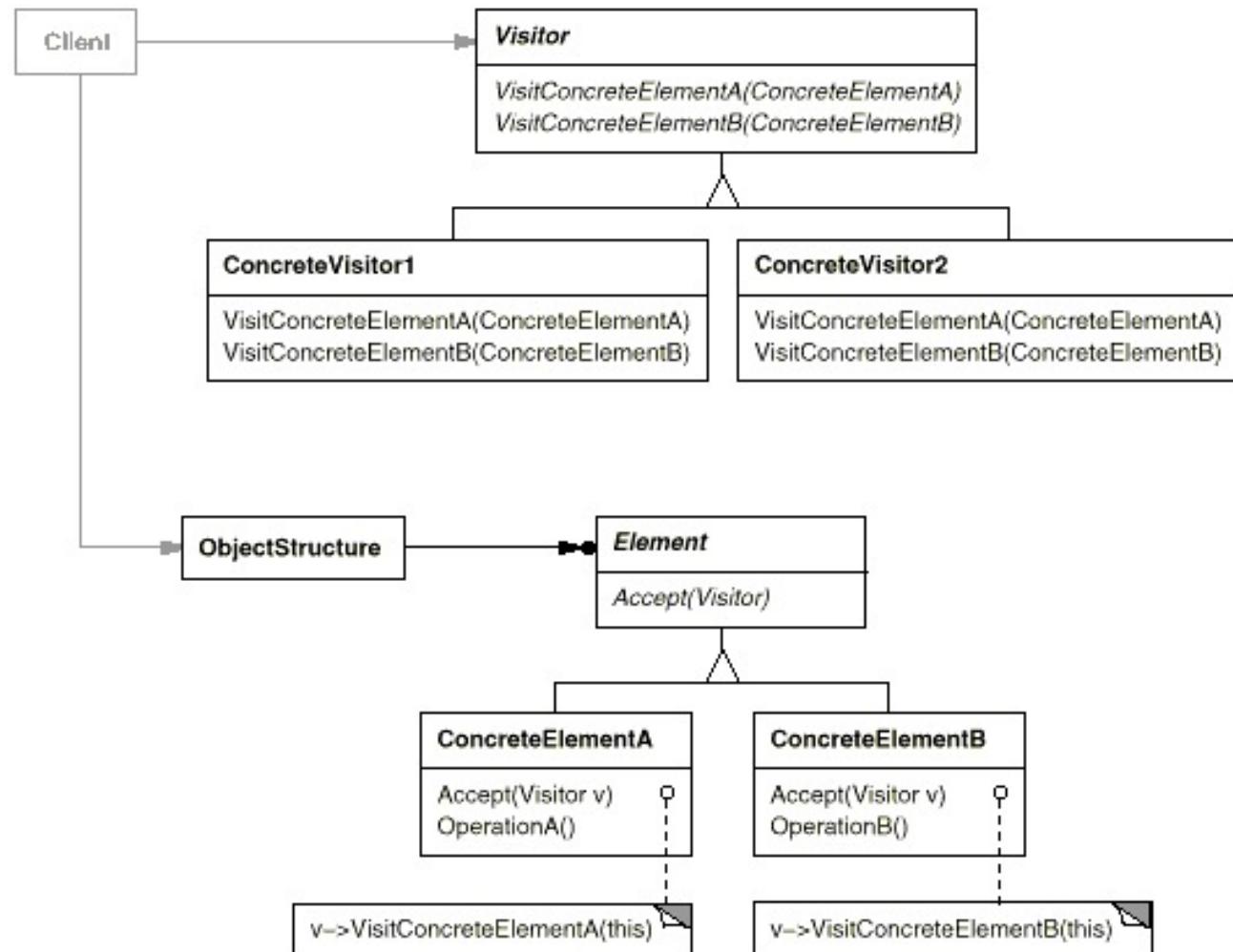
In genere visitor *usa iterator*

Pattern visitor: esempio



Pattern visitor: struttura

Visitor è utile quando:
una struttura di oggetti è costituita da molte classi con interfacce diverse ed è necessario che l'algoritmo esegua su ogni oggetto un'operazione differente a seconda dalla classe concreta dell'oggetto stesso, è necessario eseguire svariate operazioni indipendenti e non relazionate tra loro sugli oggetti di una struttura composta, ma non si vuole sovraccaricare le interfacce delle loro classi.



Pattern Visitor: conseguenze

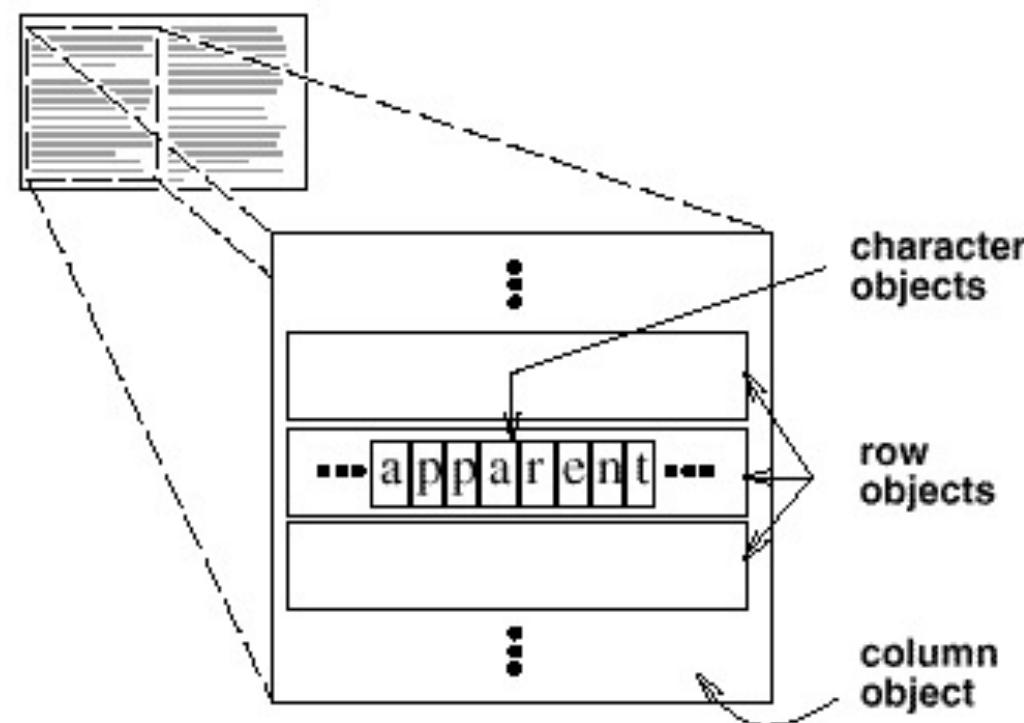
- Flessibilità delle operazioni
 - Organizzazione logica
 - Visita di vari tipi di classe
 - Mantenimento di uno stato aggiornabile ad ogni visita
-
- Rigidità della gerarchia di classi
 - Violazione dell'incapsulamento

Pattern Visitor

- Il pattern Visitor è complesso
- Di solito conviene usarlo quando si tratta con una struttura gerarchica di elementi disomogenei
- Le diverse modalità di visita della struttura possono essere definite come sottoclassi del Visitor

Problema: granularità degli oggetti

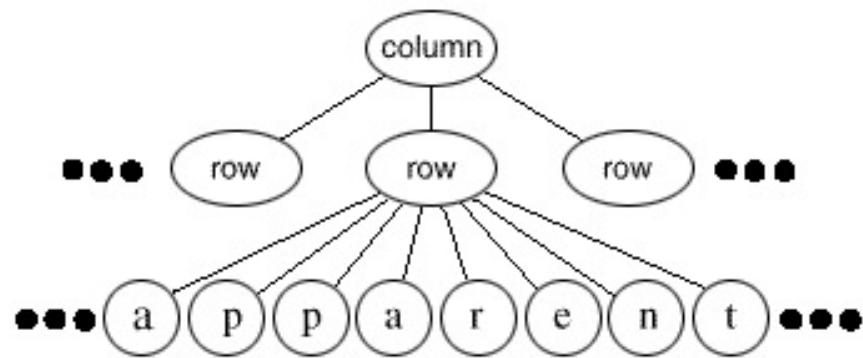
- Un text editor non dovrebbe trattare ogni glifo come singolo oggetto separato: un documento sarebbe composto da una miriade di oggetti in sostanza identici (ovvero tutte le a, tutte le b, ecc.)
- D'altra parte non vogliamo rinunciare alla flessibilità di avere glifi come oggetti



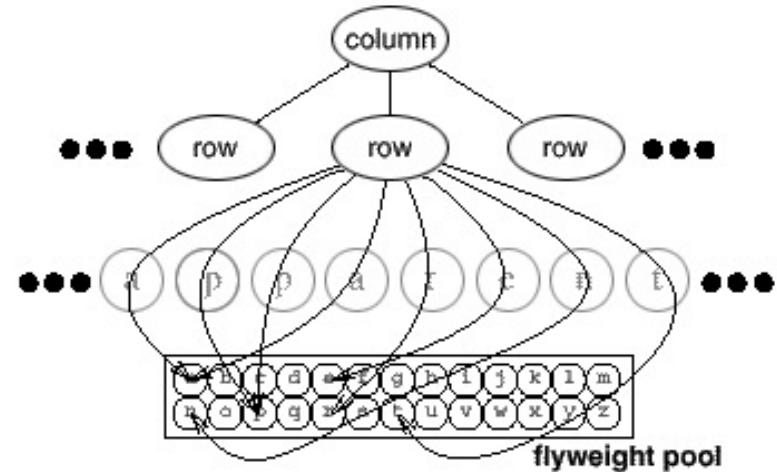
Pattern Flyweight

- Un *flyweight* è un oggetto condiviso che si può usare simultaneamente in più contesti
 - Esempio: i glifi entro un documento
- Il flyweight agisce come oggetto indipendente in ciascun contesto – è indistinguibile da un'istanza non condivisa di tale oggetto
- I flyweights non possono fare alcuna assunzione sul contesto in cui operano

Flyweight



Un oggetto per ogni occorrenza di un certo carattere entro il documento



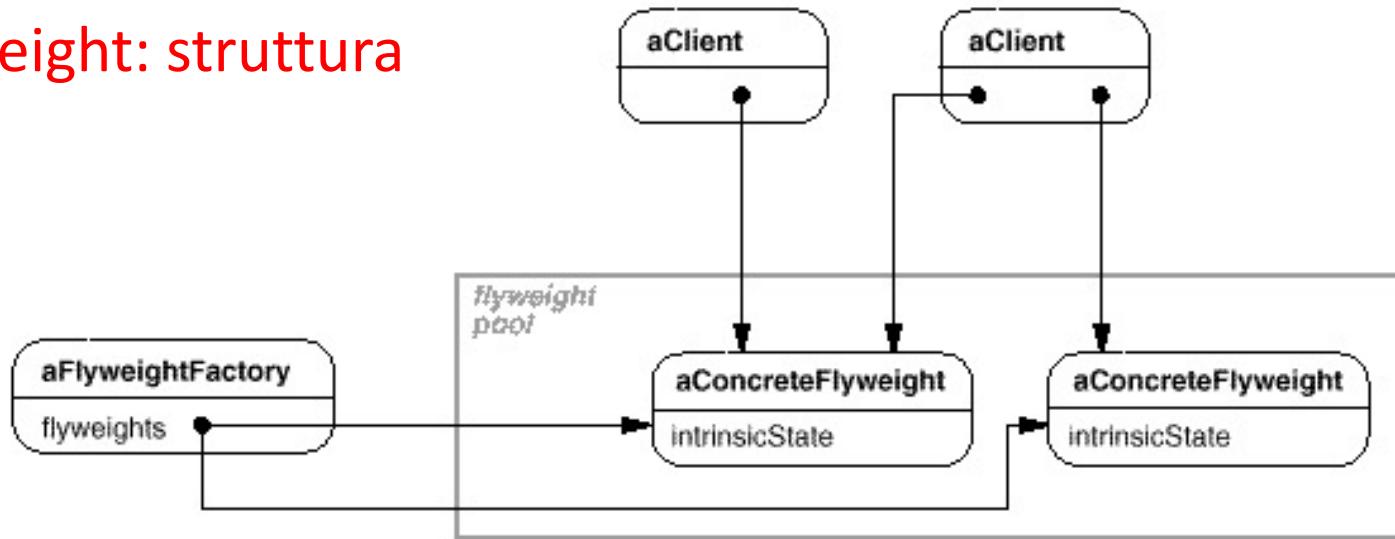
Esiste un unico oggetto flyweight per ciascun carattere, e può essere usato in diversi contesti entro il documento.

Ciascuna occorrenza di un certo oggetto carattere riferisce la stessa istanza nell'insieme condiviso (pool) degli oggetti flyweight

Pattern Flyweight: applicabilità

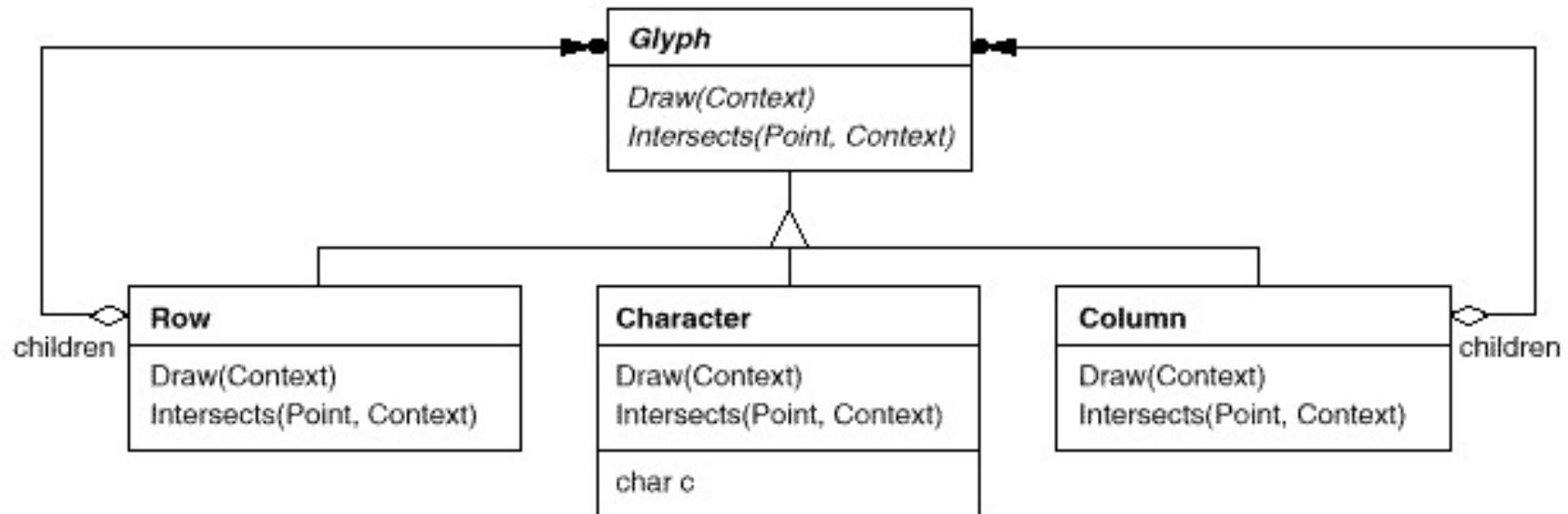
- Un'applicazione usa un gran numero di oggetti: i costi di memoria sono alti a causa della gran quantità di oggetti
- Gran parte dello stato di un oggetto può essere reso intrinseco all'oggetto.
- Molti gruppi di oggetti possono essere rimpiazzati da pochi oggetti condivisi dato che lo stato estrinseco è stato rimosso in gran parte.
- L'applicazione non dipende dall'identità degli oggetti.
- Poiché gli oggetti flyweight possono essere condivisi, i test di identità restituiranno vero anche per oggetti concettualmente distinti.

Flyweight: struttura



- **Flyweight**: dichiara l'interfaccia mediante cui i flyweights ricevono e agiscono sullo stato estrinseco
- **ConcreteFlyweight** (eg. Character): implementa l'interfaccia Flyweight e aggiunge lo stato intrinseco. Un oggetto ConcreteFlyweight dev'essere condivisibile. Ogni stato che memorizza è intrinseco ovvero indipendente dal contesto del ConcreteFlyweight
- **Flyweight factory**: crea e gestisce oggetti flyweight; ne assicura la corretta condivisione
- **UnsharedConcreteFlyweight** (eg. Row, Column): non tutte le sottoclassi di Flyweight saranno condivise. L'interfaccia Flyweight abilita la condivisione; non la forza. Normalmente gli oggetti UnsharedConcreteFlyweight hanno come figli ConcreteFlyweight a qualche livello nella struttura di condivisione (nell'esempio: le classi Row e Column)

Pattern Flyweight: esempio



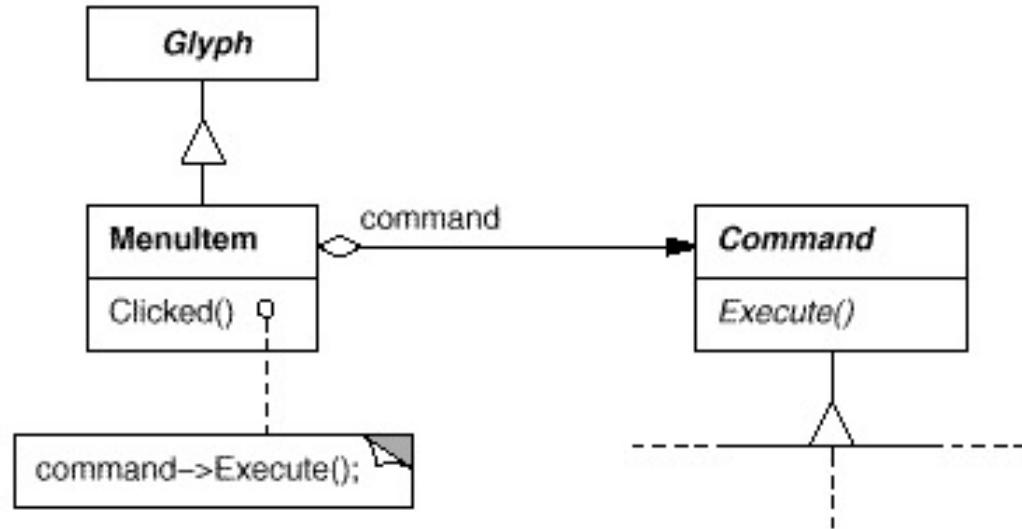
Flyweight: conseguenze

- Flyweight si combina spesso con Composite
- Flyweights può aggiungere costi a runtime per trasferire, trovare, e calcolare lo stato estrinseco, specie se era stato prima memorizzato come stato intrinseco.
- Tuttavia tali costi sono giustificati dai risparmi di memoria, che aumentano man mano che vengono condivisi più oggetti flyweights .
- I risparmi di memoria però dipendono da molti fattori:
 - La riduzione del numero totale di istanze che deriva dalla condivisione
 - La quantità di stato intrinseco per ciascun oggetto
 - Se lo stato estrinseco viene calcolato o memorizzato.

Problema: comandi dell'editor

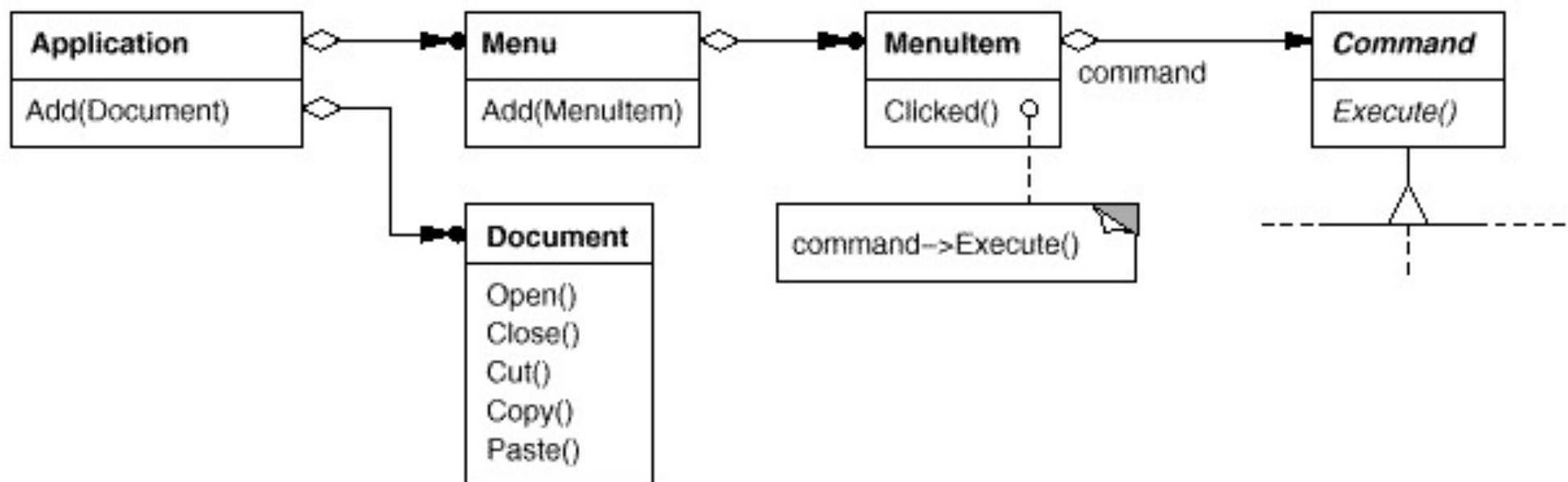
- Creare un nuovo documento,
- Aprire salvare e stampare un documento,
- Cut e paste,
- Modifica di fonte o stile di testo selezionato,
- Modifica della formattazione (allineamento e giustificazione),
- undo (non tutti: eg. Print non si può disfare)
- Uscire dall'editor - quit

Soluzione

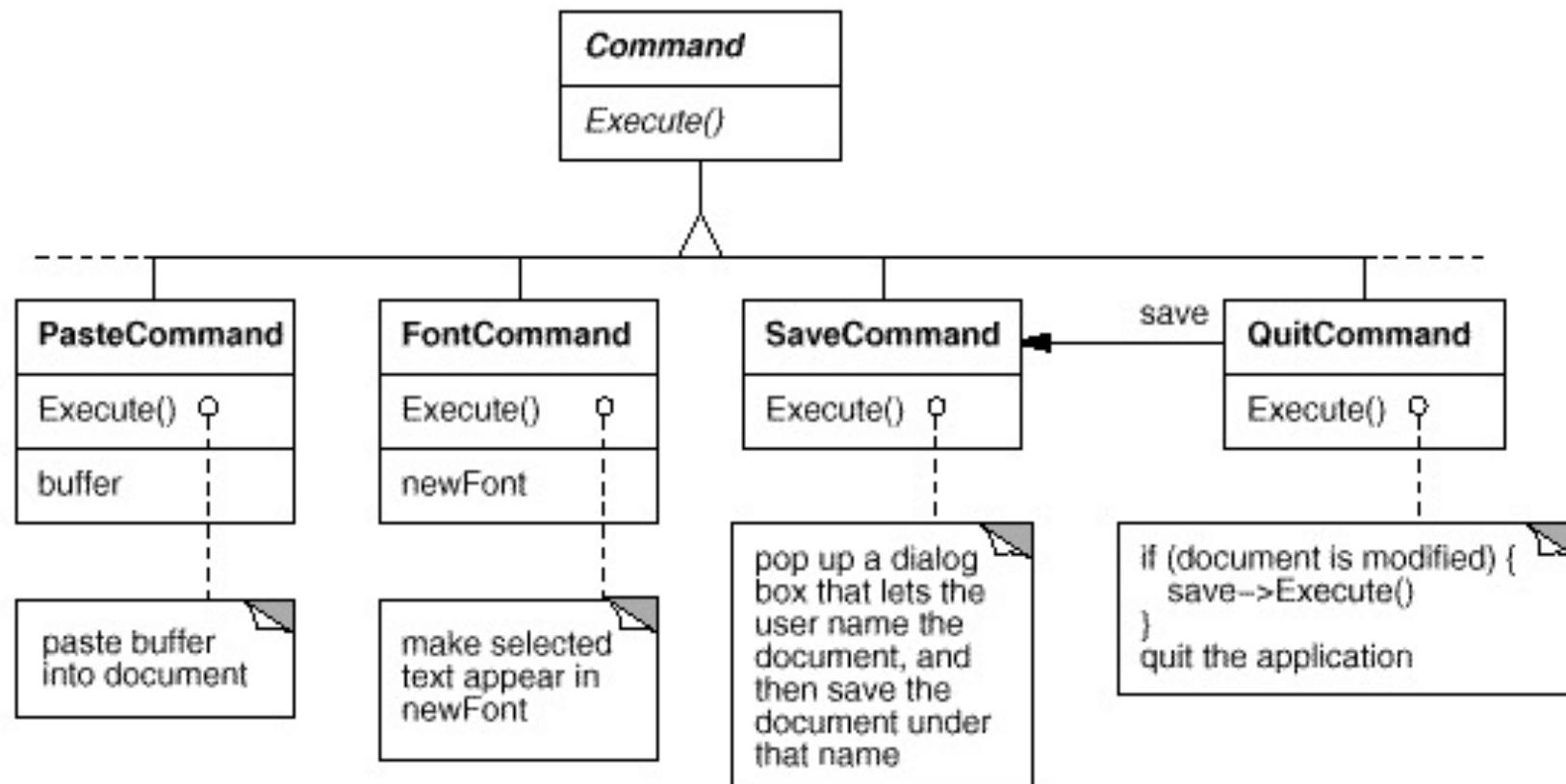


- Un menù a scomparsa per i comandi è uno speciale glifo che contiene altri glifi.
- Un menù a scomparsa è un glifo speciale con glifi nel menù che eseguono qualche operazione.
- Supponiamo che questi glifi attivi siano una sottoclasse di **Glyph** chiamata **MenuItem** e che eseguano un comando invocato da qualche cliente

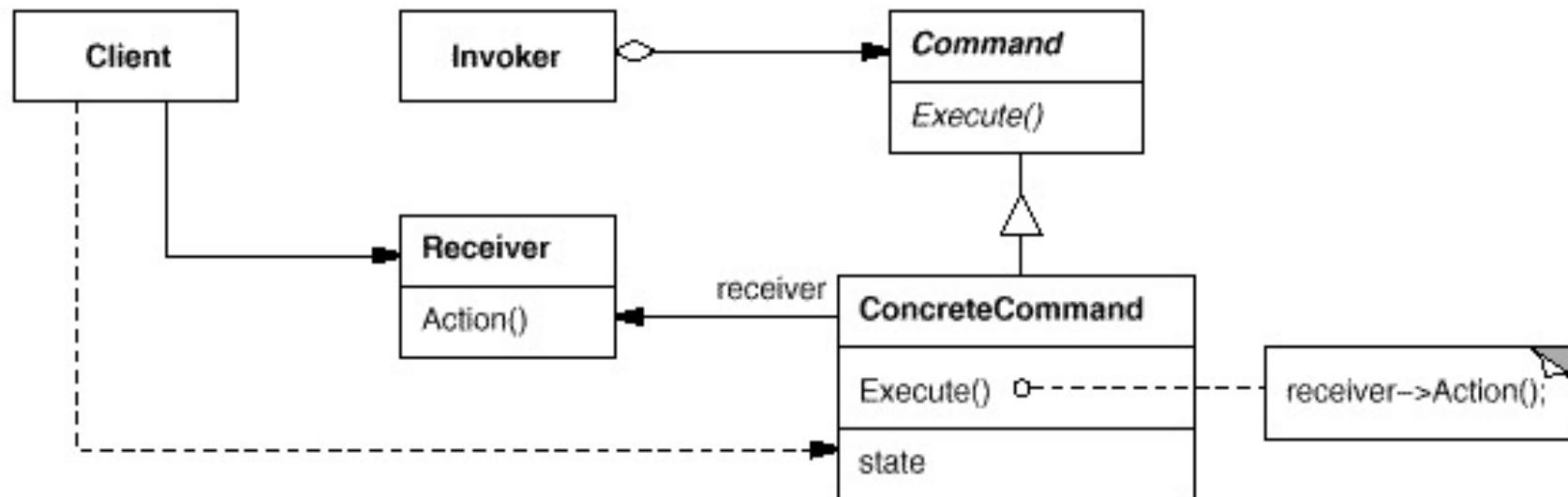
Architettura dell'editor



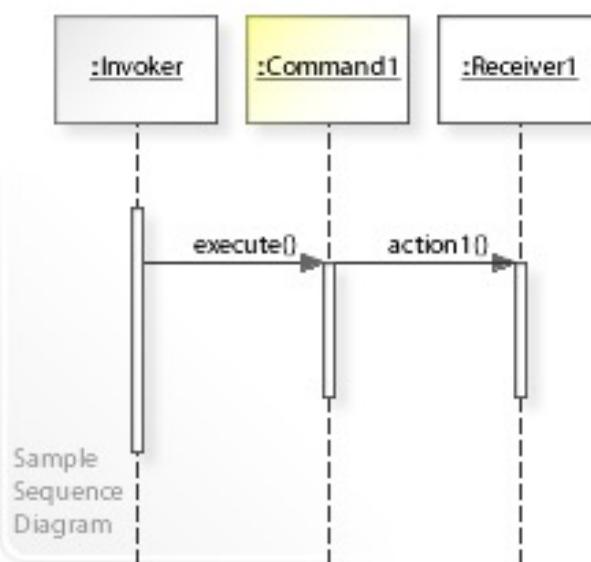
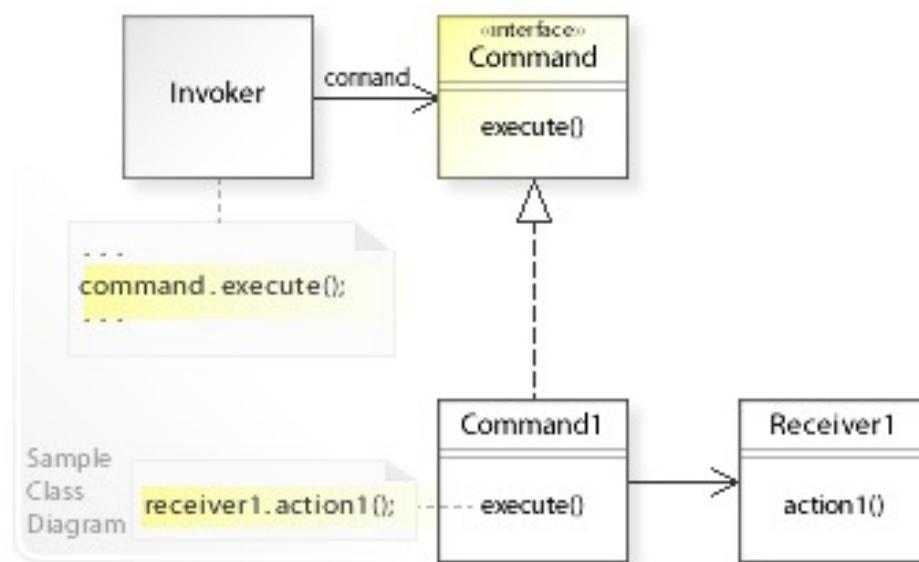
Pattern Command: esempio



Pattern Command: struttura



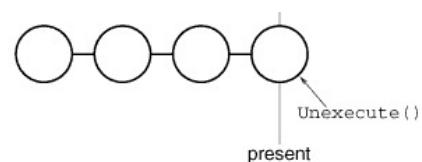
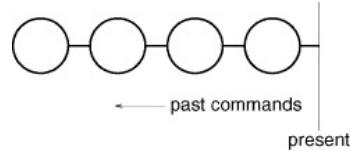
Pattern Command: dinamica



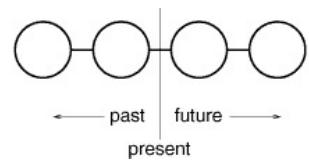
Problema: come disfare un comando tornando allo stato precedente

- Lo stato interno di un oggetto dovrebbe essere salvato al suo esterno in modo da poterlo restaurare in seguito.
- L'incapsulamento dell'oggetto non deve essere violato.
- Il problema è che un oggetto ben progettato è encapsulato e la sua rappresentazione è nascosta e inaccessibile dall'esterno dell'oggetto.

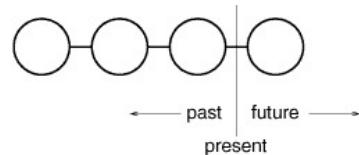
La storia dei comandi



undo



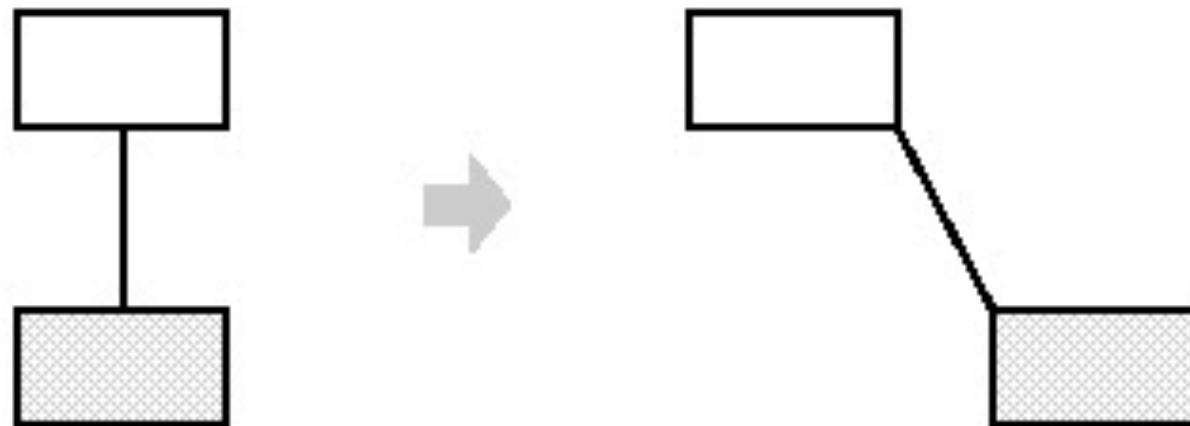
effetto di undo



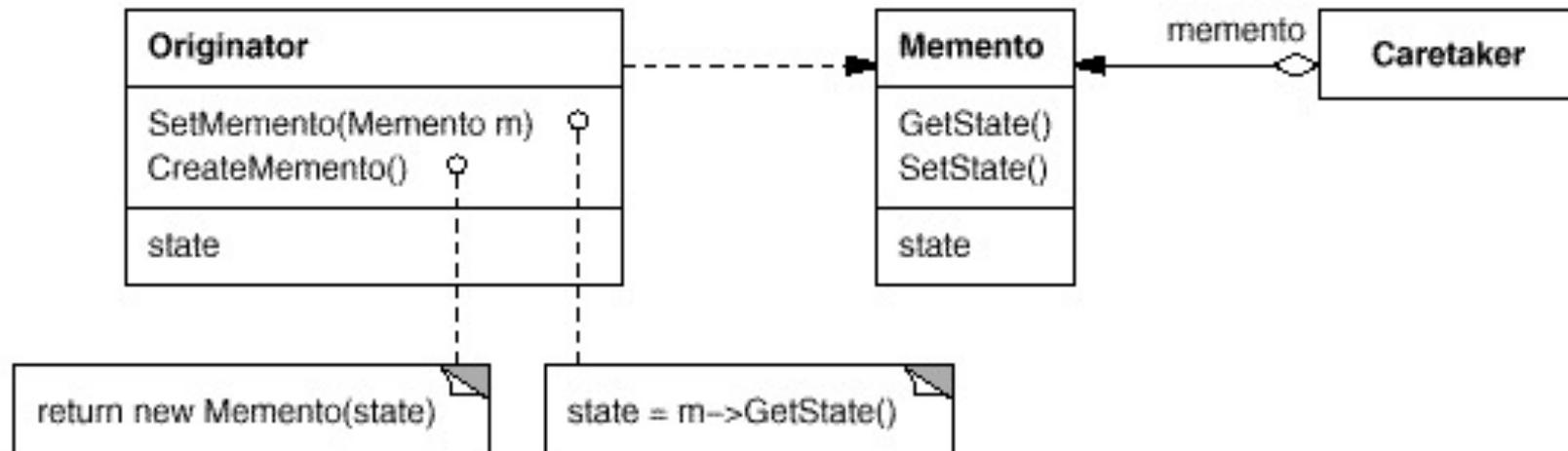
redo

Pattern Memento: intento

- **Intento:** senza violare l'incapsulamento, catturare e portar fuori lo stato interno di un oggetto, in modo che l'oggetto stesso possa recuperare questo stato più tardi.
- **Esempio:** un editor grafico che supporta linee tra figure. Un utente connette due rettangoli con una linea, e i rettangoli rimangono connessi quando l'utente li muove.
- Supportare *undo* in questo caso non è facile

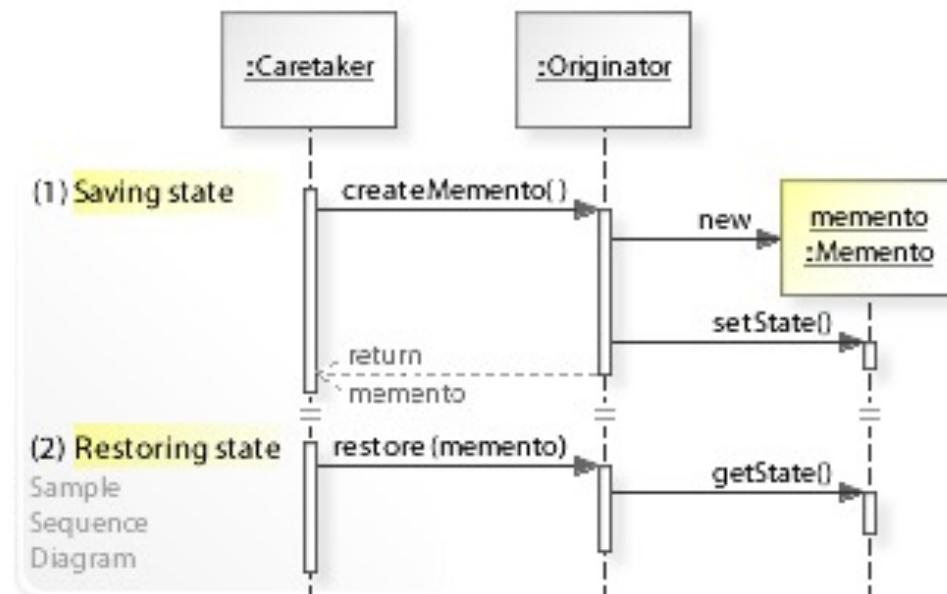
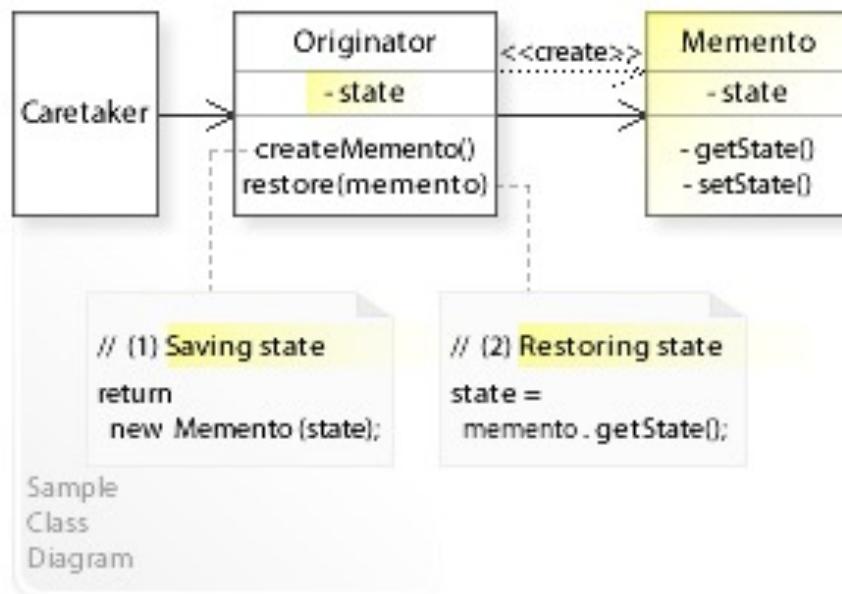


Pattern Memento: struttura



- **Originator:** memorizza in Memento il proprio stato interno; lo protegge contro tutti gli accessi altri che i propri
- **Memento:** crea un memento contenente un'istantanea dello stato interno corrente
- **Caretaker** (eg. Undo mechanism): responsabile della corretta gestione del Memento

Pattern Memento: dinamica



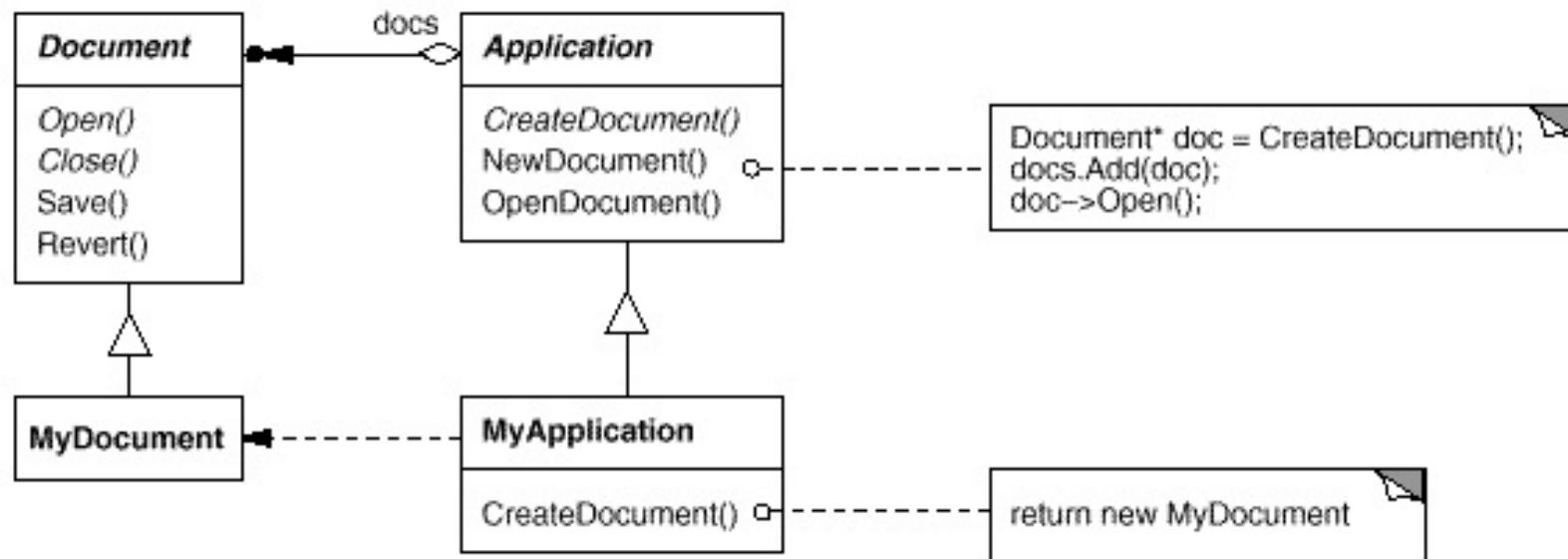
Pattern Factory method: intento

- Factory Method fa sì che una classe possa differire l'istanziazione di un oggetto alle sottoclassi
- Il pattern Factory Method definisce un'interfaccia di classe per la creazione di un oggetto, lasciando ai tipi derivati la decisione su quale oggetto debba essere effettivamente istanziato.
- Il pattern è utile quando una classe non è in grado di conoscere a priori il tipo di oggetti da creare oppure quando si vuole delegare la creazione di un oggetto alle sottoclassi.
- L'applicazione del pattern consente di eliminare le dipendenze dai tipi concreti utilizzati

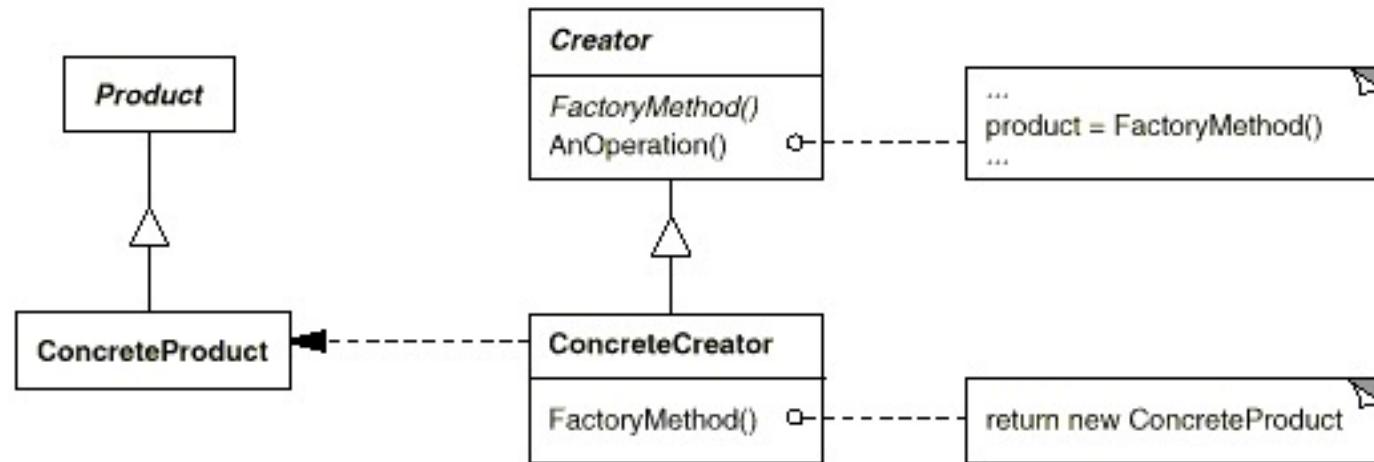
Pattern Factory Method: motivazione

- Si consideri un framework che manipola documenti, Le due astrazioni chiave sono *l'Applicazione* e il *Documento*, entrambe sono astratte e i client devono creare sottoclassi per realizzare l'implementazione specifica.
- Poiché la sottoclasse di Documento da istanziare è application-specific, l'Applicazione **sa quando** un documento deve essere creato, **ma non sa quale tipo** creare.
- Un FactoryMethod incapsula la conoscenza riguardo quale tipo di sottoclasse di documento creare, svincolandolo dal framework.

Pattern Factory method: esempio

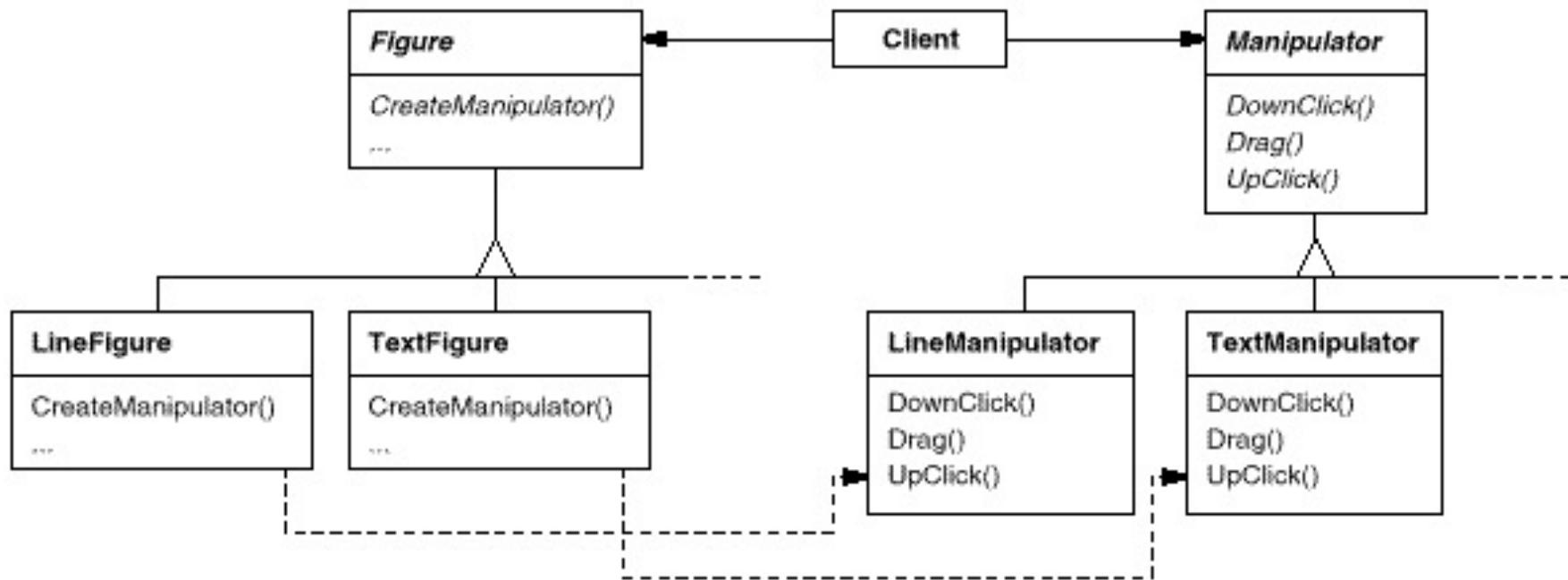


Pattern Factory method: struttura



- **Product (Document)** definisce l'interfaccia degli oggetti creati dal factory method .
- **ConcreteProduct (MyDocument)** implementa l'interfaccia del Product.
- **Creator (Application)**
 - dichiara il factory method, che ritorna un oggetto di tipo Product. Creator può anche definire un implementazione di default del factory method che restituisce un oggetto ConcreteProduct default.
 - Può invocare il factory method per creare un oggetto Product.
- **ConcreteCreator (MyApplication)** fa override del factory method per ritornare un istanza di ConcreteProduct

Pattern Factory method: uso

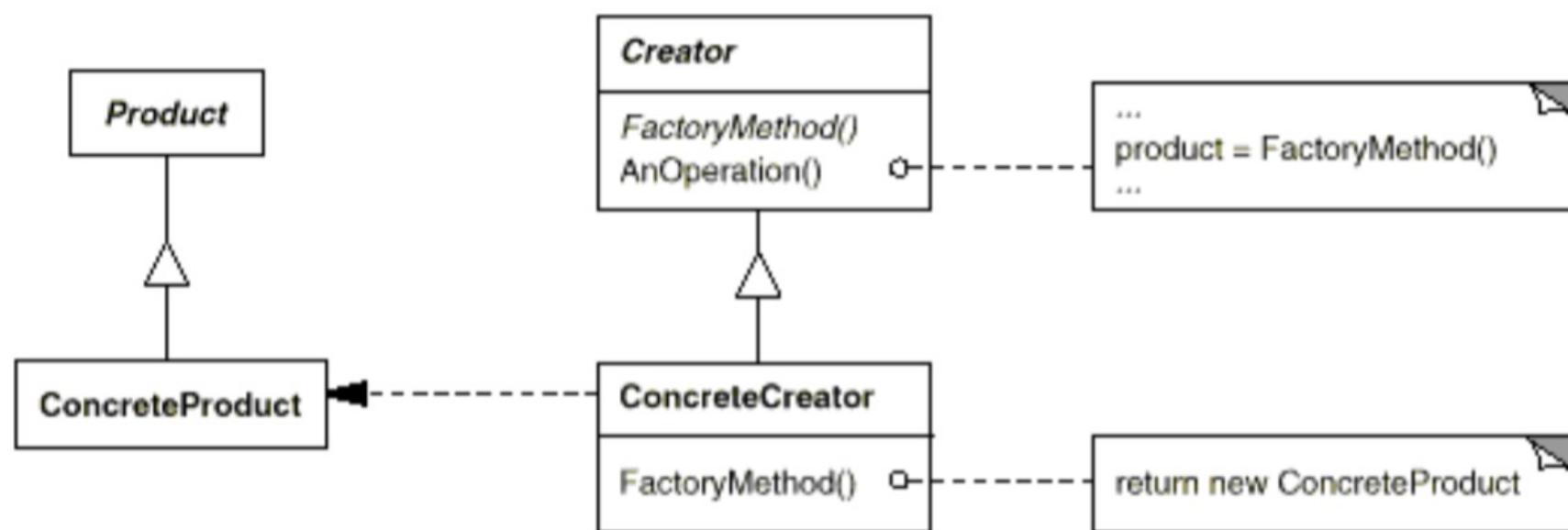


Pattern Factory Method: applicabilità

Il pattern Factory Method va usato quando:

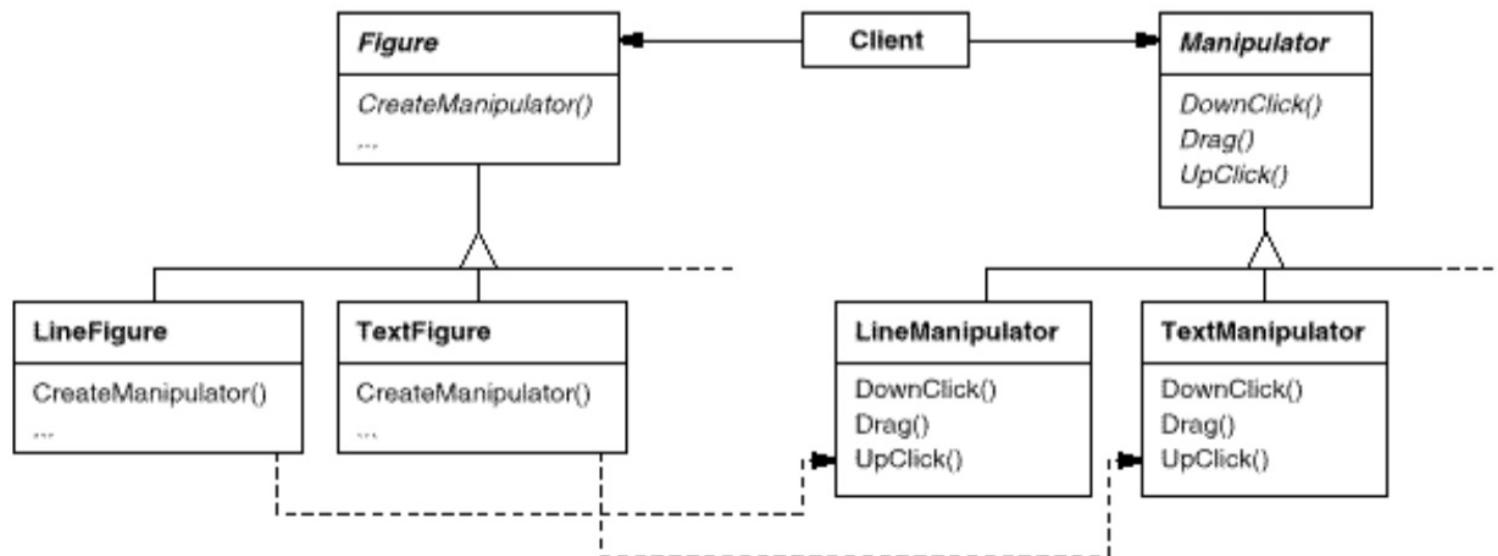
- Una classe non può sapere quale tipo di classi di oggetti deve creare.
- Una classe vuole lasciar decidere alle sottoclassi quali oggetti creare.

Pattern Factory Method: struttura



Factory Method: conseguenze

- Rimuove il bisogno di legare al codice classi application-specific ma forza a creare sottoclassi anche quando non ce ne sia il bisogno.
- Fornisce agganci per creare oggetti (metodi create), in un modo più flessibile rispetto a creare oggetti direttamente, favorendo la ridefinizione.
- Usando più Factory Method, un client può connettere *gerarchie di classi parallele* (quando una classe delega parte delle sue responsabilità a classi separate. Es. il *Document* ha il suo *DocumentFormatter*)

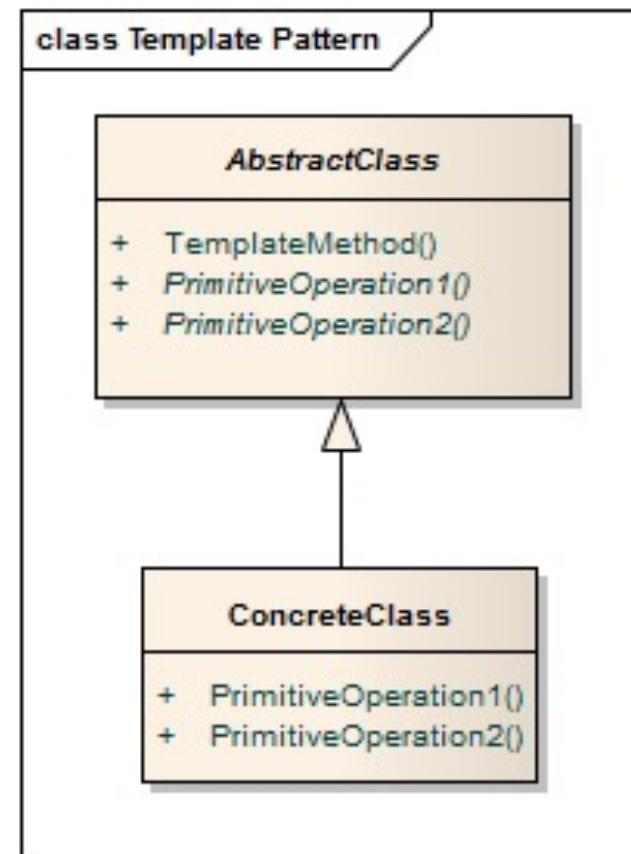


Da ricordare

- **Abstract Factory**: crea insiemi di oggetti separando i loro dettagli implementativi dal loro uso, e usa la composizione di oggetti
- **Factory method**: crea un oggetto senza specificare la classe dell'oggetto da creare
- **Template method**: (comportamentale) ridefinisce alcuni passi di un algoritmo senza modificare la struttura dell'algoritmo stesso

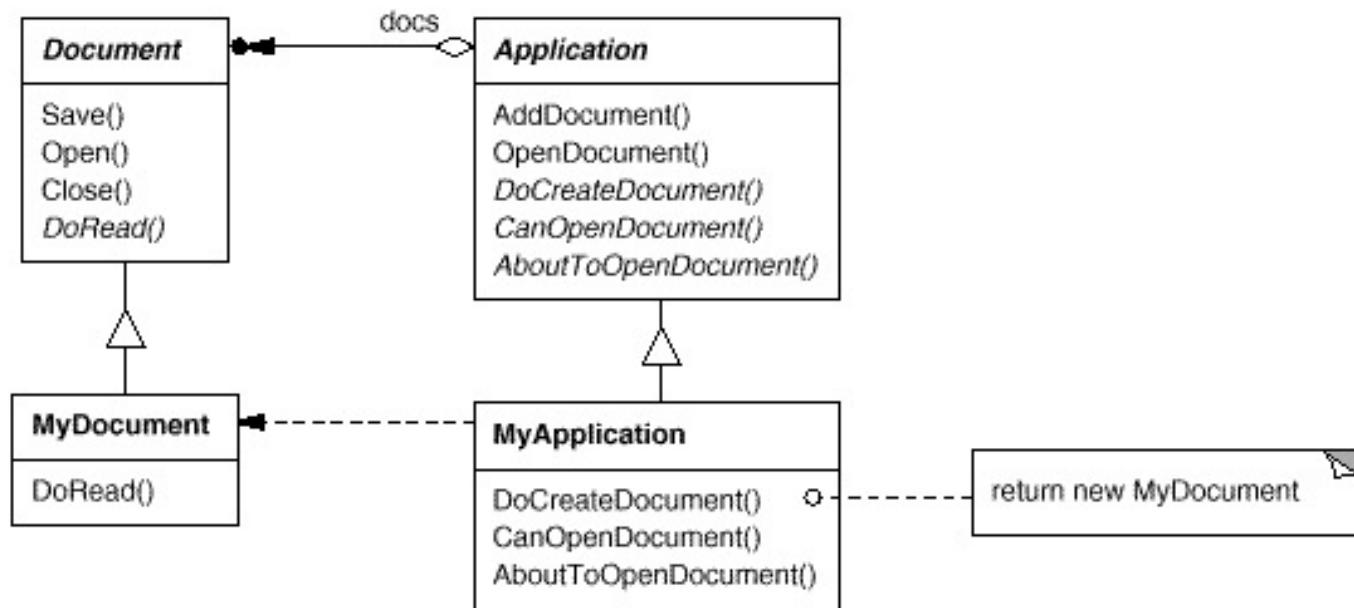
Pattern Template method: intento

- Definire lo *scheletro* di un algoritmo rimandando la definizione di alcuni passi alle sottoclassi.
- Template Method permette alle sottoclassi di ridefinire certe parti di un algoritmo senza modificare la struttura dell'algoritmo stesso

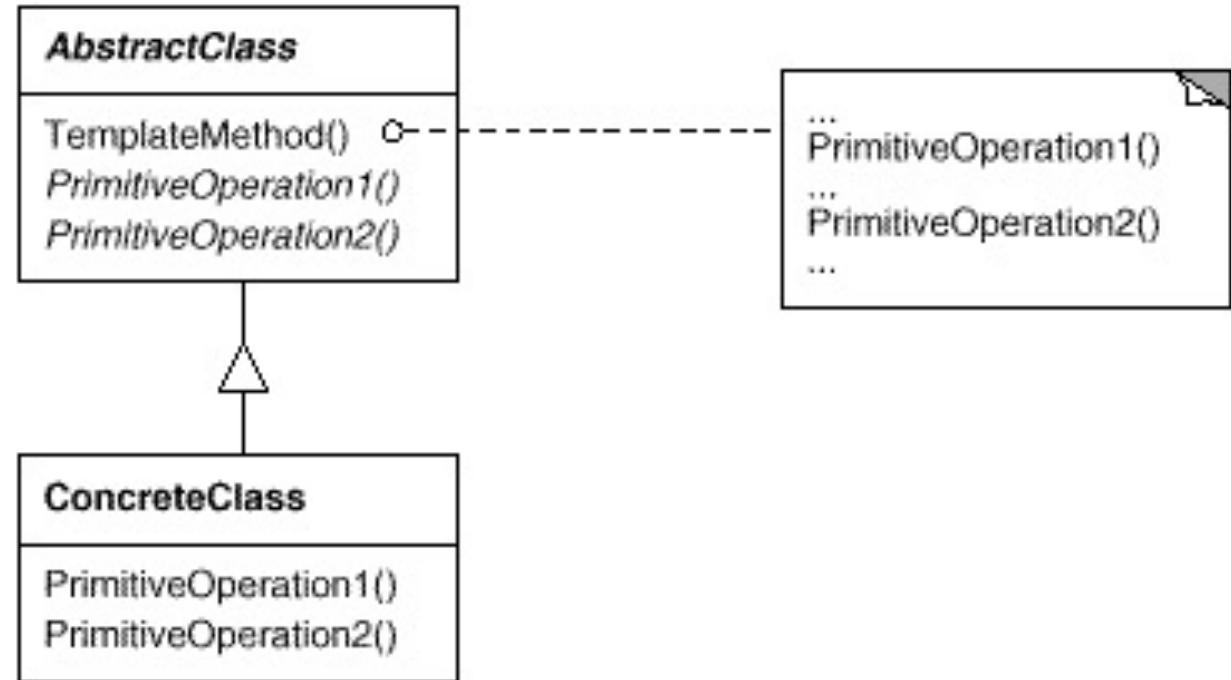


Template method: esempio

- Un framework offre le classi Application e Document.
- La classe Application ha la responsabilità di aprire i documenti memorizzati come file in un formato esterno.
- Un oggetto Document rappresenta l'informazione letta dal file
- Il framework si usa subclassando Application e Document
- Applica sistematicamente il principio dell'Inversione del Controllo



Pattern Template method: struttura



AbstractClass (Application)

- Definisce operazioni astratte primitive che le sottoclassi concrete definiscono per implementare i passi di un algoritmo.
- implementa un Template Method che definisce lo scheletro di un algoritmo; il Template method chiama le operazioni primitive definite in **AbstractClass** o quelle di altri oggetti.

ConcreteClass (MyApplication)

- implementa le operazioni primitive per eseguire i passi specializzati dell'algoritmo

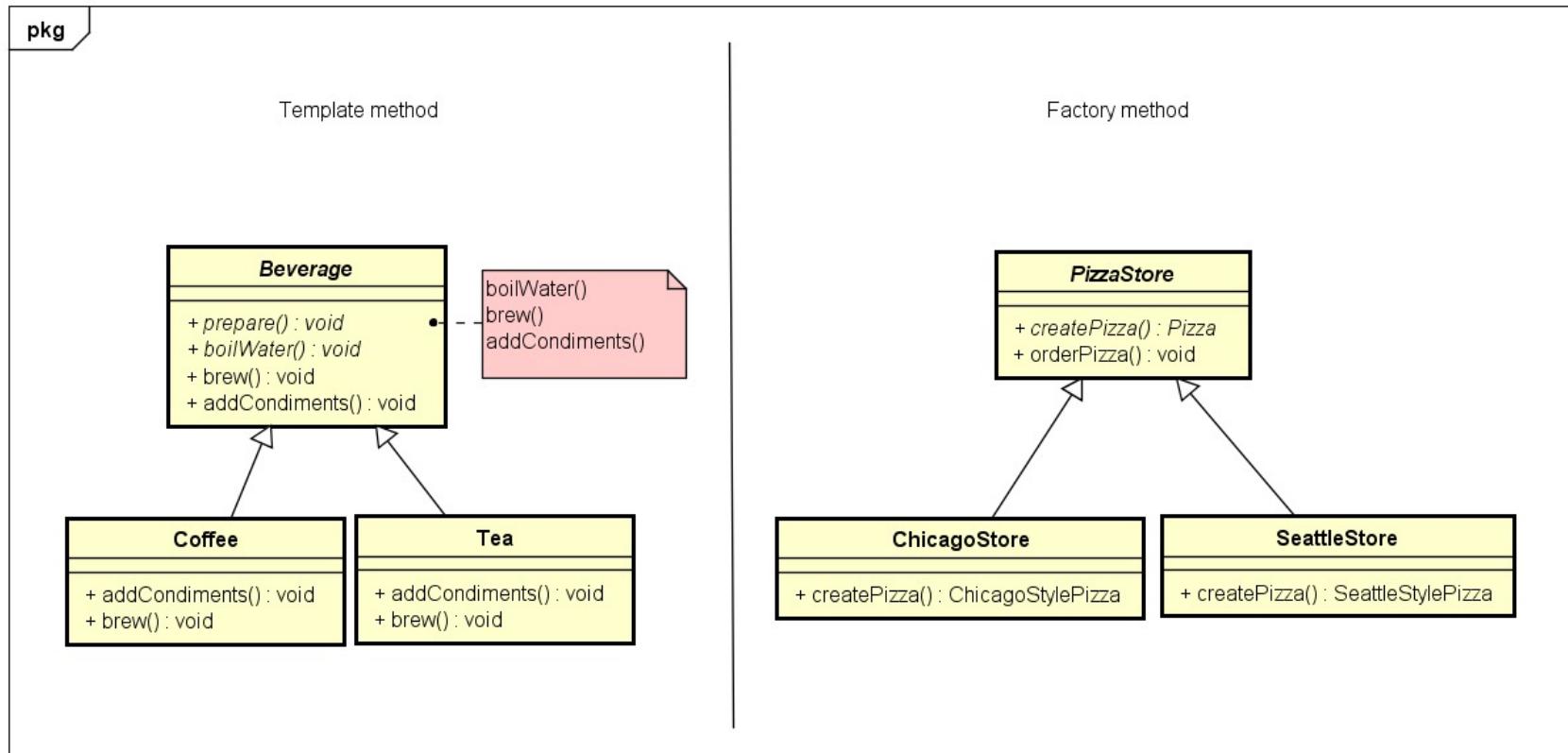
Template method: conseguenze

- Template Methods sono una tecnica importante di riuso del codice.
- Si usano spesso in librerie di classi coese, perché permettono di fattorizzare il comportamento comune.
- I Template Methods danno luogo ad una struttura di controllo invertita che segue il cosiddetto «principio Hollywood», ovvero, «*non chiamarci, ti chiameremo noi*» (Inversione del controllo).
- I Template methods chiamano questi tipi di operazioni:
 - Operazioni concrete (su ConcreteClass o sulle classi clienti);
 - Operazioni concrete AbstractClass (di solito utili per le sottoclassi)
 - Operazioni primitive (cioè astratte);
 - Factory methods (vedi il pattern Factory Method)
 - Operazioni «gancio», che offrono un comportamento default estendibile ove necessario dalle sottoclassi. Un'operazione gancio per default non fa nulla

Da ricordare

- La differenza tra «Factory method» e «Abstract factory» è che il factory method è un unico metodo, invece l'abstract factory è un oggetto.
- Factory method essendo un metodo, può essere ridefinito (overridden) in una sottoclasse, usa l'ereditarietà e si basa su una sottoclasse per gestire l'istanza desiderata
- Invece con Abstract Factory, una classe delega la responsabilità di istanziare un oggetto ad un altro oggetto via composizione
- Il pattern Factory Method pattern si usa tipicamente insieme con una caso speciale del pattern Template Method: Factory Method è in sostanza una specializzazione di Template Method

Factory Method vs Template method



powered by Astah

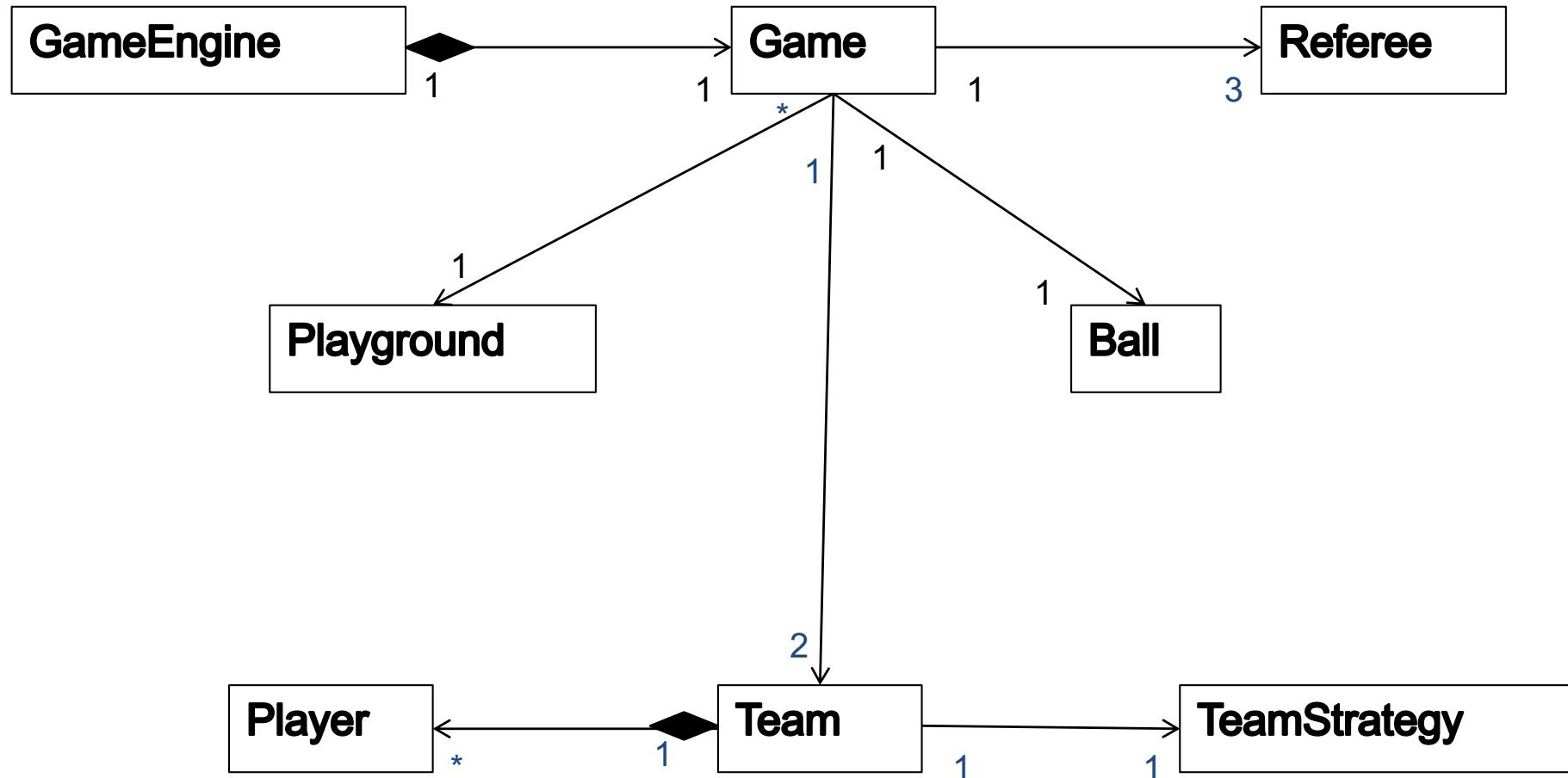
Riassunto: quali pattern GoF abbiamo visto fin qui

- Composite
- Decorator
- Abstract factory
- Bridge
- Iterator
- Visitor
- Flyweight
- Command
- Memento
- Factory method
- Template method

Caso di studio 2: videogioco

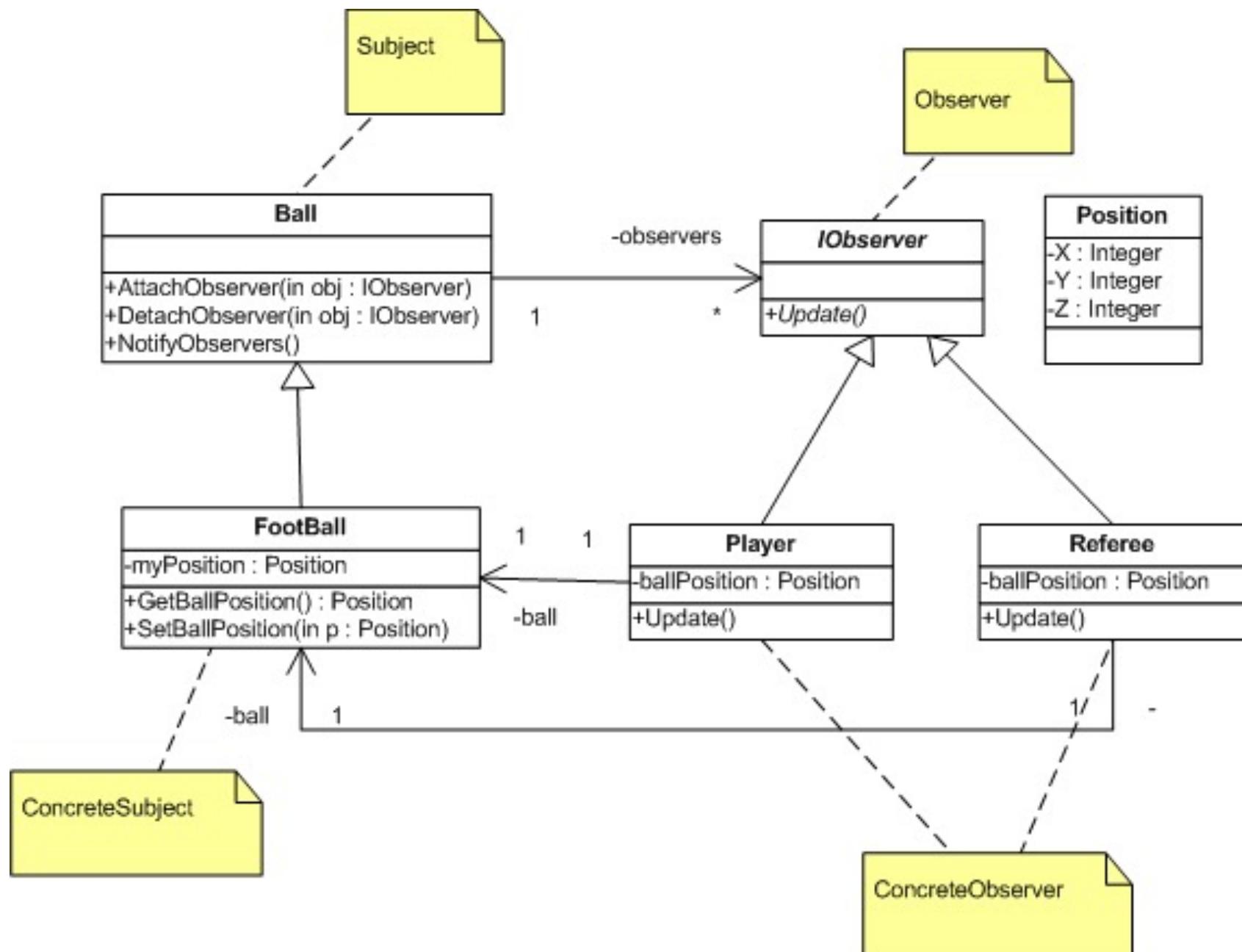
- L'utente usa il videogioco in questa sequenza
 - Attivazione
 - Scelta delle due squadre
 - Scelta dei giocatori di ciascuna squadra (aggiungi/rimuovi)
 - Scegli uno stadio
 - Inizia la partita
- I dati di libreria includono alcuni stadi, alcune squadre, ecc.
 - Oggetto Player
 - Oggetto Team
 - Oggetto Ball, gestita dai Players.
 - Oggetto Playground, lo stadio
 - Oggetto Referee, l'arbitro
- Altri oggetti utili:
 - Oggetto Game, che definisce la partita tra due squadre in uno stadio ecc.
 - Oggetto GameEngine che definisce un torneo di più partite
 - Oggetto TeamStrategy, che decide la strategia di gioco di una squadra

Analisi del dominio

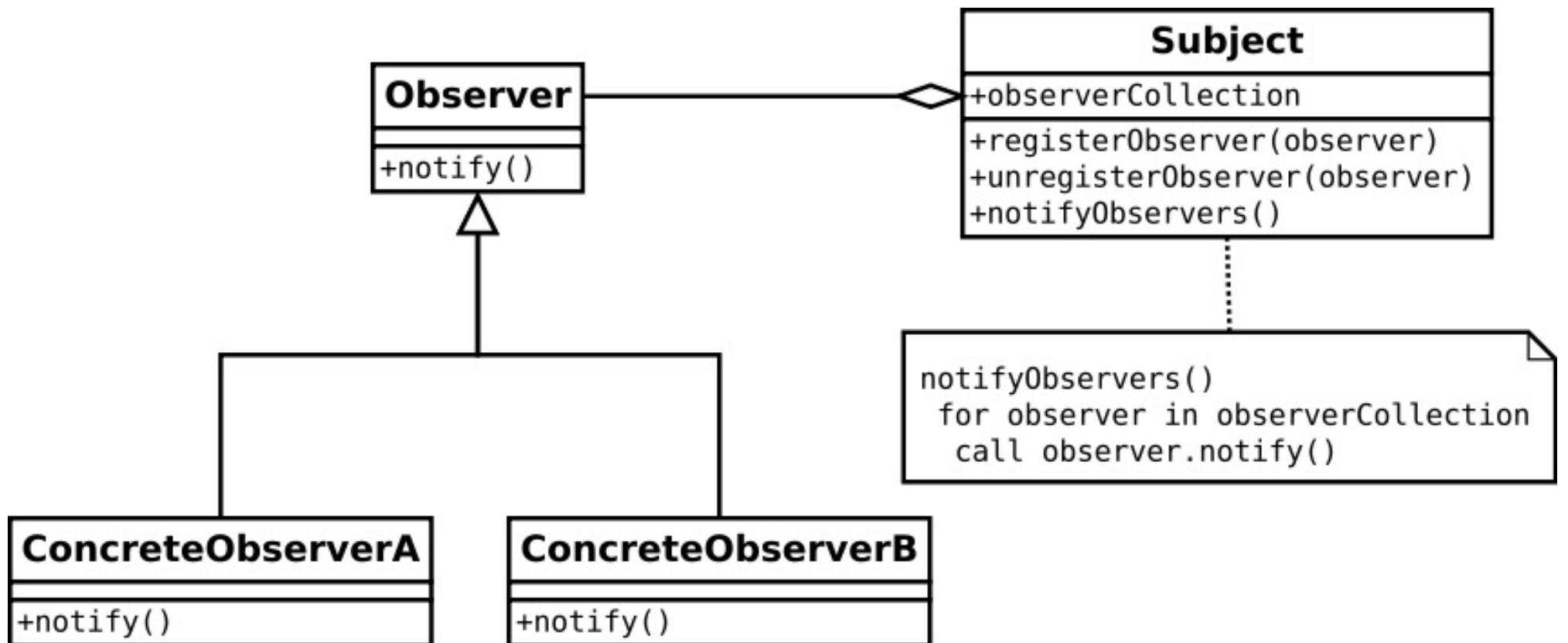


Problema

- “*Quando la palla cambia posizione, occorre avvertire tutti gli oggetti Players e Referee*”
- Problema: “Quando un oggetto (in questo caso la palla) cambia stato, tutti i suoi clienti (in questo caso i giocatori) devono essere avvertiti ed aggiornati automaticamente”
- Possiamo applicare il pattern 'Observer'
- **Pattern Observer** : Definisce una dipendenza uno-a-molti tra oggetti, cosicché quando un oggetto modifica il suo stato tutti i suoi clienti sono avvertiti ed aggiornati automaticamente



Pattern Observer



Pattern: Observer

- Questo pattern gestisce una dipendenza tra oggetti uno-a-molti: quando un oggetto cambia stato, tutti i suoi dipendenti ottengono una notifica e si aggiornano
- Uno o più oggetti (detti *Observers* o *Listeners*) si registrano o vengono registrati come interessati ad un evento creato dall'oggetto osservato (il *Subject*, o *Observable*)
- L'oggetto osservato che crea l'evento gestisce la lista dei suoi *Observers*

Pattern Observer: partecipante *Subject*

- **Subject** (anche detto Observable): classe astratta, interfaccia per gli Observers
 - Possiede la lista degli Observers
 - Contiene i metodi:
 - **Attach** (o addObserver): aggiunge un Observer alla lista
 - **Detach** (o deleteObserver): rimuove un Observer dalla lista
 - **Notify**: quando avviene una modifica di stato avverte gli Observers chiamando il loro metodo *update()*

Pattern Observer: *ConcreteSubject*

- **ConcreteSubject**: la classe che rappresenta lo stato di interesse per gli Observer
 - Notifica tutti gli *Observers* chiamando *Notify* nella sua superclasse (*Subject*)
 - Contiene il metodo:
 - **GetState**: restituisce lo stato di *Subject*

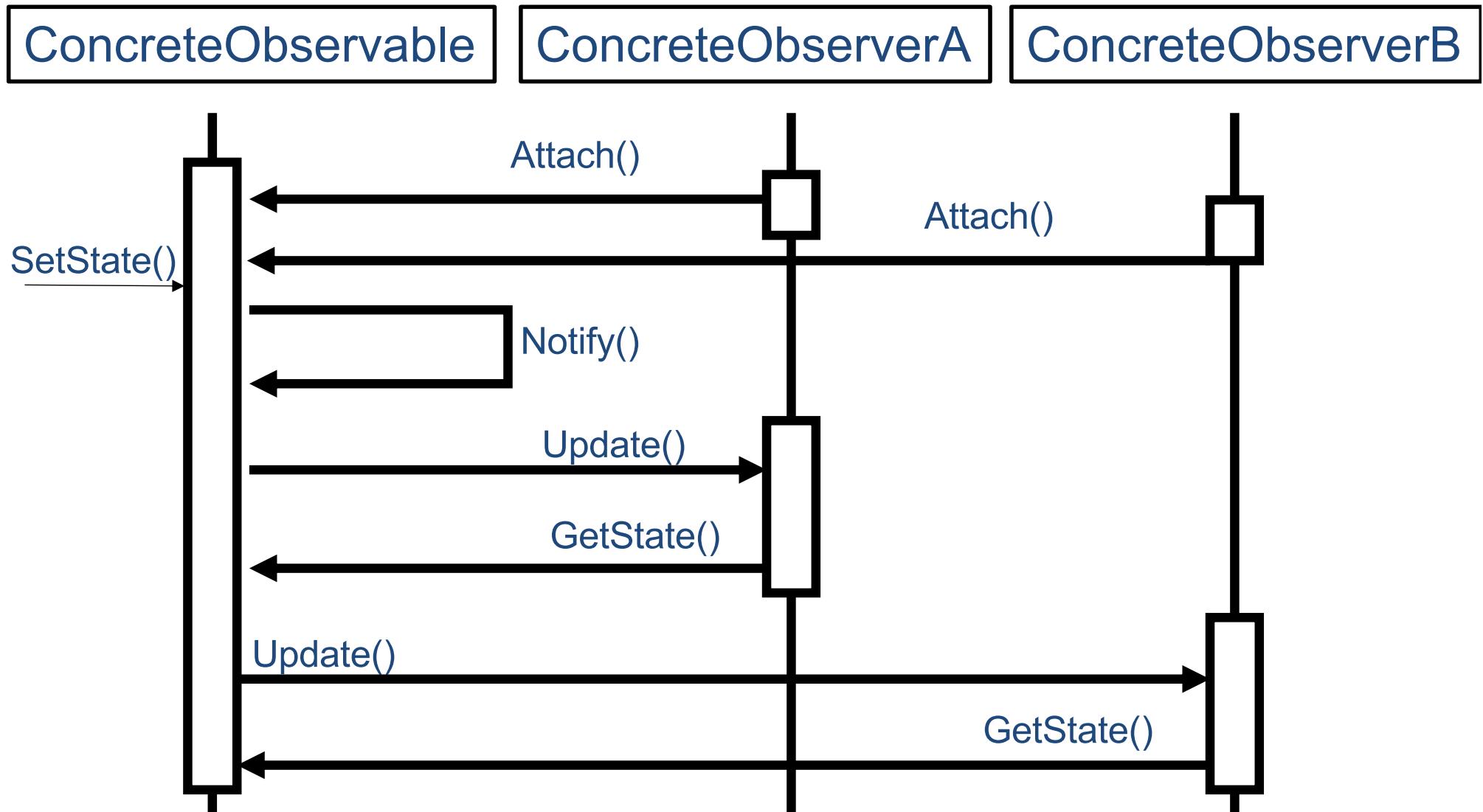
Pattern Observer: *Observer*

- **Observer**: classe che definisce l'interfaccia di tutti gli *Observers* che devono essere avvertiti dei cambiamenti di stato
 - Usata come classe astratta per realizzare *Observers* concreti
 - Contiene il metodo:
 - **Notify**: metodo astratto ridefinito (overridden) dagli *Observers* concreti

Pattern Observer: *concreteobserver*

- **ConcreteObserver**: classe che riferisce *Concrete Subject* per gestire l'evento che modifica lo stato
 - Contiene il metodo:
 - **Notify**: ridefinito nella classe concreta
 - Quando è avvisato dal *Subject*, il *ConcreteObserver* invoca il metodo *GetState* di *Subject* per aggiornare l'informazione che ha sullo stato di *Subject*
 - Quanto l'evento avviene ogni *Observer* riceve una callback che può essere
 - Metodo virtuale *Notify()* di *Observer*
 - Puntatore a funzione (oggetto funzione o "funtore") passato come argomento al metodo di registrazione ascoltatori
 - Ogni *ConcreteObserver* implementa il metodo *Notify* e di conseguenza implementa un suo comportamento specifico quando arriva la notifica

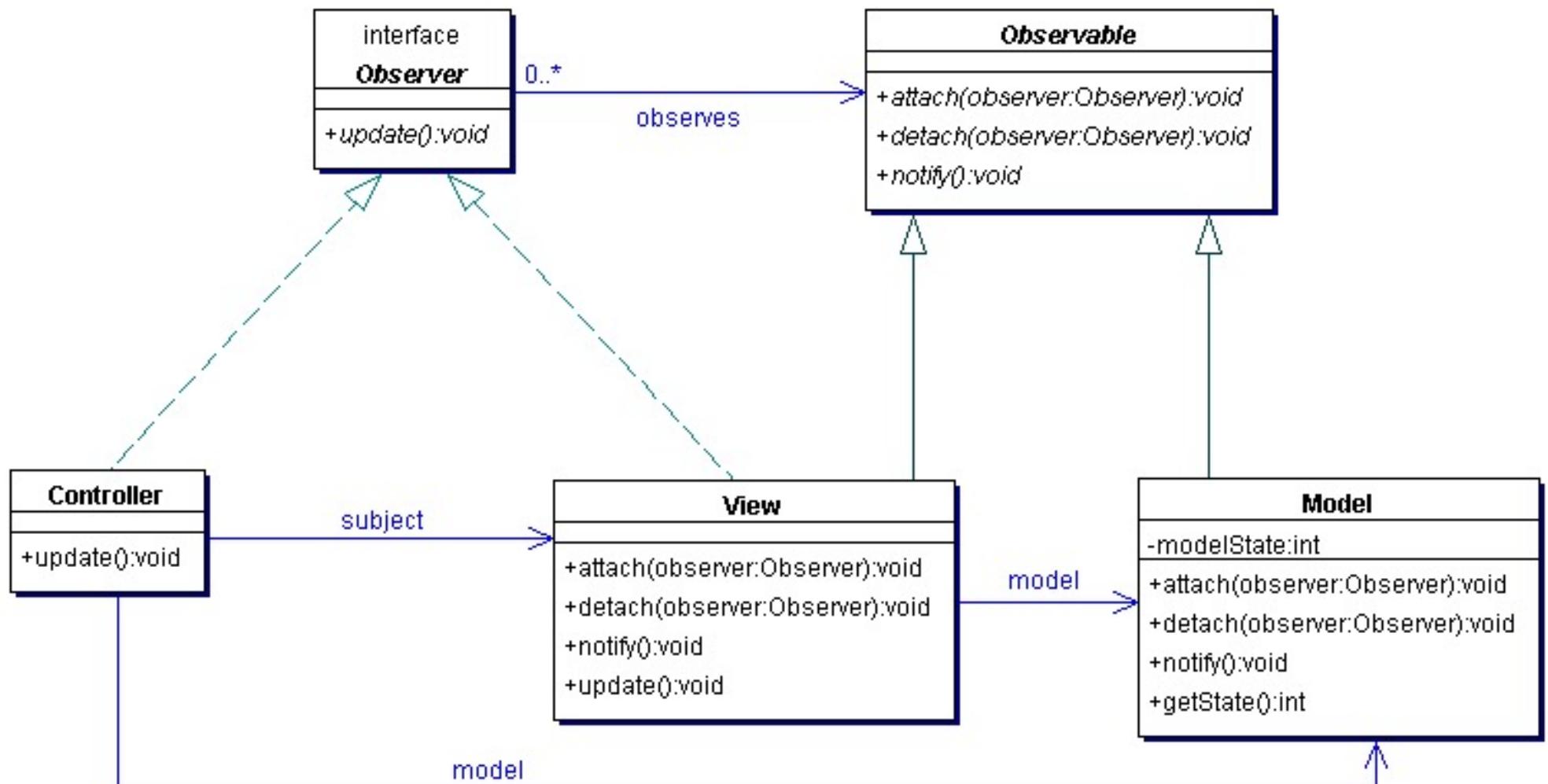
Pattern observer: diagramma di sequenza



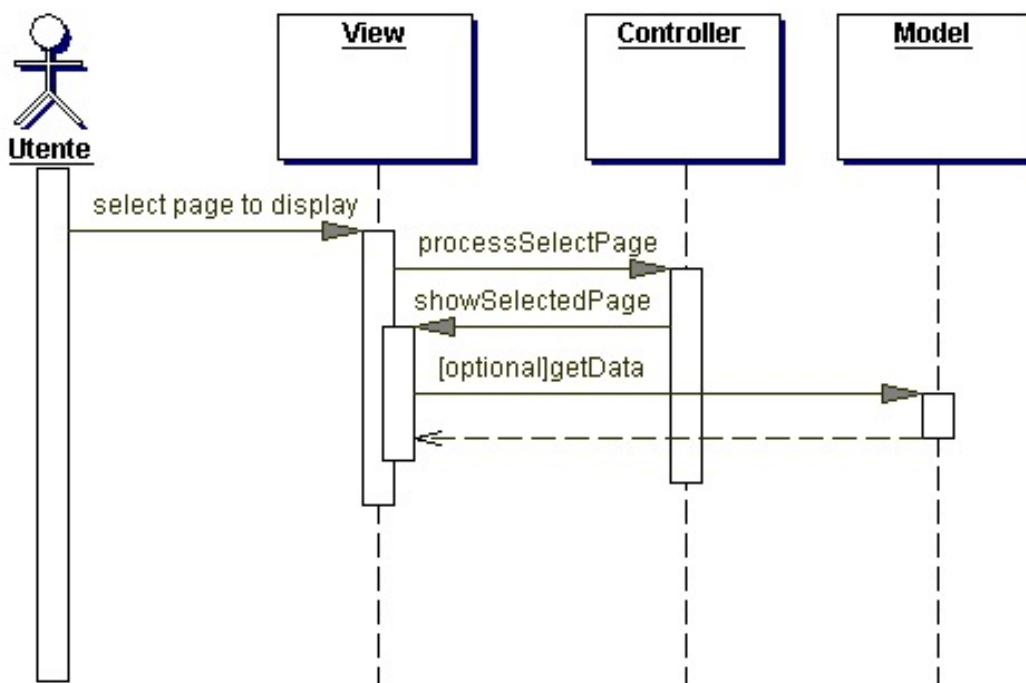
Pattern Observer: uso

- **Usi tipici:**
 - Reazione ad un evento esterno (es. azione utente)
 - Reazione a modifiche del valore di una proprietà di un oggetto
 - Mailing list: ogni volta che qualcuno spedisce un msg tutti lo ricevono
- **Spesso associato al pattern architettonale Model-View-Controller (MVC)**
 - Il pattern disaccoppia Model da View: le modifiche al modello (logica applicativa) sono notificate agli observer che sono le viste (output)

Model View Controller: un pattern architetture



MVC: seq diagram



La View non decide quale sarà la schermata richiesta: delega la decisione al Controller (che se usa il pattern Strategy può modificarsi a runtime); View si occupa della costruzione e della presentazione all'utente della schermata; Controller e View sono observers di Model

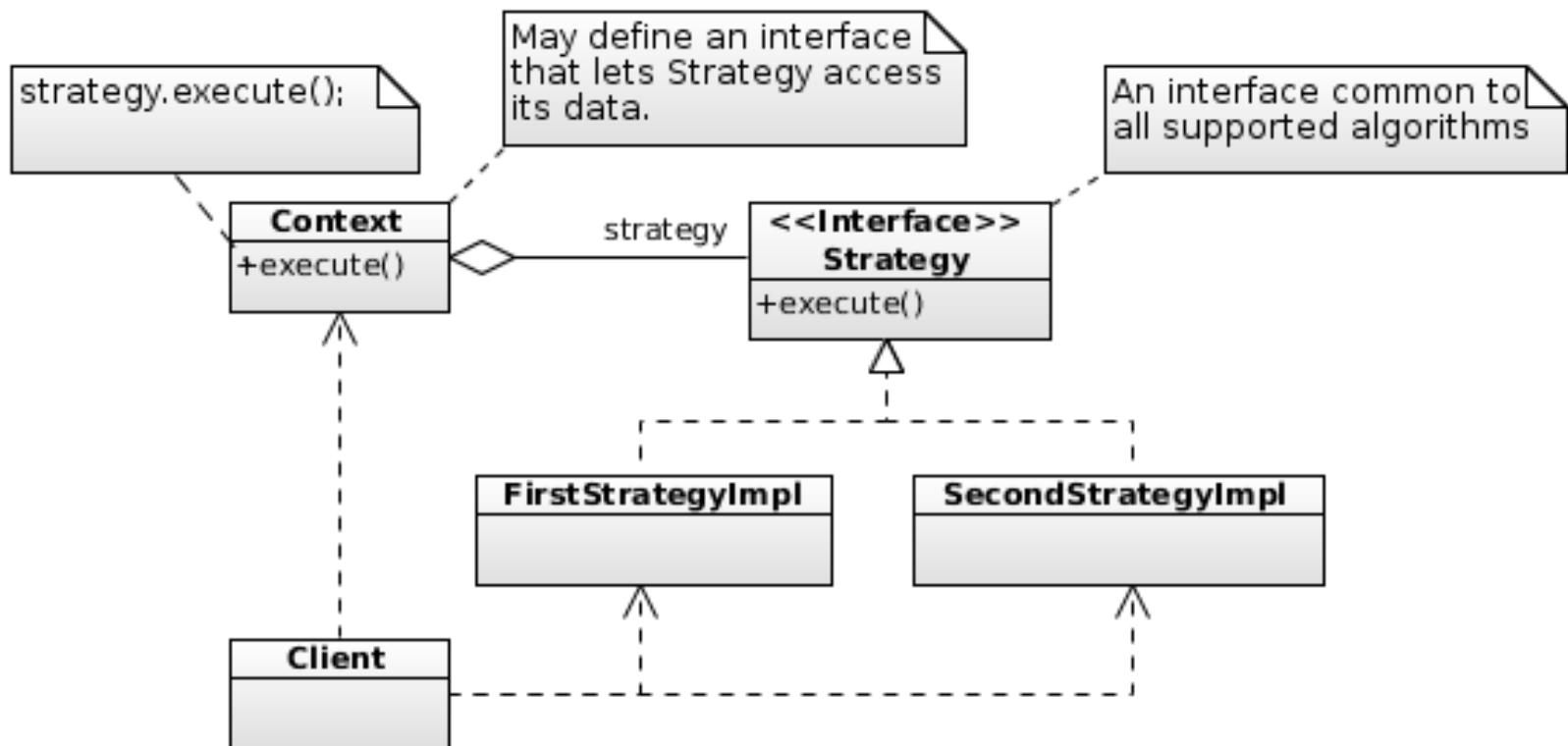
Problema

"durante una partita il giocatore può modificare la strategia della sua squadra (es. da Attacco a Difesa)"

Il pattern 'Strategy' risolve questo problema di design:

- *intento: "occorre che un algoritmo (TeamStrategy) possa essere modificato in modo indipendente da chi lo usa (nel nostro caso, Team)"*
- **Pattern Strategy:** definisce una famiglia di algoritmi, li incapsula, e li rende interscambiabili
Strategy disaccoppia gli algoritmi dai clienti che vogliono usarli dinamicamente

Pattern Strategy



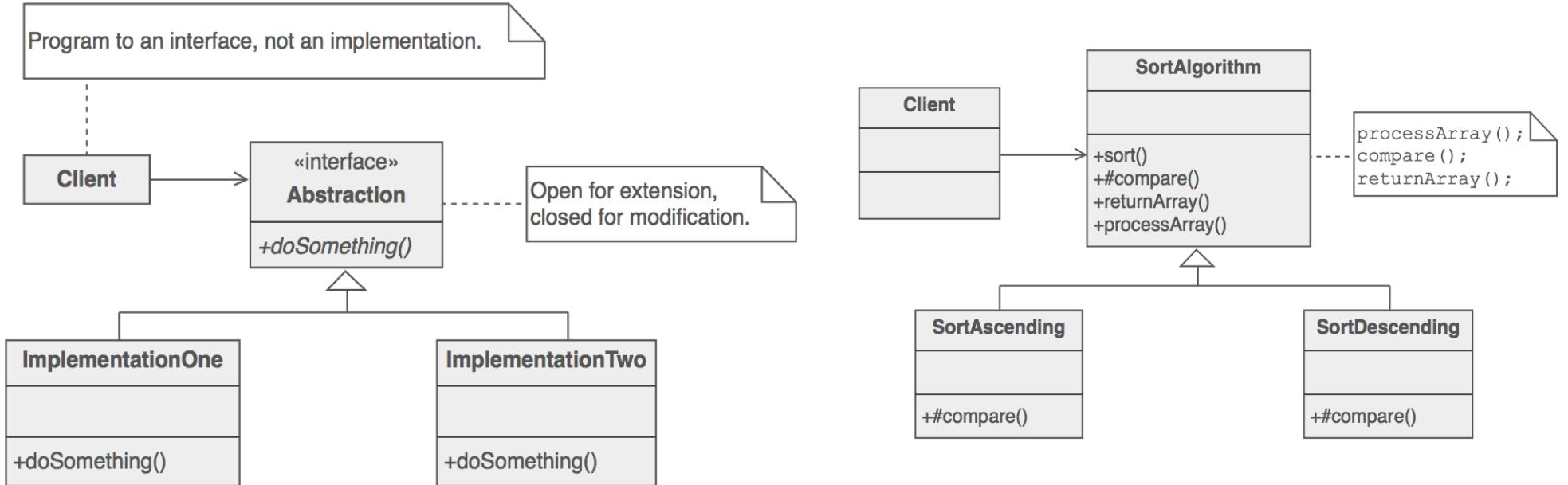
Pattern: Strategy

- Questo pattern permette che un oggetto client possa usare indifferentemente uno o l'altro algoritmo (strategy) di una stessa famiglia
- È utile ove sia necessario modificare il comportamento a runtime di una classe

Strategy vs Bridge

- Il diagramma delle classi di Strategy è simile a quello di Bridge
- Tuttavia,
 - il loro *intento* è diverso: *Strategy* si occupa di comportamento, *Bridge* si occupa di struttura
 - L'accoppiamento tra contesto e strategie in *Strategy* è più forte dell'accoppiamento tra astrazione e implementazione in *Bridge*

Strategy vs Template method



Strategy: risponde al principio Open-Closed: le classi derivate nascondono le implementazioni alternative (usa delegation, perché modifica specifici oggetti)

Template method: simile a Strategy, ma usa l'ereditarietà e modifica l'intera classe. Nota: Factory method specializza Template method

Pattern Strategy: esempio

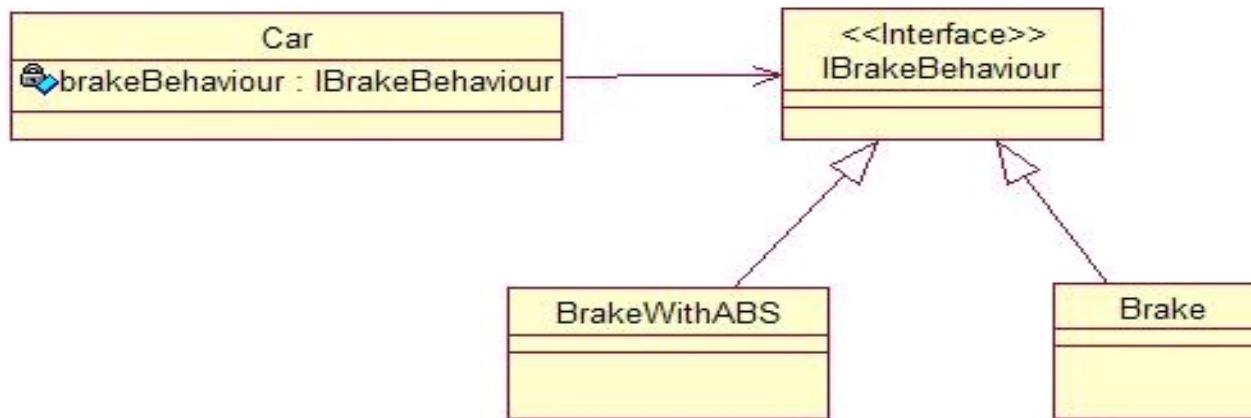
Esempio: Si consideri una classe Auto

- Due possibili comportamenti di Auto sono “frena” e “accelera”
- Siccome tali comportamenti saranno usati spesso in diversi modelli in modo diverso, è normale usare le sottoclassi per realizzarli
- Questo approccio è debole, perché bisognerà cablare “accelera” e “frena” in ogni nuova classe derivata da Auto
 - Duplicazione di codice
 - Solo il codice definisce davvero tali comportamenti, e quindi occorrerà esaminarlo caso per caso

Pattern Strategy

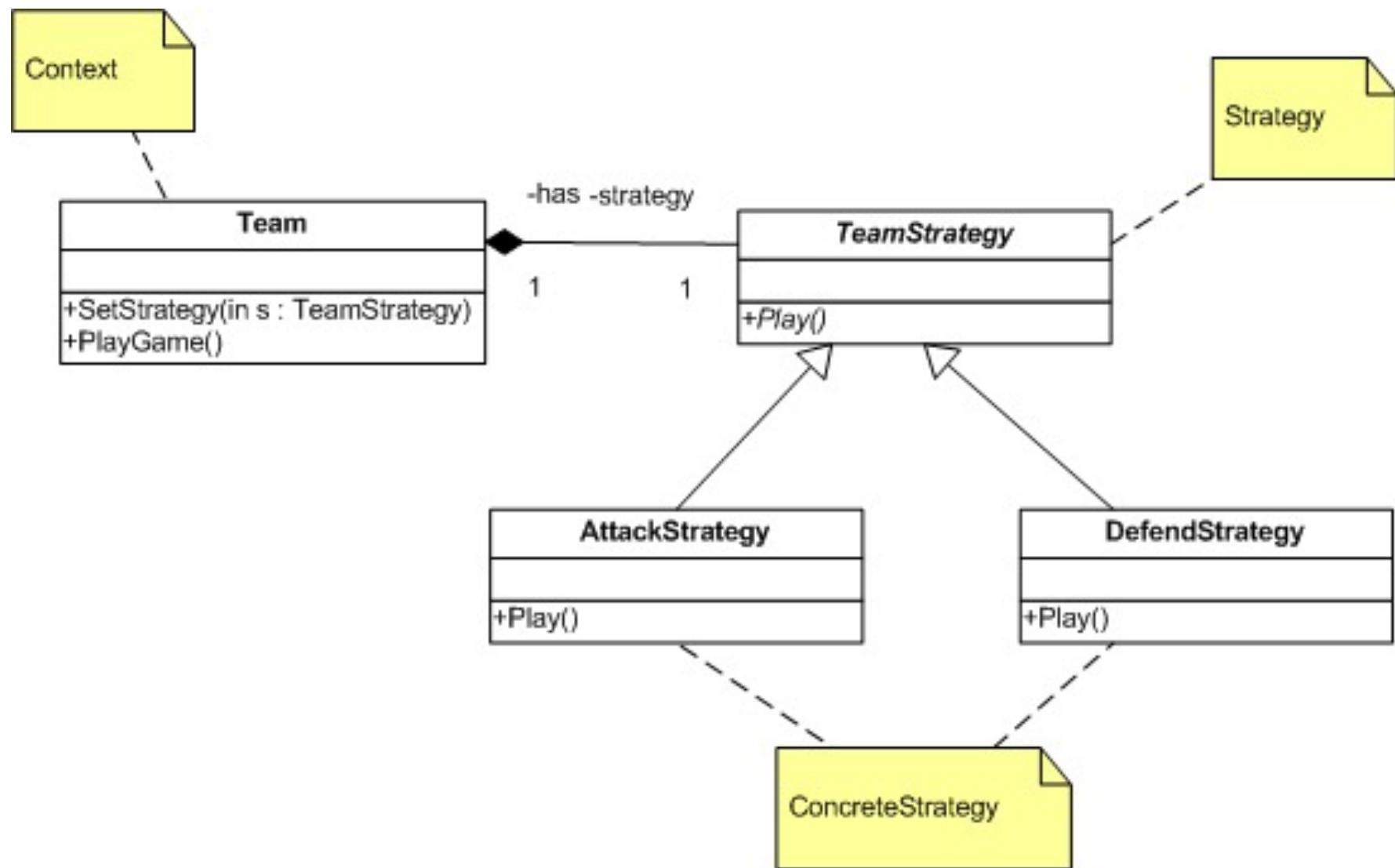
- Il comportamento “frena” di Auto si può modificare da *BrakeWithABS()* a *Brake()* modificando l’attributo *brakeBehavior* :

```
brakeBehavior = new Brake();
```



Strategy: conseguenze

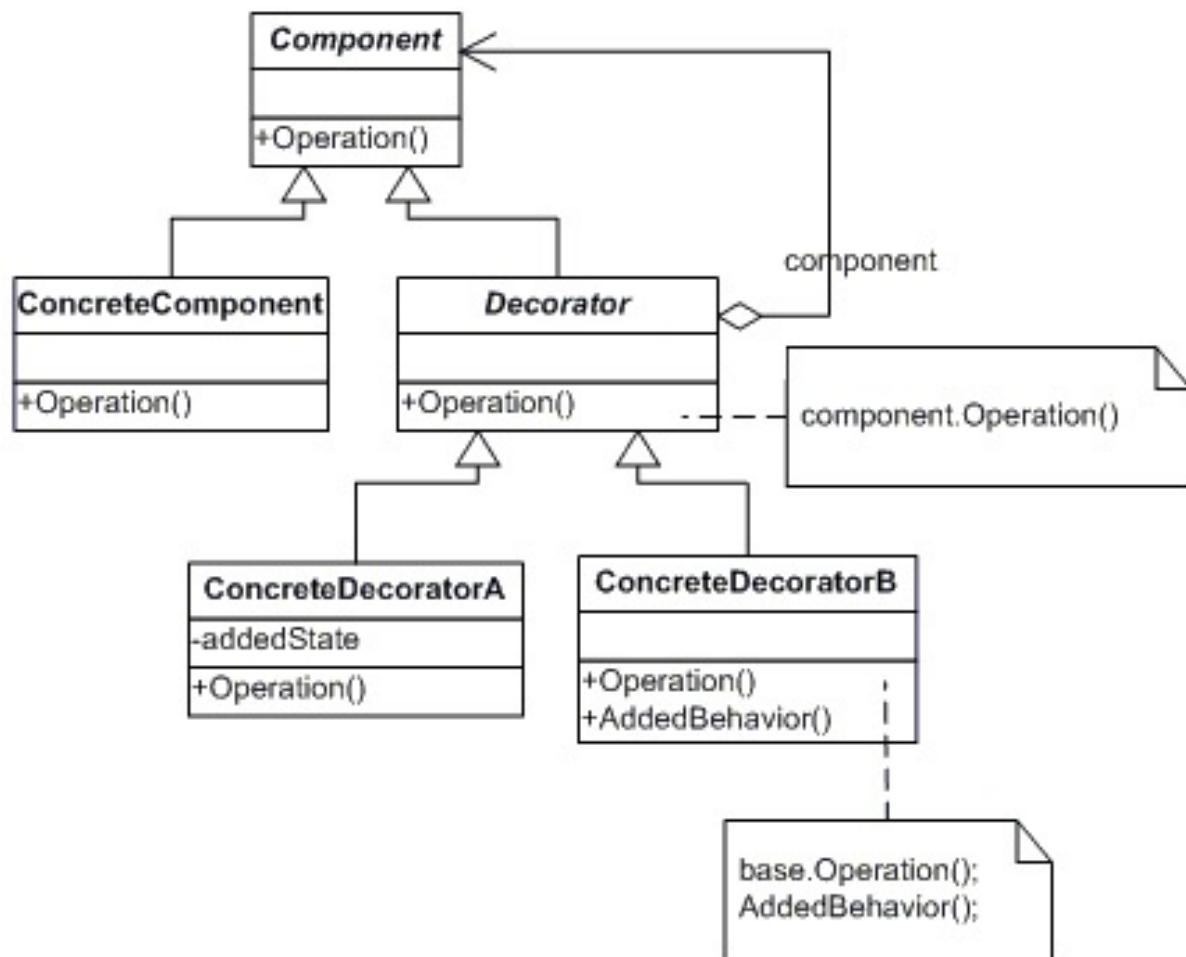
- *Strategy* usa la composizione invece dell'ereditarietà: i comportamenti di una classe non dovrebbero essere ereditati ma piuttosto encapsulati usando la dichiarazione di interfaccia
- I comportamenti vengono definiti da interfacce separate e da classi specifiche che implementano tali interfacce
 - Disaccoppiamento tra il comportamento e la classe che lo usa
 - Comportamento modificabile senza modificare le classi che lo usano
 - Le classi possono mutare comportamento modificando la specifica implementazione che usano senza richiedere alcuna modifica significativa del codice.
 - I comportamenti possono mutare sia a tempo di esecuzione che a tempo di design

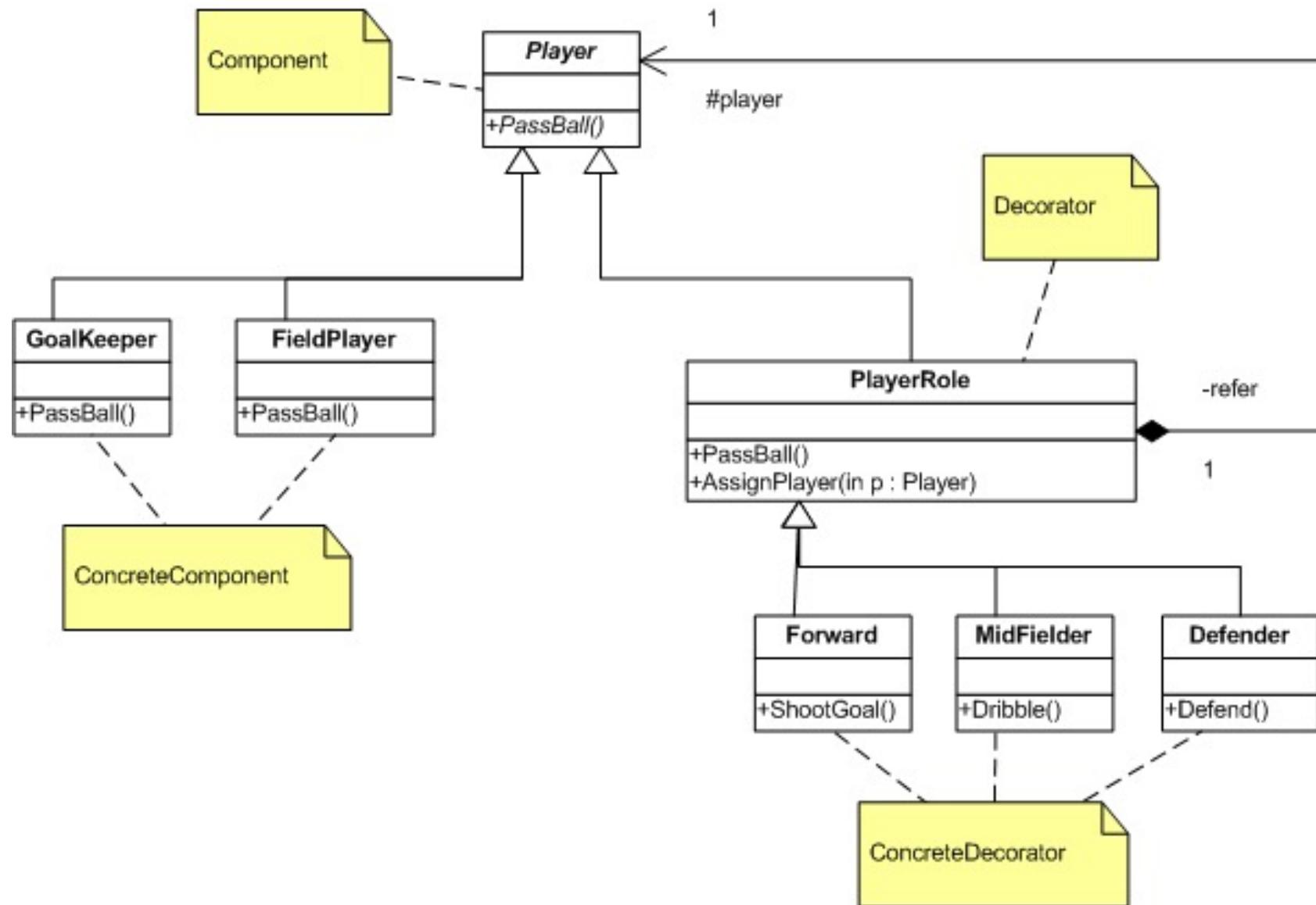


Problema

- *"Un giocatore di una squadra ha un ruolo statico (es. Portiere) ma anche ruoli dinamici assegnabili a runtime, come attaccante, difensore, ecc."*
- *intento: "Vogliamo assegnare dinamicamente responsabilità addizionali (come Attaccante, Difensore ecc.) ad un oggetto (nel nostro caso Player), senza usare la specializzazione (subclassing)"*
- **Pattern Decorator:** Aggiunge dinamicamente nuove responsabilità ad un oggetto
- I Decorator estendono le funzionalità senza bisogno di usare la specializzazione (subclassing)

Decorator

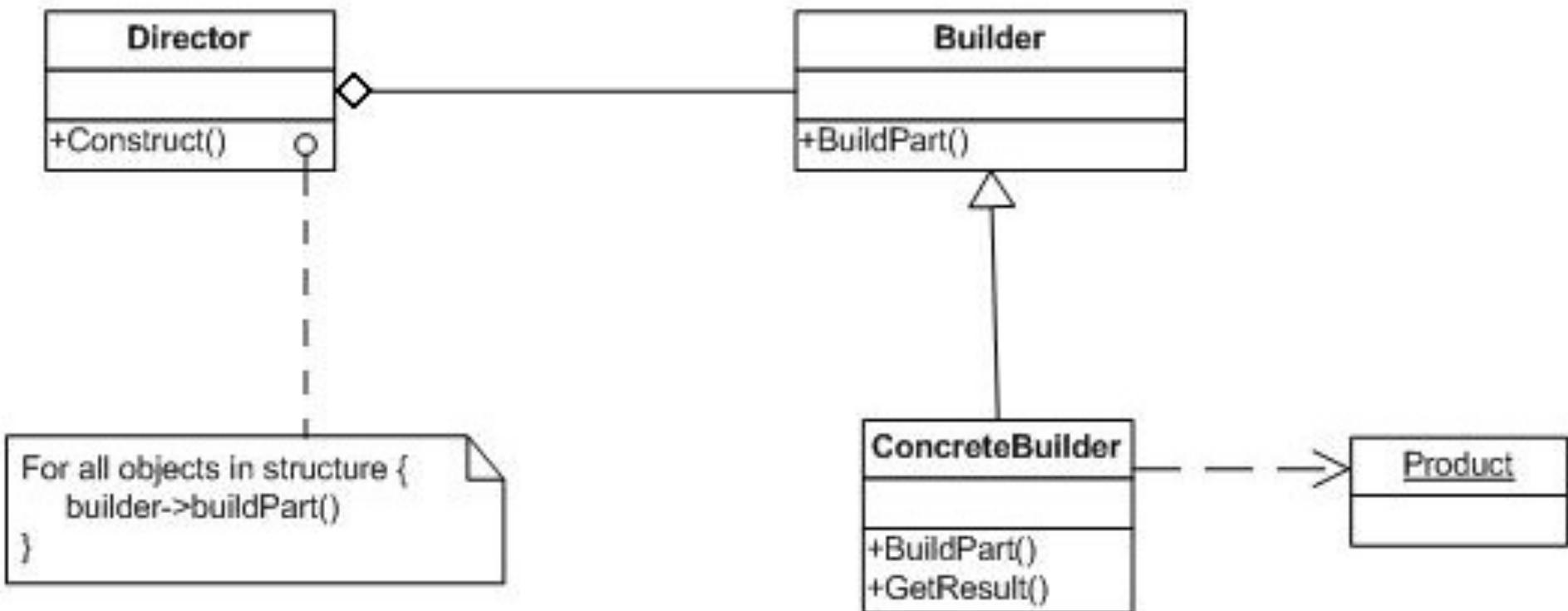




Problema

- *"Ogni stadio include le tribune, il campo, il pubblico, ecc., ed ogni stadio ha aspetto diverso."*
- *Intento:* "Vogliamo separare la costruzione di un oggetto (stadio) dalla sua rappresentazione (aspetto dello stadio) e inoltre vogliamo usare lo stesso processo costruttivo per creare diverse rappresentazioni."
- **Pattern Builder:** Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo costruttivo possa creare rappresentazioni diverse

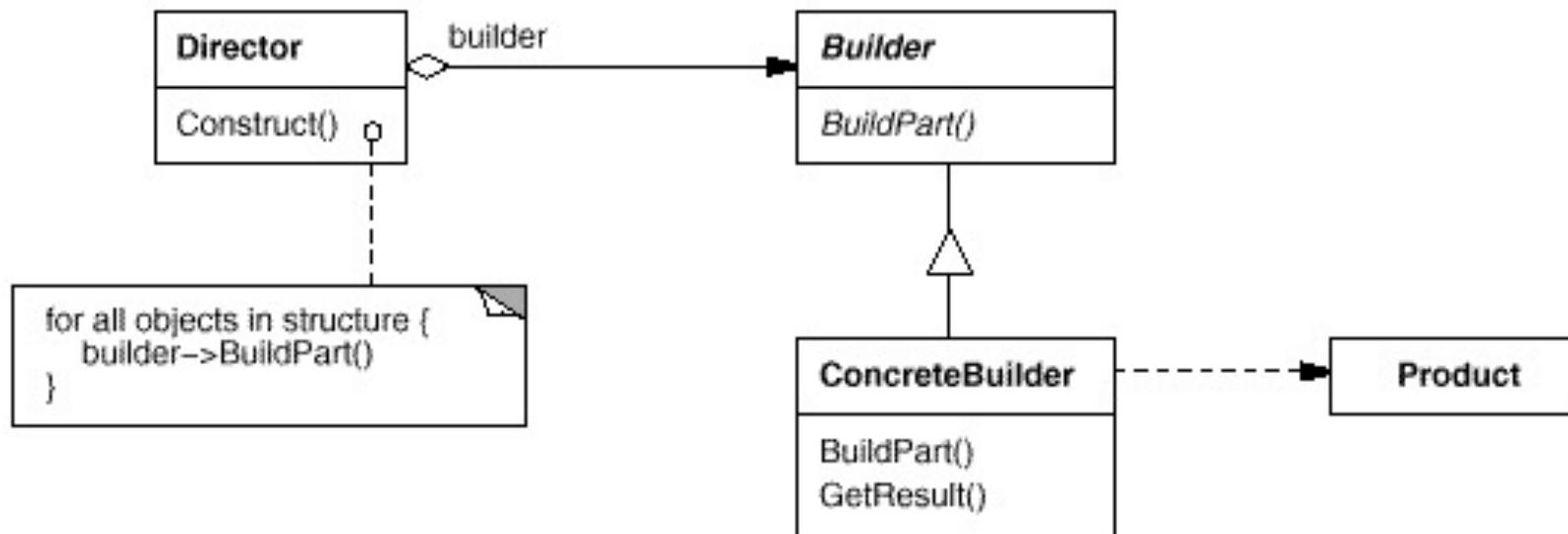
Builder



Pattern Builder

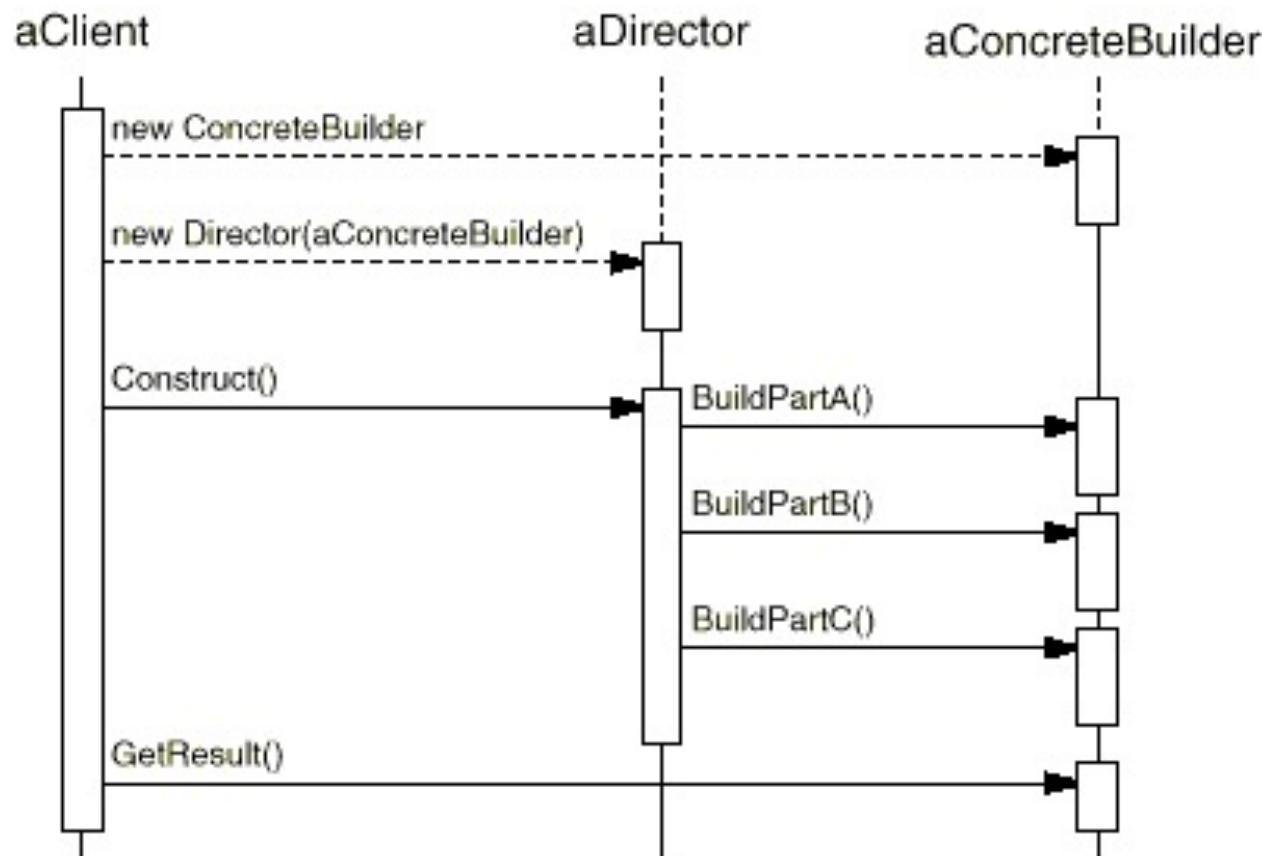
- Può usare uno degli altri pattern creazionali per implementare i diversi componenti da costruire
- Costruisce un oggetto complesso passo dopo passo

Pattern Builder: struttura



- **Builder:** specifica un'interfaccia astratta per creare parti di un oggetto Prodotto.
- **ConcreteBuilder**
 - Costruisce e assembla parti del prodotto implementando l'interfaccia Builder.
 - Definisce e tiene traccia della rappresentazione che crea.
 - Fornisce un interfaccia per ottenere il prodotto.
- **Director:** costruisce un oggetto usando l'interfaccia Builder.
- **Product**
 - Rappresenta l'oggetto complesso in costruzione. ConcreteBuilder costruisce la rappresentazione interna del prodotto e definisce il processo di assemblaggio.
 - Include le classi che definiscono le parti costituenti, comprese le interfacce per assemblare le parti nel risultato finale.

Pattern Builder: comportamento



Builder: uso

- Spesso si usa per costruire un oggetto definito col pattern Composite
- Il design può iniziare con *Factory Method*
 - Meno complesso, più customizzabile, ma proliferazione di sottoclassi
- Il design può evolvere verso *Abstract Factory*, *Prototype*, o *Builder*
 - Più flessibili e complessi man mano che il progettista scopre dove c'è più bisogno di flessibilità

Esercizio



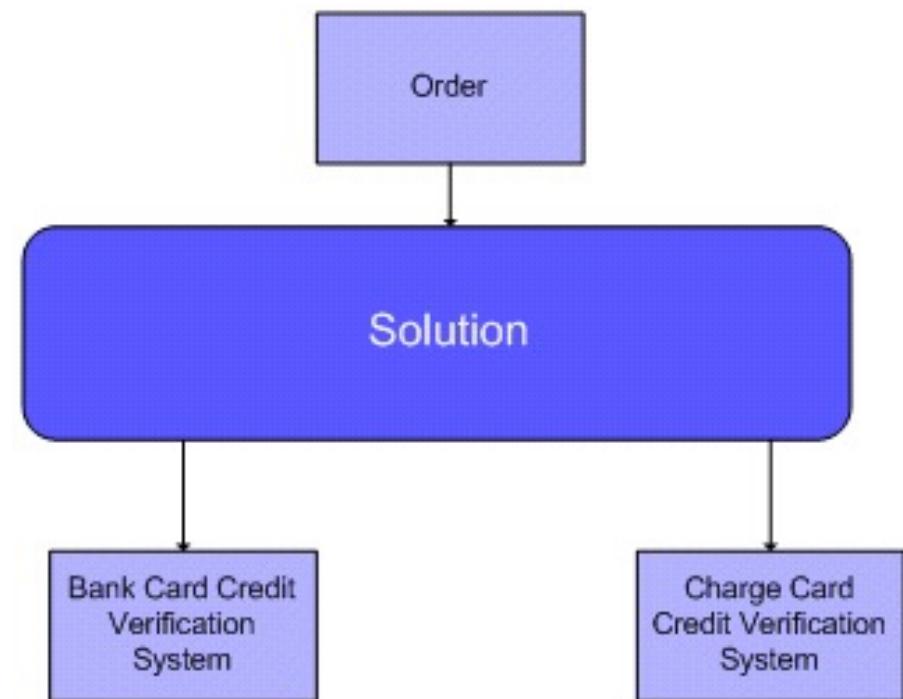
Applicare Builder al problema di costruire lo stadio (playground) della partita di calcio

Pattern nuovi visti

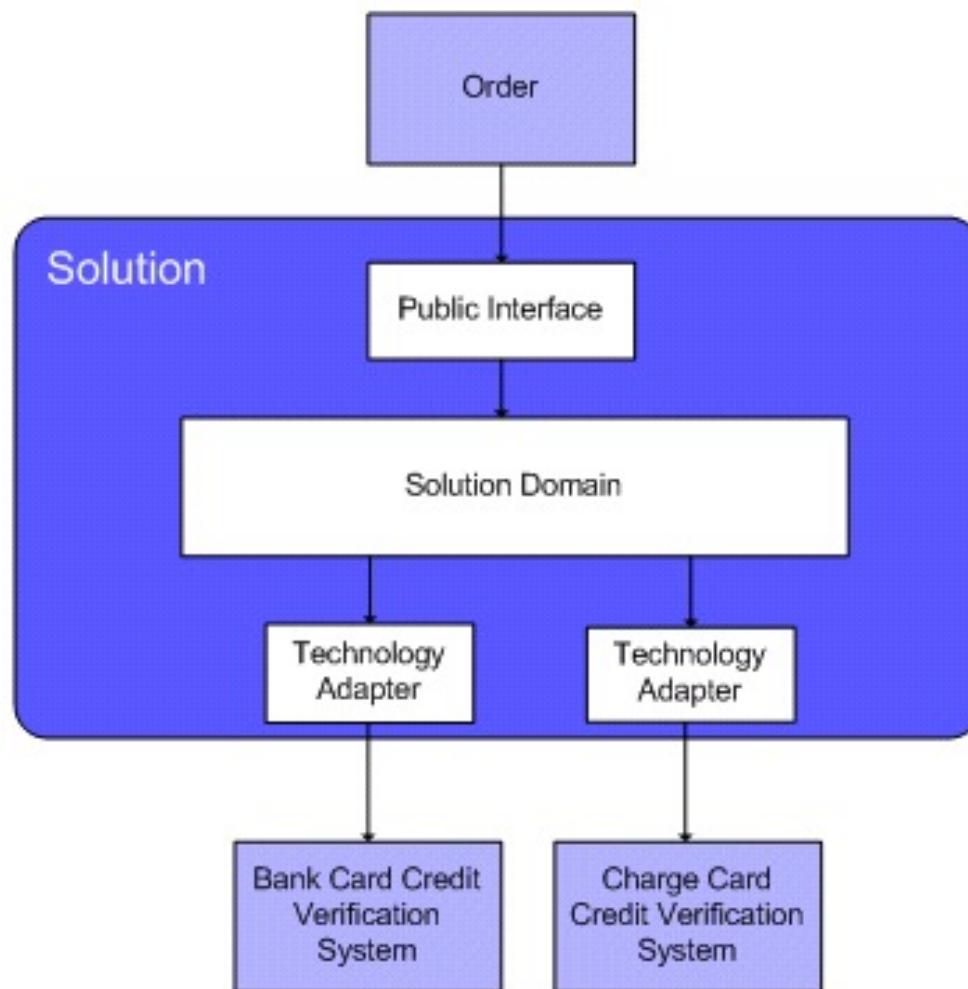
- Observer
- Strategy
- Builder

Caso 3: sistema legacy

- Sistema di verifica pagamenti per un sito commerciale
- Il sistema accetta un ordine di pagamento e lo verifica o col sottosistema Bank Card Verification oppure col sottosistema Charge Card Verification



Struttura della soluzione



Pattern Adapter

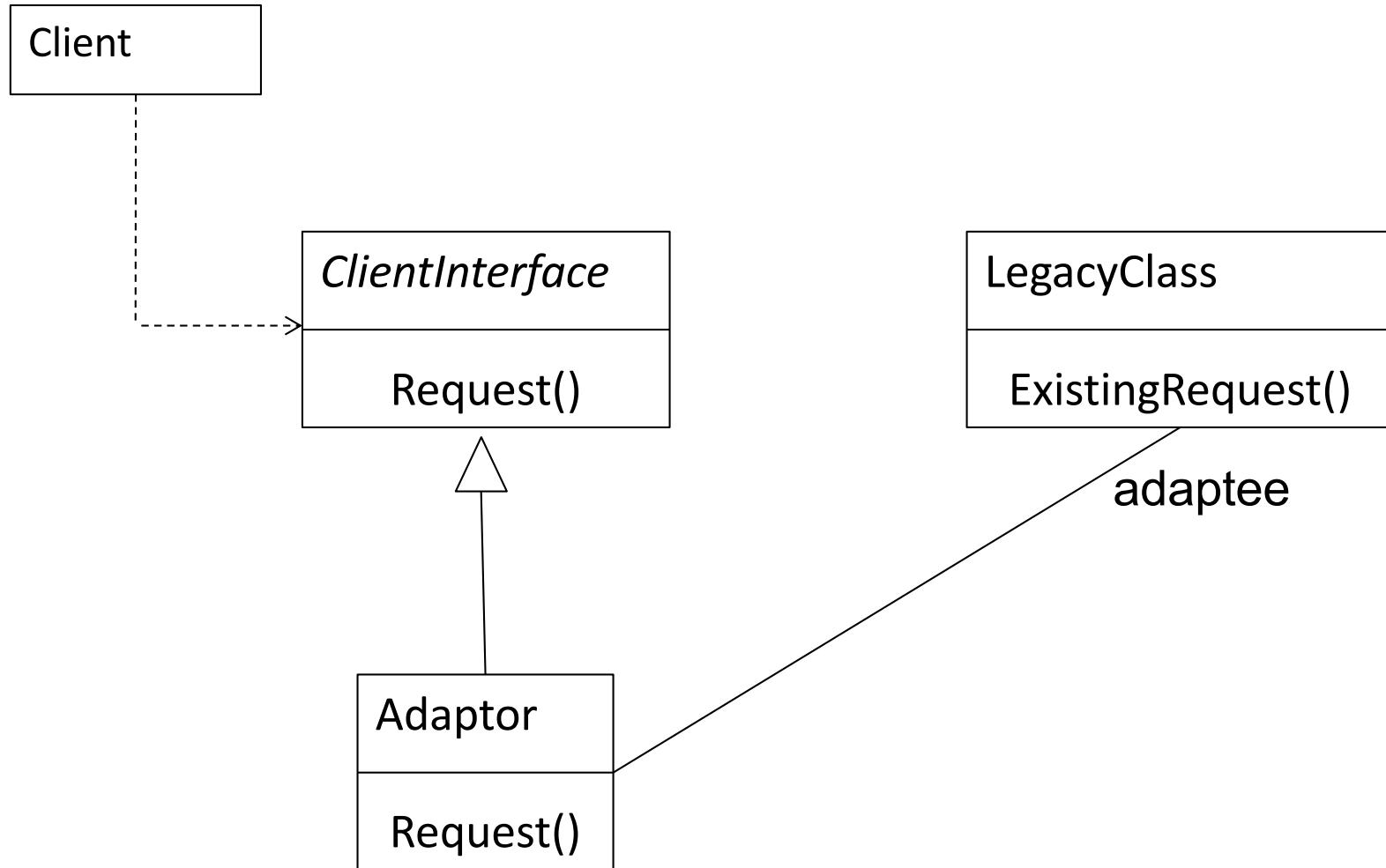
- Traduce l'interfaccia di una classe in una interfaccia compatibile con le attese dei clienti
- L'adattatore che riceve un'invocazione di un suo metodo la trasforma in una chiamata ad un metodo della classe originale adattata
- Esistono due “gusti” di questo pattern:
 - L'adapter a livello di oggetto contiene un'istanza della classe che adatta
 - L'adapter a livello di classe usa interfacce polimorfe

Scopo di Adapter

- Converte l'interfaccia di una classe nell'interfaccia attesa dai clienti della classe
- Permette a classi non progettate per lavorare assieme di diventare compatibili anche se le interfacce non sono compatibili

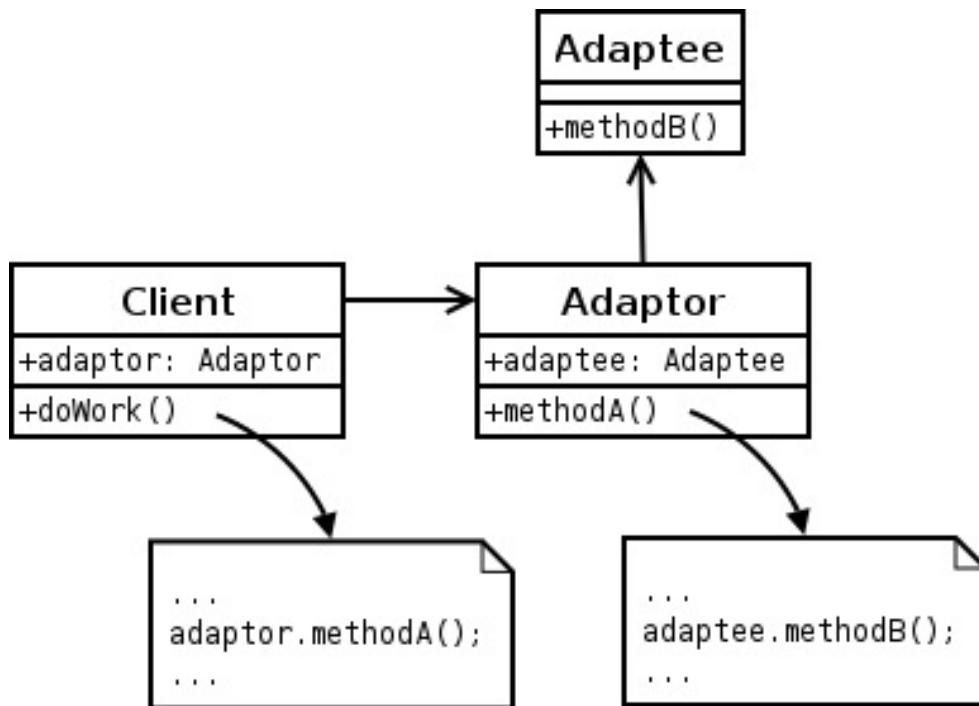


Adapter

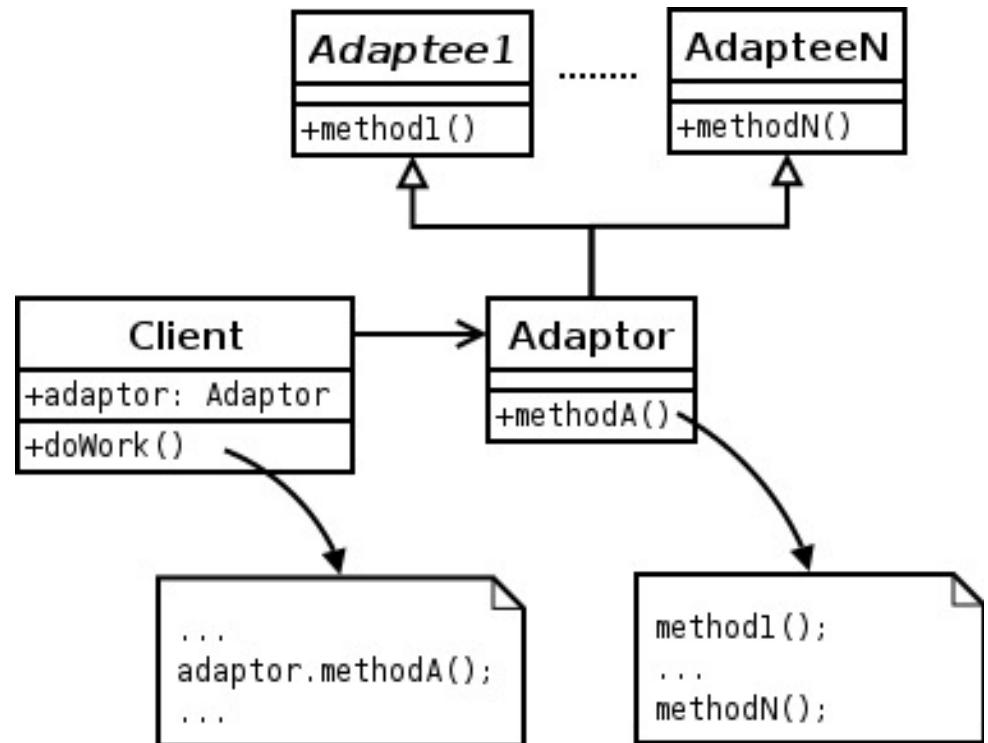


Pattern Adapter

Un adattatore (Adapter) aiuta due interfacce incompatibili a lavorare assieme

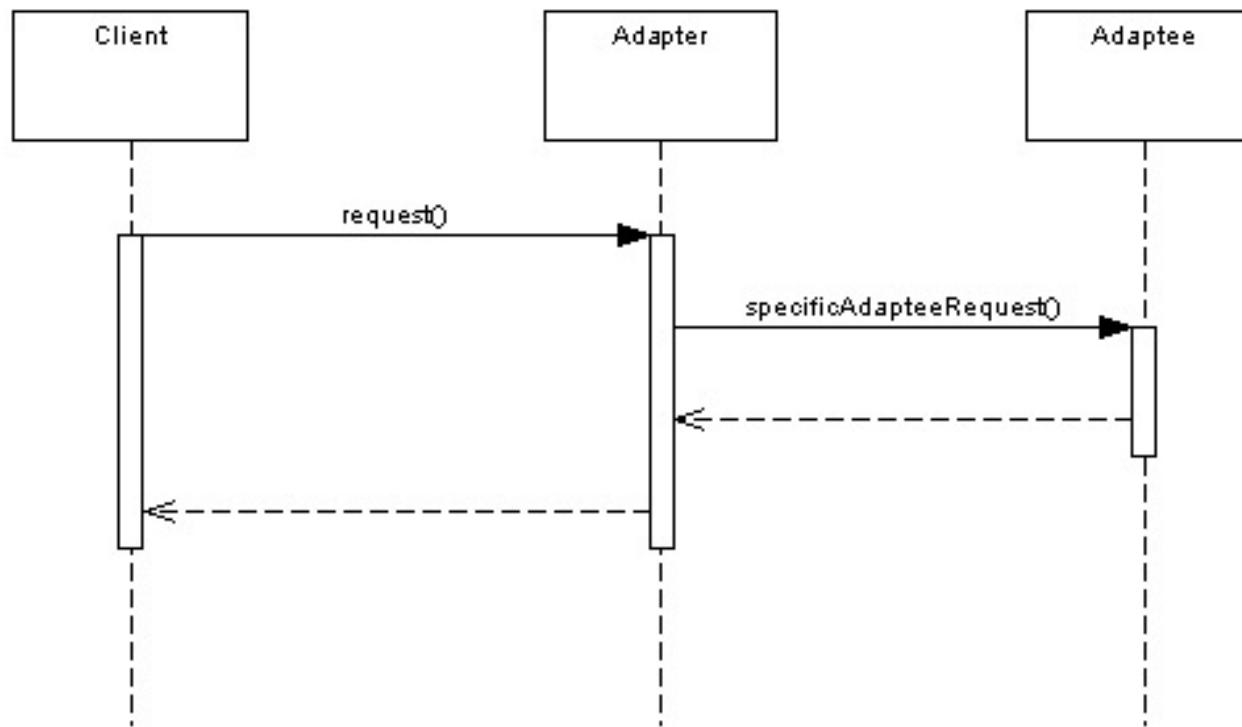


Adattare un oggetto



Adattare più classi

Adapter: comportamento



Adapter – Scopo

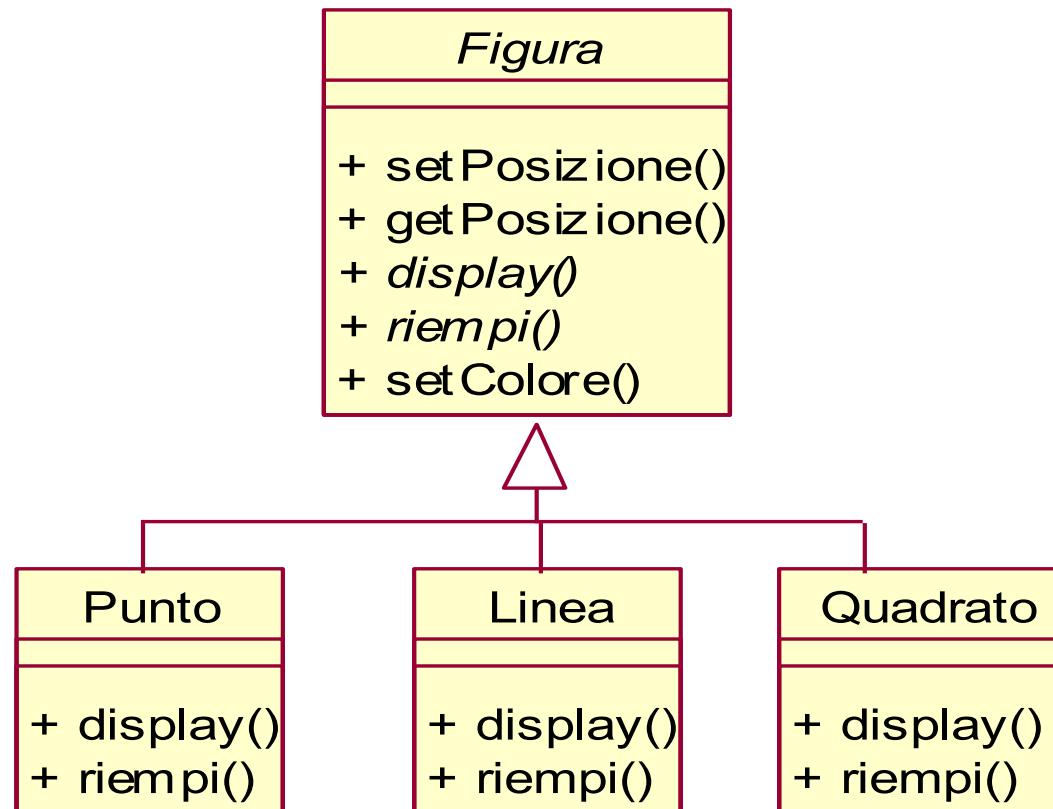
- ☞ Adattare l'interfaccia di una classe già pronta all'interfaccia che il cliente si aspetta
 - “Adapter” come l'adattatore per prese di corrente.

Adapter – Motivazione 1/4

Talvolta una classe progettata per il riuso non è riusabile solo perché la sua interfaccia non coincide con quella del dominio specifico di un' applicazione.

Adapter – Motivazione 2/4

Si consideri un editor grafico che fornisce l' interfaccia Figura i cui metodi *riempi* e *display* sono implementati dalle classi Punto, Linea e Quadrato.

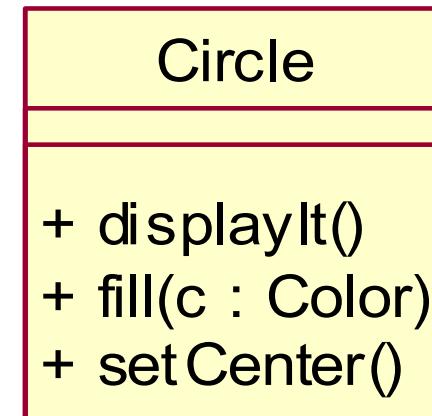


Adapter – Motivazione 3/4

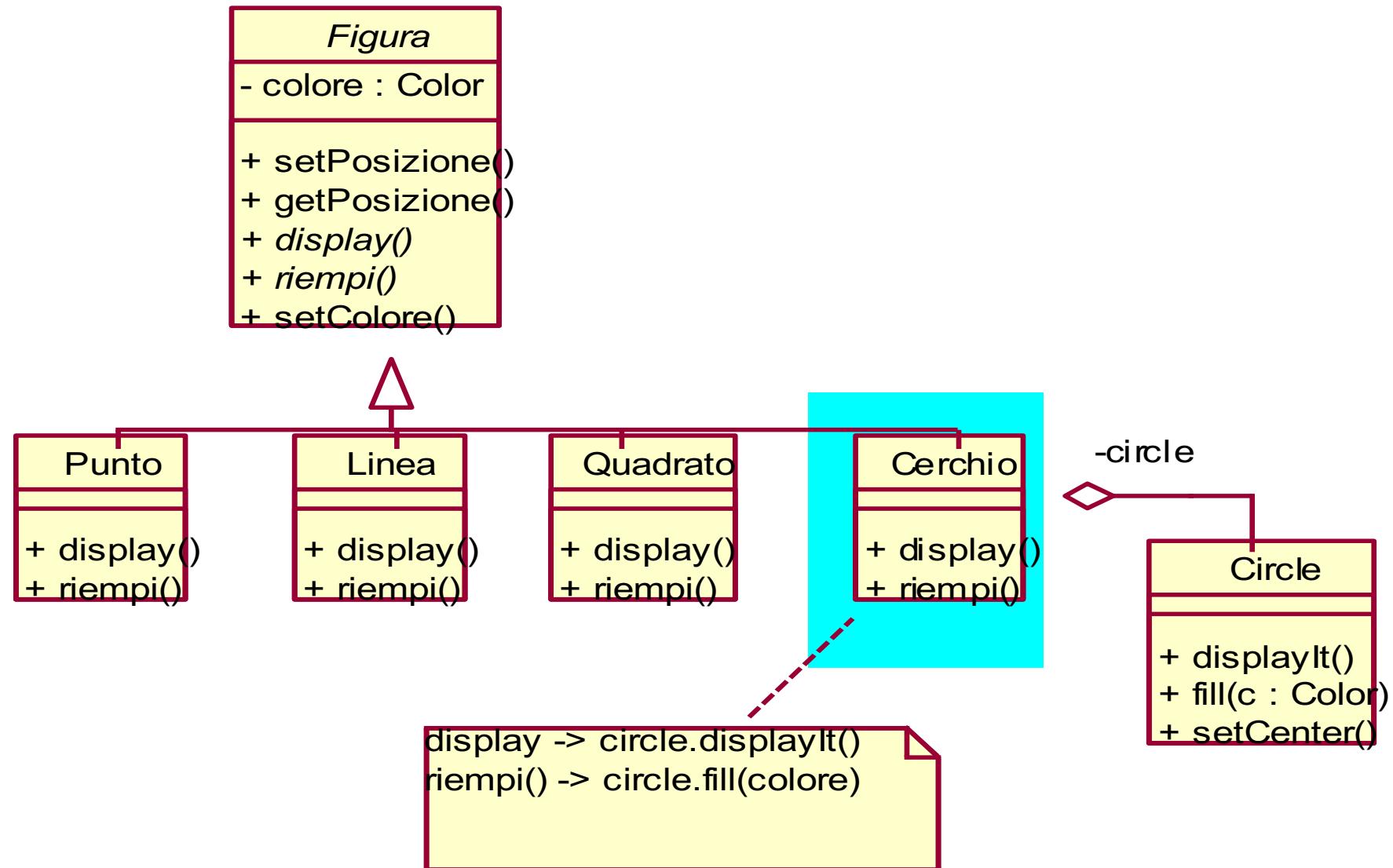
☞ Vogliamo aggiungere Cerchio

- Lo implementiamo da zero?
 - Fatica inutile...
- Usiamo la classe Circle?
 - Ha un'interfaccia diversa che non possiamo modificare.

☞ Creiamo un adattatore.



Adapter – Motivazione 4/4



Adapter – Applicabilità

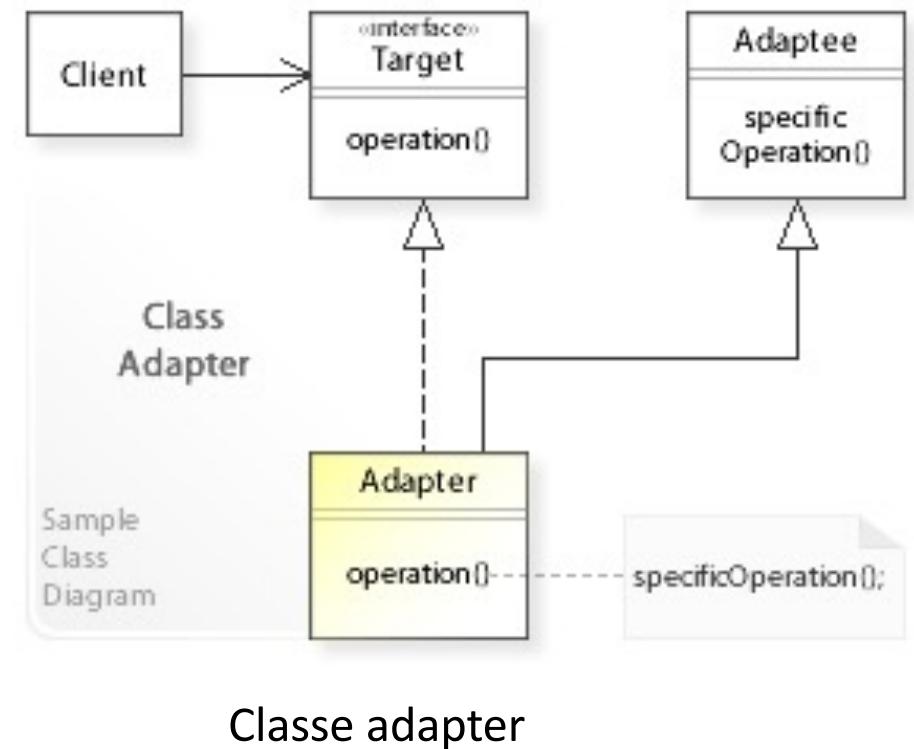
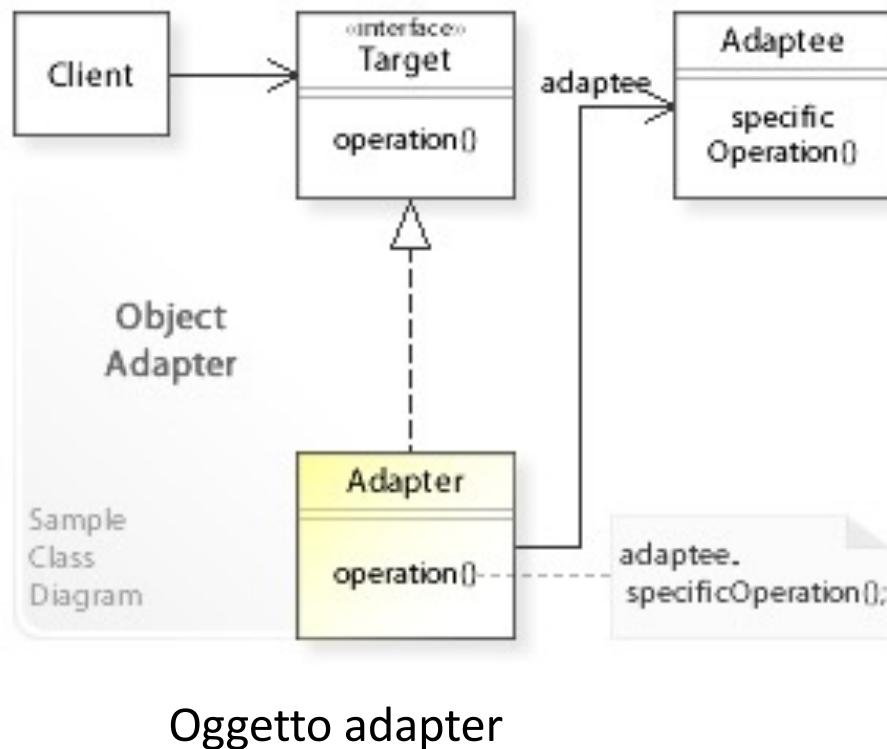
☞ Usiamo Adapter quando:

- Vogliamo utilizzare una classe esistente la cui interfaccia non risponde alle nostre necessità;
- Si vuole realizzare una classe riusabile che coopera con classi che non necessariamente hanno un'interfaccia compatibile;
- Occorre utilizzare sottoclassi esistenti le cui interfacce non possono essere adattate sottoclassando ciascuna di esse. Un object adapter può adattare l'interfaccia della classe base.

Due tipi di Adapter

- Object Adapter
 - Basato su delega/composizione.
- Class Adapter
 - Basato su ereditarietà;
 - L'adattatore eredita sia dall'interfaccia attesa sia dalla classe adattata;
 - No eredità multipla: l'interfaccia attesa deve essere un'interfaccia, non una classe.

Adapter – Struttura



Adapter – Partecipanti

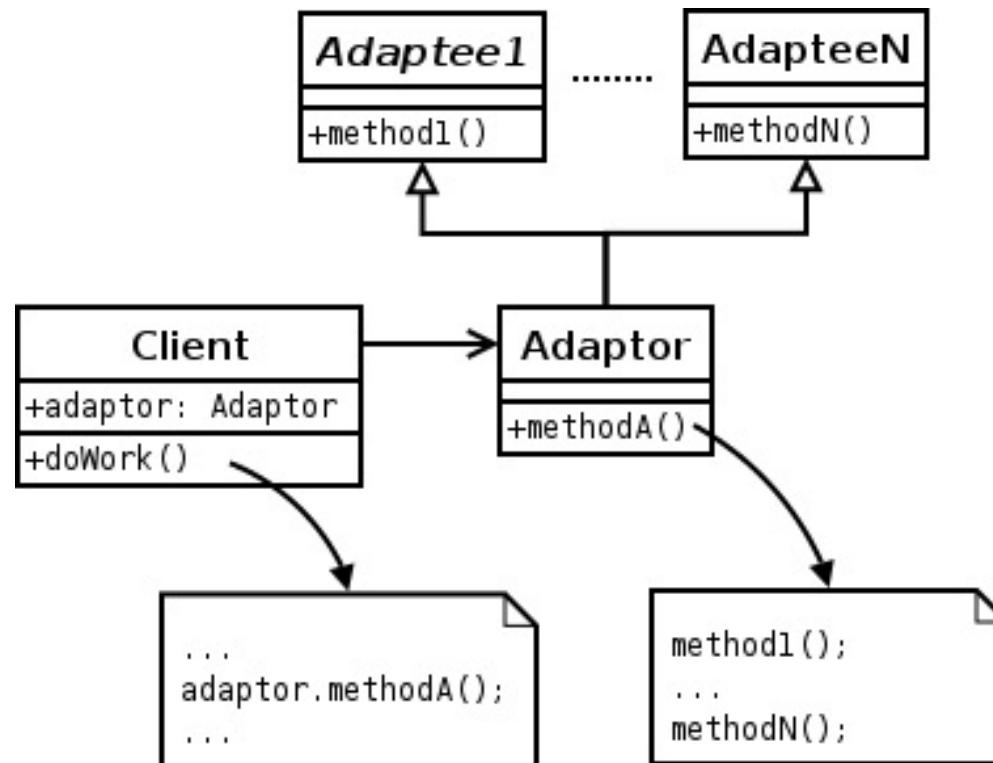
- Target
 - Definisce l'interfaccia usata dal client
- Client
 - Collabora con gli oggetti conformi all'interfaccia Target.
- Adaptee
 - L'interfaccia esistente che deve essere adattata.
- Adapter
 - Adatta l'interfaccia di Adaptee all'interfaccia Target.

Adapter – Conseguenze 1

- Class Adapter:
 - Non va bene se vogliamo adattare anche le sottoclassi;
 - Adapter potrebbe sovrascrivere dei comportamenti di Adaptee;
- Object Adapter:
 - Un singolo Adapter funziona con gli oggetti Adaptee e quelli delle sottoclassi;
 - È difficile sovrascrivere il comportamento di Adaptee.
- In alcuni linguaggi (Smalltalk) è possibile realizzare classi che includono un adattamento dell’interfaccia (*pluggable adapter*).
- Se due client richiedono di vedere un oggetto in maniera diversa è possibile definire dei two-way adapters.

Adapter – Conseguenze 2

Se due client richiedono di vedere lo stesso oggetto in maniera diversa è possibile definire dei two-way adapters usando l' ereditarietà multipla



La teoria dei pattern

Introduzione

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

[Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel. A Pattern Language. Oxford Univ. Press, New York, 1977.]

- I pattern sono soluzioni a problemi ricorrenti che si incontrano in applicazioni reali.
- I pattern spiegano come realizzare oggetti e le loro interazioni

Tipi di Pattern

I pattern sono applicabili in ogni fase del ciclo di vita di un software

Esistono:

- pattern di analisi;
- pattern architetturali;
- pattern di progettazione (design);
- pattern di codifica (implementazione)
- pattern di test

Vantaggi dei pattern

- Riuso della conoscenza/esperienza di progettazione
 - Raramente i problemi sono nuovi e unici
 - I pattern forniscono indicazioni su “dove cercare soluzioni ai problemi”
- terminologia comune e condivisa
 - Es.: basta dire “Qui ci serve un Façade”
- prospettiva di alto livello
 - Evitano di dover gestire troppo presto i dettagli della progettazione.

pattern architetturali e framework

- Il progetto di architetture software può essere guidato da pattern e framework
- Un esempio di pattern architetturale è MVC
- Un esempio di framework architetturale è il modello “4+1 viste” di Krutchen
- Su una scala più piccola, i design pattern sono “micro-architetture”

Design Pattern

Generalmente i design pattern sono catalogati e descritti utilizzando pochi elementi essenziali:

- **nome**: descrive in maniera simbolica il problema di progettazione e le sue soluzioni;
- **problema**: descrive *quando* applicare il pattern. Può descrivere problemi di progettazione specifici o anche strutture di classi o di oggetti sintomo di progettazione non flessibile;
- **soluzione**: descrive gli elementi (oggetti, classi e idiom) che costituiscono il progetto, le loro relazioni, le responsabilità e le collaborazioni. La soluzione è la descrizione astratta di un problema di progettazione e del modo in cui una configurazione di elementi (classi e oggetti, nel nostro caso), può risolverlo, ma non descrive una particolare progettazione o implementazione;
- **conseguenze**: sono i risultati e i vincoli che si ottengono applicando il pattern, utili per valutare soluzioni alternative, capire e stimare i benefici derivanti dall' applicazione del pattern.

Un po' di storia ...

- Nel 1987 W. Cunningham e K. Beck lavoravano al linguaggio Smalltalk e individuarono alcuni pattern
- La nozione di pattern è stata resa popolare da Gamma, Helm, Johnson e Vlissides, che lavoravano alla definizione di framework di sviluppo (E++, Unidraw)
- Essi divennero noti come la Banda dei Quattro (“*Gang of four*”, Go4).
- I design patterns da essi definiti adoperano un approccio di documentazione uniforme

The Gang of Four

Gamma, Helm, Johnson, Vlissides at OOPSLA



Evoluzione dei Design Patterns

Christopher Alexander

The Timeless Way of Building

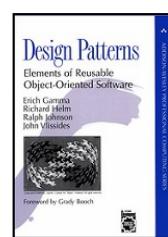
A Pattern Language: Towns, Buildings, Construction

Architecture

1970'

Gang of Four (GoF)

Design Patterns: Elements of Reusable Object-Oriented Software



Object Oriented Software Design

1995'

Many Authors

Other Areas:
Middleware, HCI,
Management, Education, ...

2000'

Catalogo GoF dei Design Patterns

- Il primo e più famoso libro di pattern:
 - Gamma e altri, *Design Patterns – Elements of Reusable Object-Oriented Software*, AddisonWesley 1994
- Il volume descrive 23 pattern utili per la programmazione a oggetti in C++.

Struttura di ogni pattern:

Name and classification;

Also Known As;

Applicability;

Participants;

Consequences;

Sample Code;

Related Patterns.

Intent;

Motivation;

Structure;

Collaborations;

Implementation;

Known Uses.

Classificazione dei pattern GoF

- I design pattern possono essere classificati in base a due criteri:
- Scopo (**purpose**), che riflette cosa fa il pattern:
 - **Creational**: astraggono il processo di creazione (instanziazione) di oggetti;
 - **Structural**: trattano la composizione delle classi o oggetti per formare strutture più complesse;
 - **Behavioral**: si occupano del modo (algoritmi) in cui classi o oggetti interagiscono reciprocamente e distribuiscono fra loro le responsabilità;
- Ambito (**scope**), specifica se il pattern è relativo a classi o ad oggetti:
 - **Class**: trattano le relazioni statiche, determinate a tempo di compilazione, tra classi e sottoclassi;
 - **Object**: trattano relazioni dinamiche, che variano a tempo di esecuzione, tra oggetti.

Classificazione

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

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

Design pattern GoF

Design pattern creazionali

Design pattern creazionali

- I design pattern creazionali aiutano a creare oggetti in modo efficiente
- Consentono di rendere un sistema indipendente da come gli oggetti sono creati, rappresentati e delle relazioni di composizione tra essi.
- Se basati su classi, utilizzano l'ereditarietà per modificare la classe istanziata.
- Se basati su oggetti, delegano l'istanziazione ad altri oggetti.
- Incapsulano la conoscenza relativa alle classi concrete utilizzate dal sistema.
- Nascondono come le istanze delle classi sono create e assemblate.

Design pattern creazionali

- **Abstract Factory**: crea un oggetto che serve a creare famiglie di altri oggetti
- **Factory Method**: permette di creare un oggetto senza specificare la sua classe
- **Builder**: separa la costruzione di un oggetto complesso dalla sua rappresentazione
- **Prototype**: costruisce un'istanza completa di un oggetto che serve per essere clonato
- **Singleton**: crea un oggetto che restituisce una istanza di se stesso che rimarrà unica

Pattern Singleton – Intento

- Assicura che una classe abbia una sola istanza e un unico punto globale di accesso ad essa
- Singleton: in italiano si dice “singololetto”
- In matematica, un *singololetto* è un insieme contenente esattamente un unico elemento.
 - Per esempio, l'insieme $\{0\}$ è un **singololetto**.
 - Anche l'insieme $\{\{1,2,3\}\}$ è un **singololetto**: l'unico elemento in esso contenuto è un insieme (che invece non è un **singololetto**).

Pattern Singleton – Motivazione

- In alcuni sistemi è importante che una classe sia istanziata al massimo una sola volta
- La classe stessa è responsabile di tenere traccia della sua unica istanza e fornire un modo per accedere a tale istanza.

Singleton - esempio

- Dobbiamo gestire una risorsa specifica condivisa ma presente in singola istanza: es. un printer spooler
- Dobbiamo essere certi che in ogni momento lo spooler è unico, altrimenti potrebbero arrivare richieste in conflitto per la stessa risorsa

Singleton – Struttura

Singleton

- Instance: Singleton = null

+ getInstance(): Singleton
- Singleton(): void

Come garantire che un oggetto sia unico?

```
Public class Singleton{  
    Private static Singleton uniqueInstance;  
    //other useful instance variables here  
  
    Private Singleton() {}  
  
    Public static Singleton getInstance() {  
        if (uniqueInstance == null){  
            uniqueInstance= new Singleton();  
        }  
        Return uniqueInstance,  
    }  
    //other useful methods here  
}
```

Singleton
- instance : Singleton = null
+ getInstance() : Singleton
- Singleton() : void

Pattern Singleton – Applicabilità

Il Singleton va usato quando:

- deve esserci esattamente una istanza di una classe e deve essere accessibile ai clients da un punto di accesso globale;
- l'unica istanza deve essere estendibile mediante subclassing e i clienti devono essere capaci di usare una istanza estesa senza modificare il loro codice.

Singleton – Partecipanti

Singleton

- Definisce un'operazione `Instance()` che consente ai clienti di accedere alla sua unica istanza;
- `Instance()` è un metodo di classe (i.e. static);
- Può essere responsabile di creare la sua propria unica istanza.

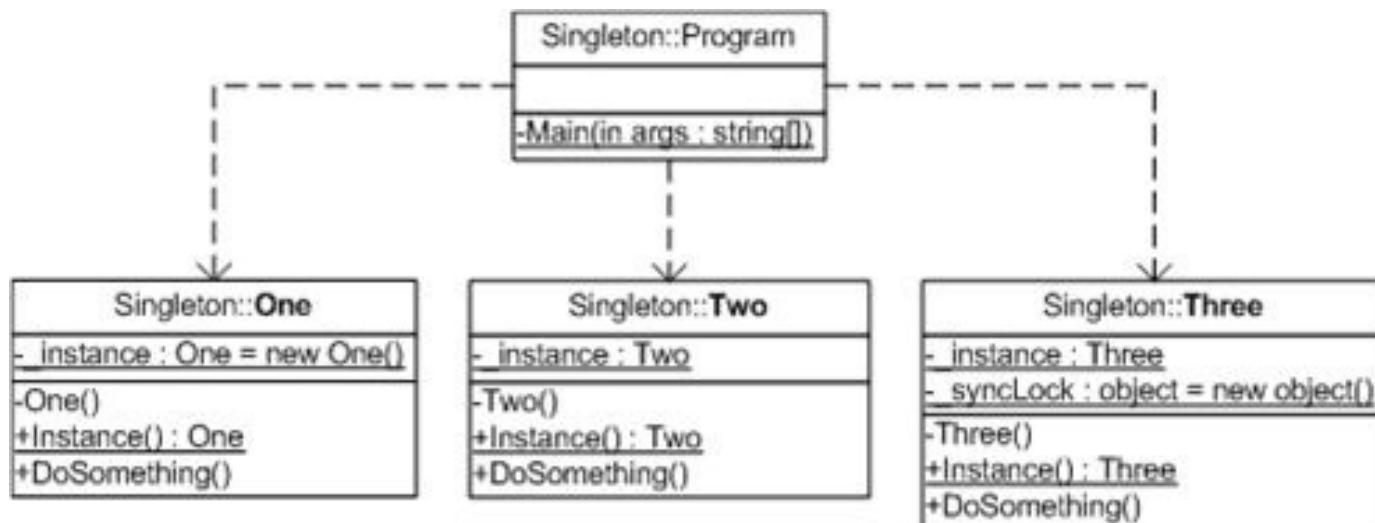
Singleton – Conseguenze

- Accesso controllato all'unica istanza.
- Un sostituto elegante per variabili esterne (consente di evitare l' “inquinamento” del name space con numerose variabili esterne).
- Il Singleton può essere esteso per consentire un numero variabile di istanze.

Pattern Singleton: variante thread safe

1. La classe One prevede l'inizializzazione statica dell'istanza. La proprietà Instance ritorna l'oggetto equivalente di tipo One statico e privato.
2. La classe Two prevede l'inizializzazione dinamica su richiesta tramite il controllo del riferimento all'istanza. La proprietà Instance ritorna anche in questo caso l'oggetto equivalente di tipo Two statico e privato.
3. La classe Three effettua un doppio controllo sul riferimento all'istanza, dentro e fuori ad un blocco a mutua esclusione e in base ad esso attiva l'istanza. La proprietà Instance ritorna l'oggetto equivalente di tipo Three statico e privato.

I primi due casi non sono thread-safe, il terzo sì. La presenza della regione critica garantisce che la creazione dell'istanza sia effettivamente eseguita una volta sola, anche in un contesto multi-thread



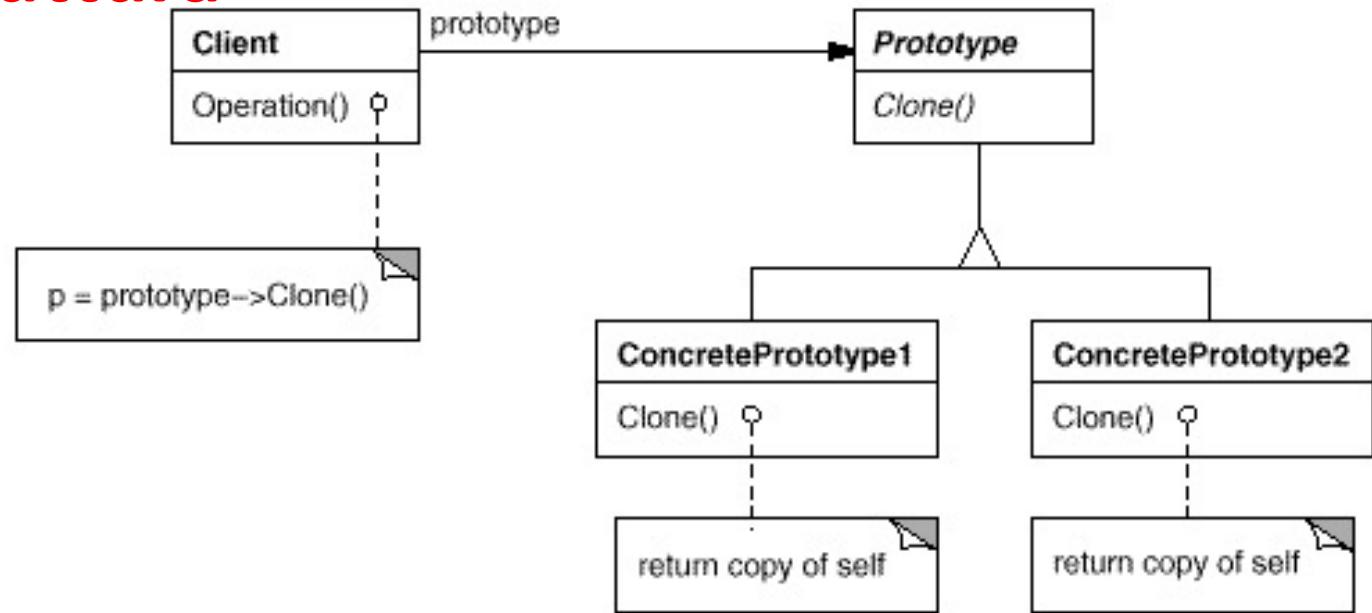
Pattern Prototype: intento

- Creazionale
- Specifica i tipi di oggetti da creare usando un'istanza prototipale, e crea nuovi oggetti copiando tale prototipo
- Prototype permette di creare nuovi oggetti clonando un oggetto iniziale, detto *prototipo*.
- A differenza di pattern quali Abstract Factory o Factory method, Prototype permette di specificare nuovi oggetti a run-time, utilizzando un gestore per salvare e reperire dinamicamente le istanze degli oggetti desiderati

Pattern Prototype: conseguenze

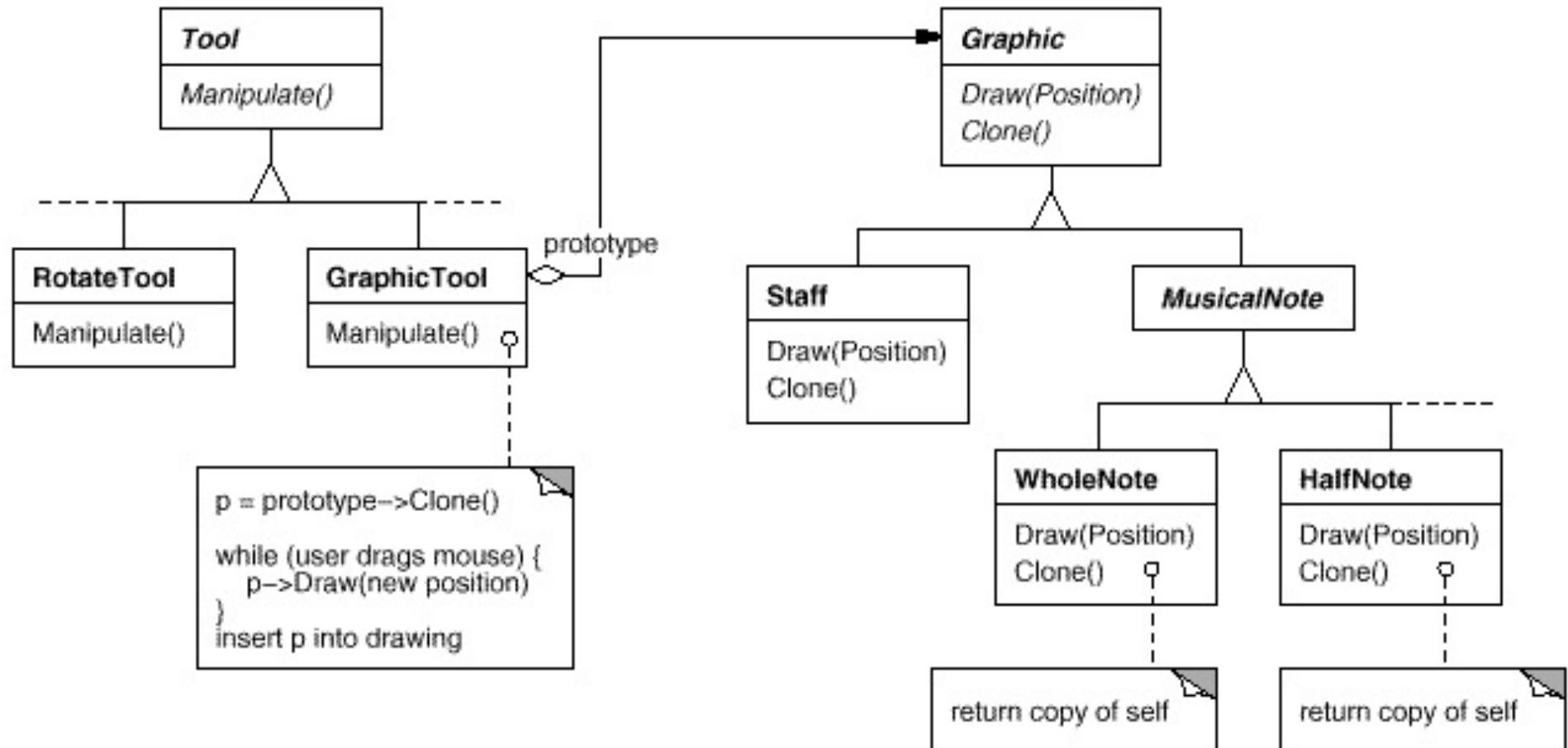
- Rende indipendente un sistema dal modo in cui i suoi oggetti vengono creati.
- Inoltre può essere utile quando
 - le classi da istanziare sono specificate solamente a tempo d'esecuzione, per cui un codice statico non può occuparsi della creazione dell'oggetto,
 - per evitare di costruire una gerarchia di factory in parallelo a una gerarchia di prodotti, come avviene utilizzando Abstract factory e Factory method,
 - quando le istanze di una classe possono avere soltanto un limitato numero di stati, per cui può essere più conveniente clonare al bisogno il prototipo corrispondente piuttosto che creare l'oggetto e configurarlo ogni volta

Prototype: struttura



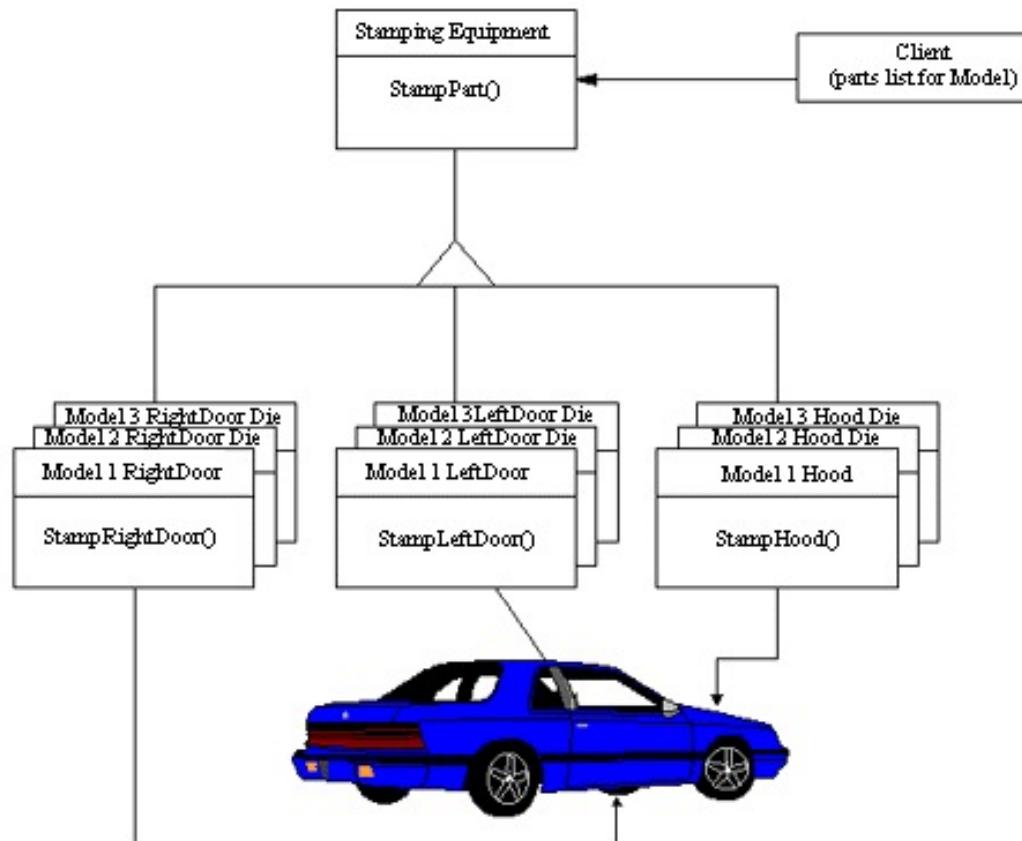
- Prototype (Graphic) dichiara un'interfaccia per clonare se stesso.
- ConcretePrototype (Staff, WholeNote, HalfNote) implementa un'operazione per clonare se stesso.
- Client (GraphicTool) crea un nuovo oggetto chiedendo ad un prototipo di clonare se stesso

Pattern Prototype: esempio



Abstract Factory – Intento

Offre un'interfaccia per creare famiglie di oggetti correlati, senza specificare le loro classi concrete.



Abstract Factory – Motivazione

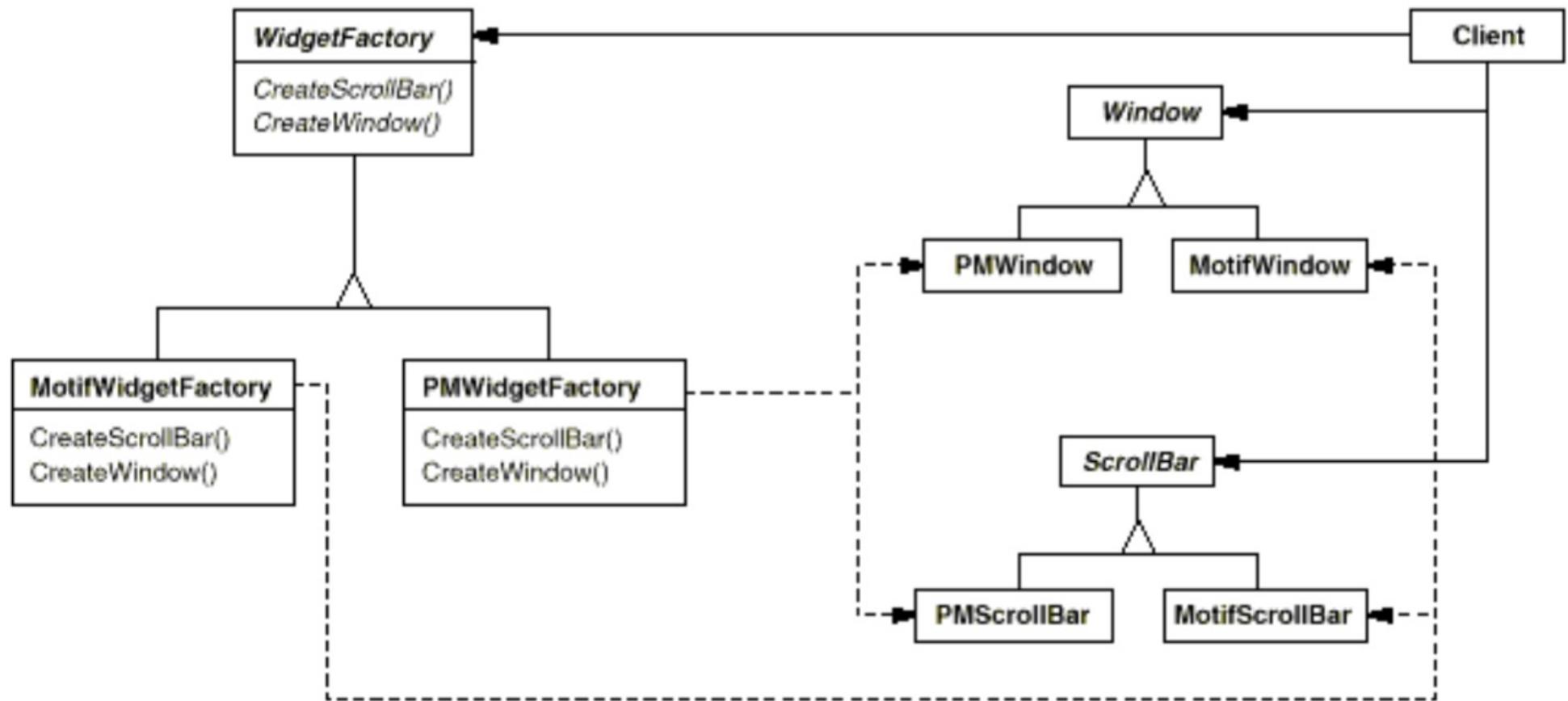
- Ci sono casi in cui vogliamo creare differenti elementi tra loro in relazione senza legarci alle loro istanze concrete
- Ad esempio, un toolkit di interfacce grafiche deve supportare più standard look-and-feel, che definiscono l'estetica e il comportamento dei “widget” (finestre, scrollbar, pulsanti, etc.)
- I client usano una WidgetFactory astratta, che dichiara una interfaccia per creare ogni tipo base di widget, definiti a loro volta come classi astratte. Quindi, esiste una sottoclassificazione concreta che implementa ogni tipo di widget per uno specifico look-and-feel.

Abstract Factory – Applicabilità

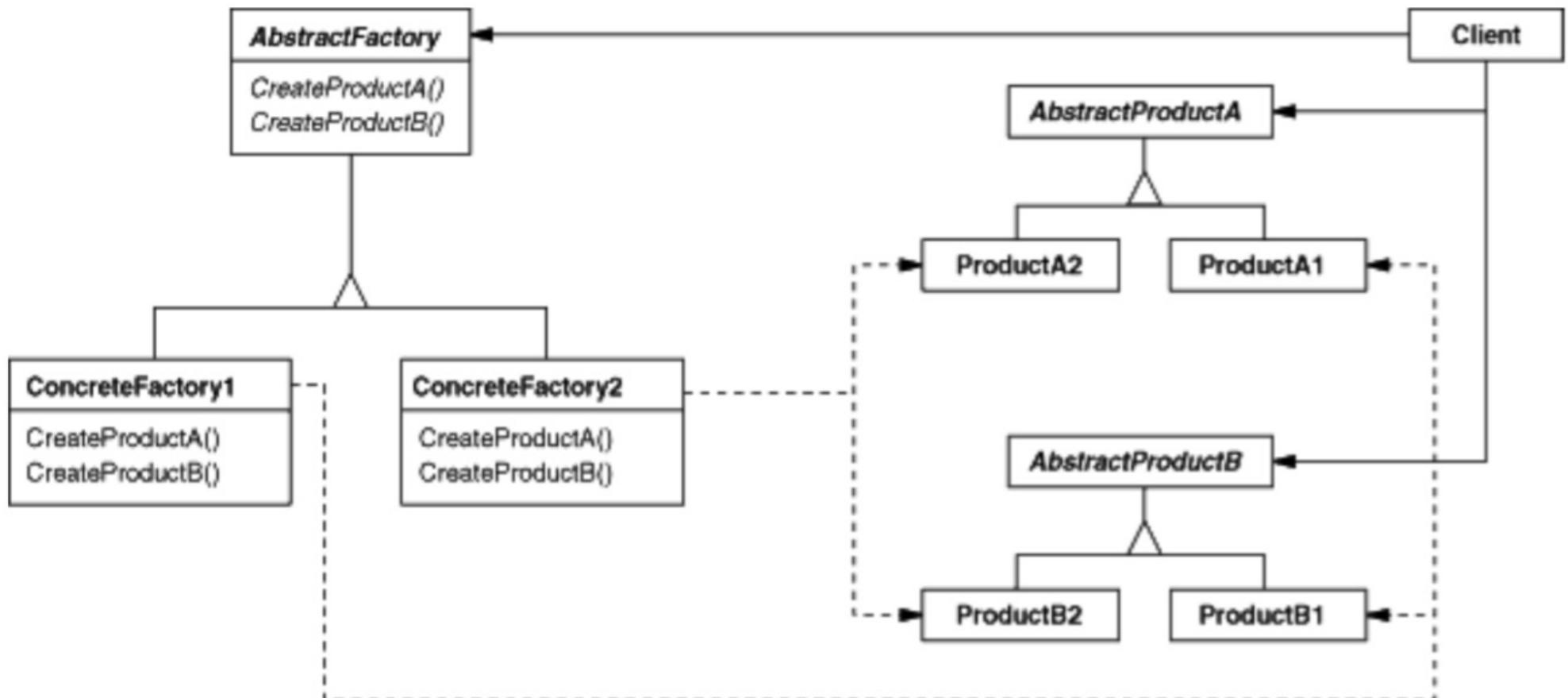
Abstract factory va usato quando:

- Un sistema deve essere indipendente da come i suoi prodotti sono creati, composti e rappresentati.
- Un sistema va configurato per più *famiglie di prodotti*
- Una famiglia di prodotti è progettata per servirsi di un solo insieme di classi per volta, ed occorre garantire questo vincolo
- Occorre offrire una libreria di prodotti, ma si vuole esporre solo le loro interfacce e non le loro implementazioni.

Abstract Factory – Struttura (1/2)



Abstract Factory – Struttura (2/2)



Abstract Factory – Partecipanti

- **AbstractFactory (WidgetFactory)**
 - Dichiara una interfaccia per le operazioni che creano oggetti astratti
- **ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)**
 - Implementa le operazioni per creare oggetti concreti
- **AbstractProduct (Window, ScrollBar)**
 - Dichiara una intefaccia per un tipo di oggetto prodotto
- **ConcreteProduct (MotifWindow, MotifScrollBar)**
 - Definisce un oggetto prodotto che deve essere creato dalla corrispondente concrete factory
- **Client**
 - Usa solo interfacce dichiarate da AbstractFactory e AbstractProduct

Abstract Factory – Conseguenze

- Isola le classi concrete: aiuta a controllare le classi di oggetti che una applicazione crea, isolandole nelle concrete factory piuttosto che nel codice del client.
- Rende semplice cambiare famiglie di prodotti scegliendo una concrete factory.
- Promuove la coerenza tra gli oggetti creati quando gli oggetti prodotti in una famiglia sono progettati per lavorare insieme.
- Supportare nuovi tipi di prodotti è difficile, perché necessita di cambiare la Factory Interface e tutte le sottoclassi.

Design pattern GoF

Design pattern strutturali

Pattern strutturali

- I pattern strutturali propongono classi o oggetti per formare strutture più grandi.
- I pattern strutturali basati su classi utilizzano l'ereditarietà per generare classi che combinano le proprietà di classi base.
- I pattern strutturali basati su oggetti mostrano come comporre oggetti per realizzare nuove funzionalità.
- Permettono composizione che viene modificata a run-time, cosa impossibile con la composizione statica (quella con classi).

Design pattern strutturali

structural patterns GoF già visti:

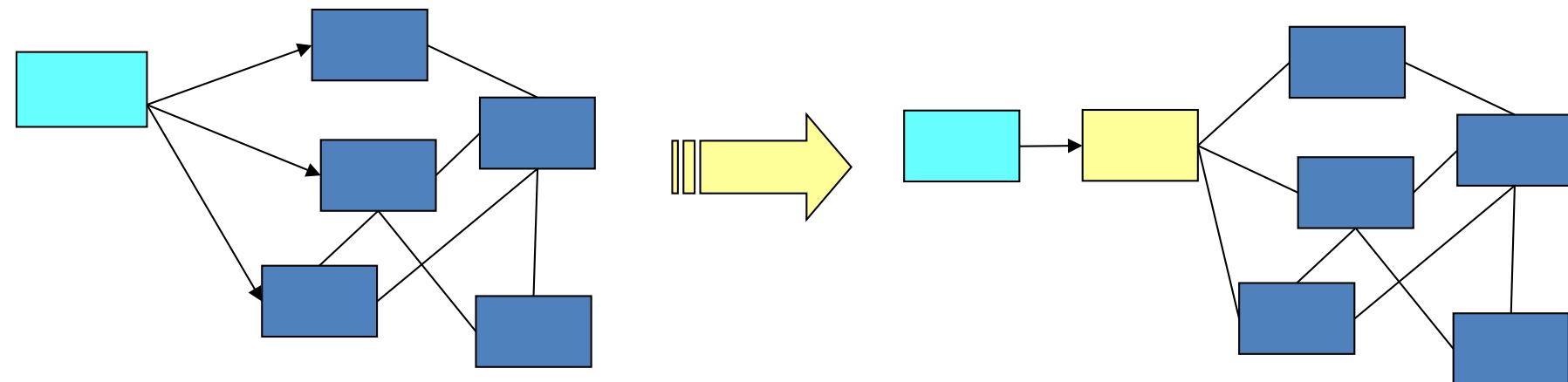
- **Adapter:** Unisce le interfacce di differenti classi
- **Bridge:** separa l' interfaccia di un oggetto dalla sua implementazione
- **Composite:** una struttura ad albero per rappresentare oggetti complessi
- **Decorator:** aggiunge funzionalità (responsabilità) ad oggetti dinamicamente
- **Flyweight:** permette di separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima fra differenti istanze

Altri structural patterns GoF:

- **Facade:** una singola classe che rappresenta un' intero sistema
- **Proxy:** un oggetto che rappresenta un altro oggetto

Façade – Intento

- Rendere più semplice l'uso di un (sotto)sistema.
- Fornire un'unica interfaccia per un insieme di funzionalità “sparse” su più interfacce/classi.



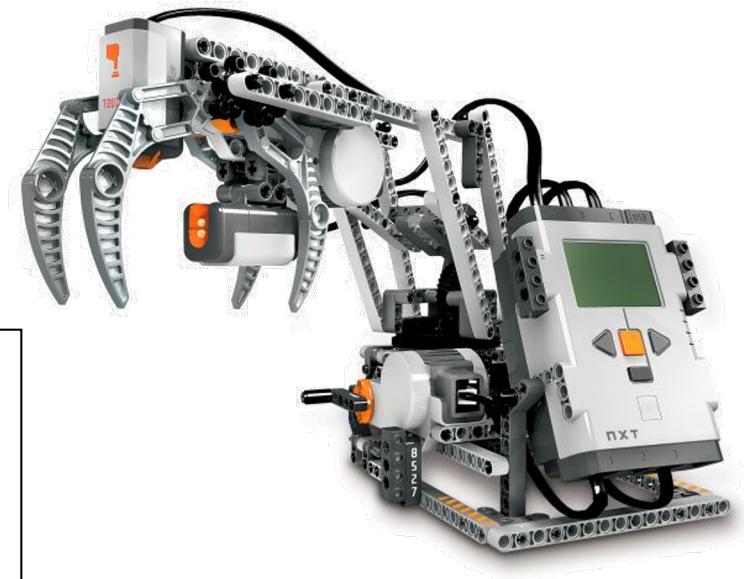
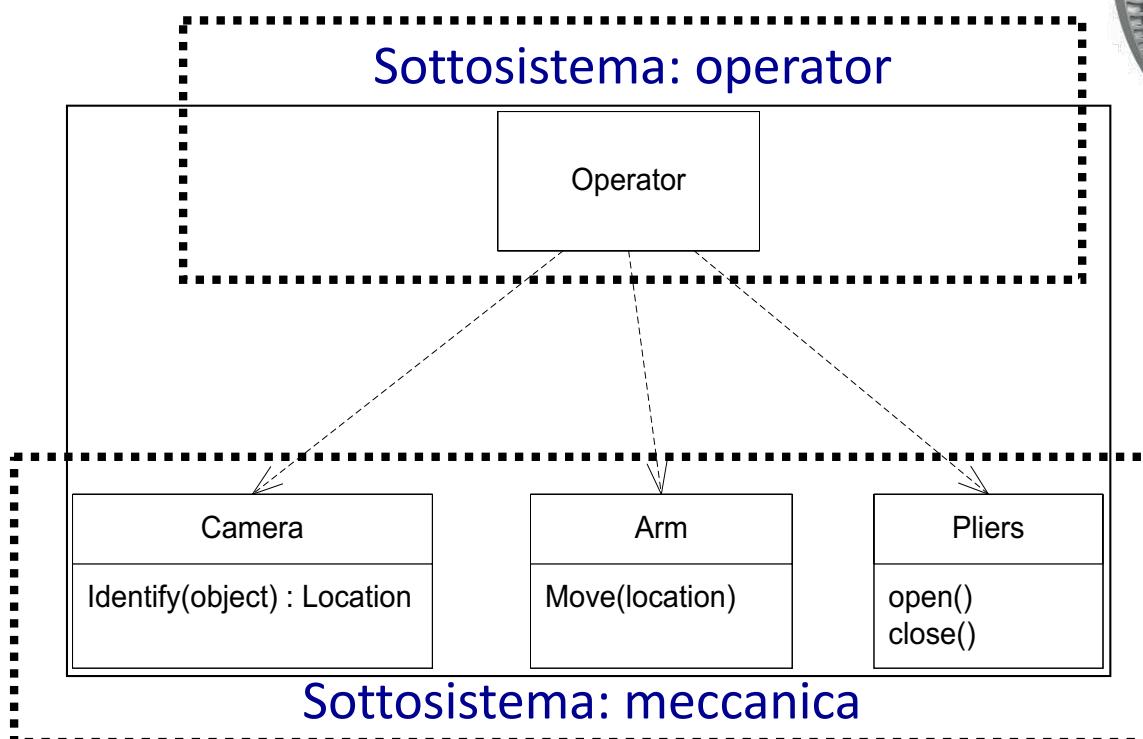
Façade – Motivazione 1/4

- La suddivisione di un sistema in sottosistemi aiuta a ridurre la complessità.
- Tuttavia, occorre diminuire le dipendenze dei client con i sottosistemi.
- L’ utilizzo di un oggetto façade offre un’ interfaccia semplice e univoca alle funzionalità di ciascun sottosistema.

Façade – Motivazione 2/4

Robot con quattro classi:

- teleCamera (per identificare gli oggetti)
- Arm (mobile)
- Pliers (per afferrare)



Façade – Motivazione 3/4

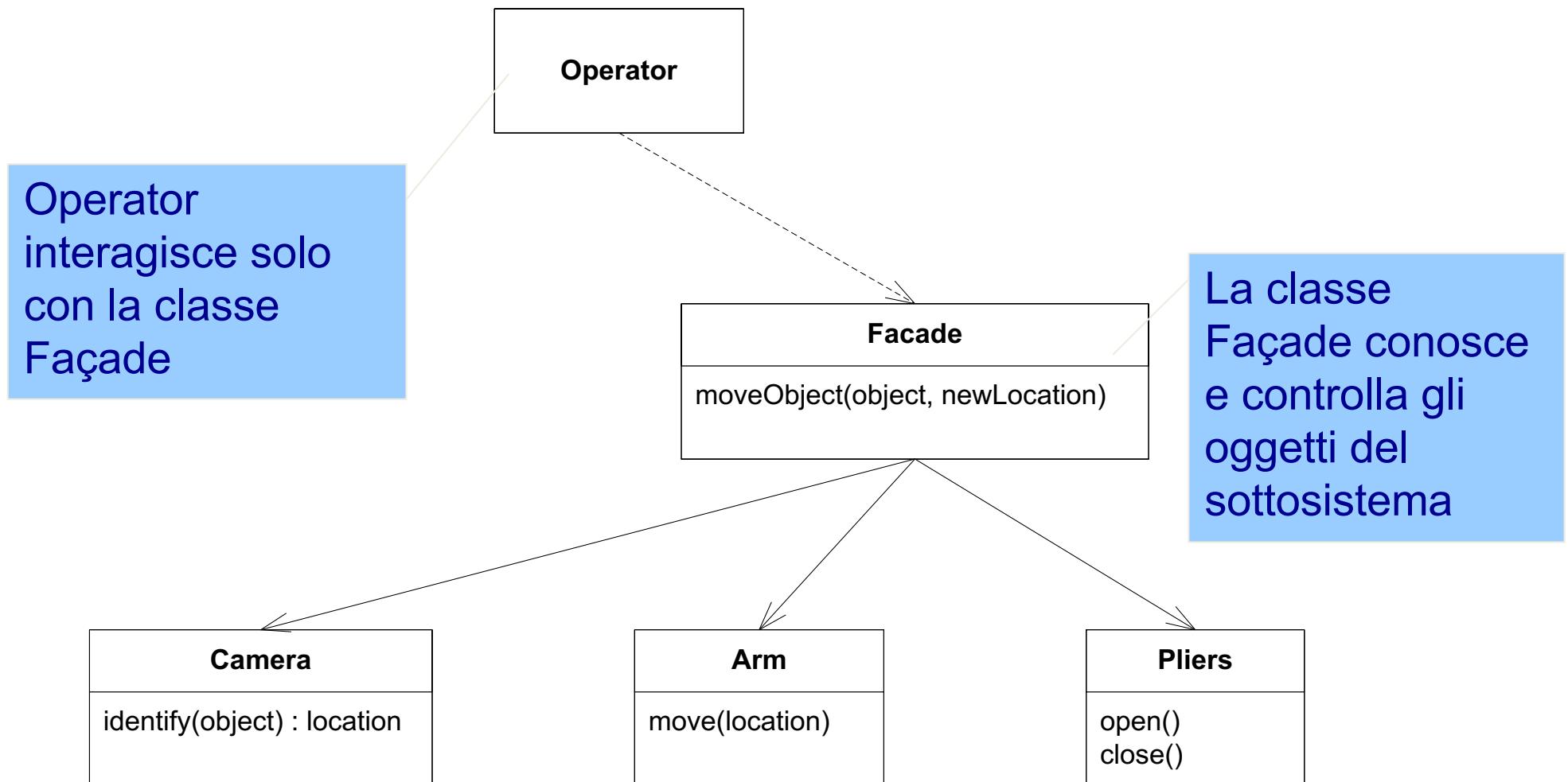
- Quanto deve conoscere della meccanica l'operatore?
- Si supponga di voler individuare un oggetto e spostarlo in una locazione predefinita:

```
oldLocation = Camera.identify(object);  
Arm.move(oldLocation);  
Pliers.close();  
Arm.move(newLocation);  
Pliers.open();
```

Problema: non c'è encapsulamento:

–L'operatore deve conoscere la struttura ed il comportamento del sottosistema

Façade – Motivazione 4/4

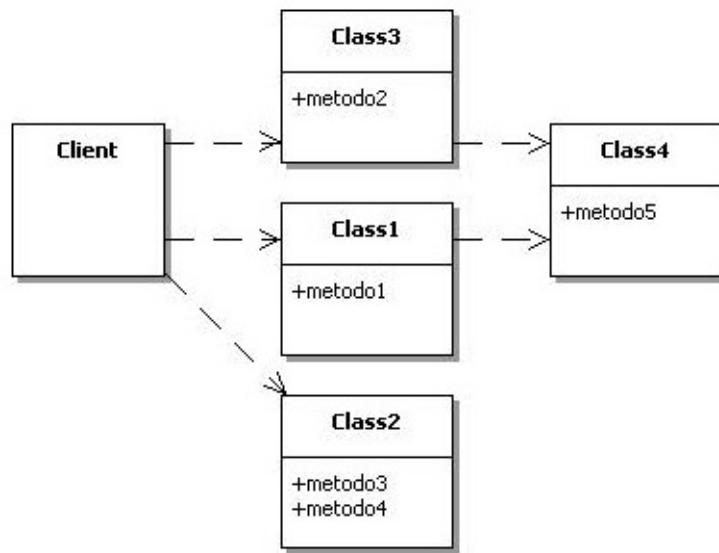


Façade – Applicabilità

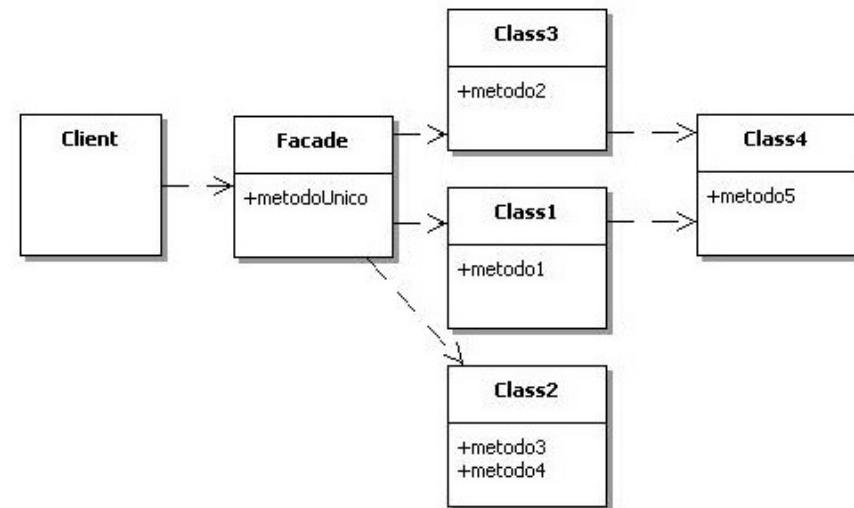
Utilizzare il pattern Façade:

- Per fornire una vista semplice e di default di un sottosistema complesso;
- Nel caso di sottosistema stratificato, è possibile utilizzare una Façade come entry point per ciascun livello.

Façade – Struttura



Senza uso di Façade



Con uso di Façade

Façade – Partecipanti

- Façade:
 - Conosce quali classi di un sottosistema sono responsabili per una richiesta;
 - Delega le richieste del client agli oggetti appropriati del sottosistema.
- Classi del sottosistema:
 - Implementano le funzionalità del sottosistema;
 - Gestiscono il lavoro assegnato da Façade;
 - Non hanno alcun riferimento della facciata.

Façade – Conseguenze

- Diminuisce l'accoppiamento fra cliente e sottosistema.
- Nasconde al cliente le componenti del sottosistema.
- Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema.

Façade – Implementazione

L'accoppiamento tra un client e un sottosistema può essere ridotto rendendo Façade una classe astratta con sottoclassi concrete per diverse implementazioni di un sottosistema.

- In tal caso, i client possono comunicare con il sottosistema attraverso l'interfaccia della classe astratta Façade.

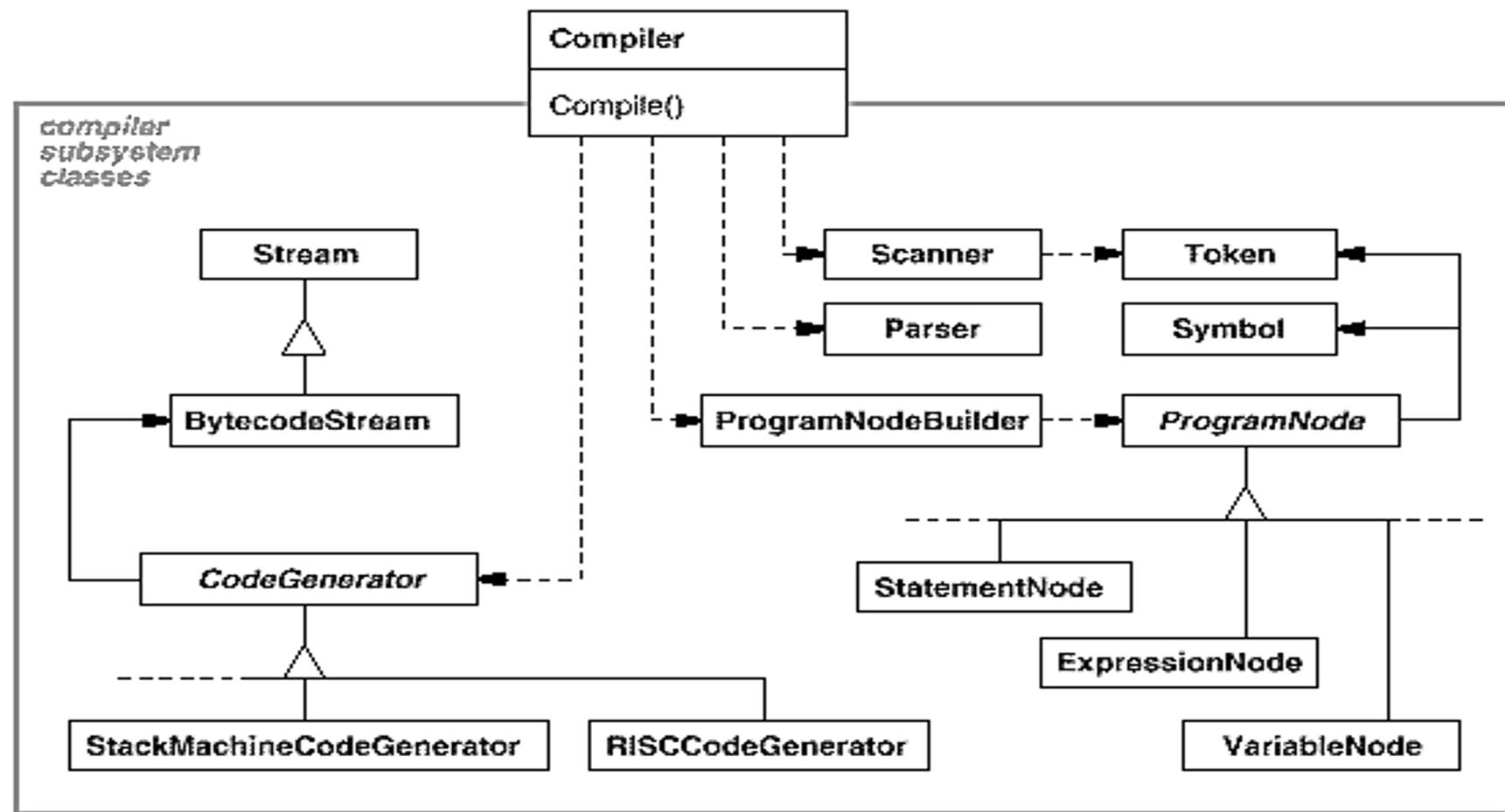
Façade – Esempio

- Compilatore:

- Ha parecchie classi: Parser, Scanner, Token, SyntacticTree, CodeGenerator, ecc....;
- Façade: Classe Compiler con metodo compile().

Façade – Esempio 1/2

- ☞ Si consideri il class diagram seguente relativo ad un ambiente di programmazione che dà alle applicazioni accesso al sottosistema per la compilazione.



Façade – Esempio 2/2

- Il sottosistema di compilazione definisce una classe `BytecodeStream` che implementa un flusso di oggetti `Bytecode`. Ciascun oggetto `Bytecode` incapsula un bytecode che specifica le istruzioni macchina.
- La classe `Scanner` del sottosistema riceve un flusso di caratteri e genera un flusso di token, un token per volta.
- Il `Parser` effettua l' analisi dei token generati dallo `Scanner` e genera un parse tree utilizzando un `ProgramNodeBuilder`.
- Il parse tree è fatto di istanze di sottoclassi di `ProgramNode`.
- `ProgramNode` utilizza un oggetto `CodeGenerator` per generare codice macchina nella forma di oggetti `Bytecode` in un flusso `BytecodeStream`.
- La classe `Compiler` è una “facciata” che mette tutti i pezzi insieme fornendo una semplice interfaccia per la compilazione del codice sorgente e la generazione del codice per una particolare macchina.

Façade vs. Adapter

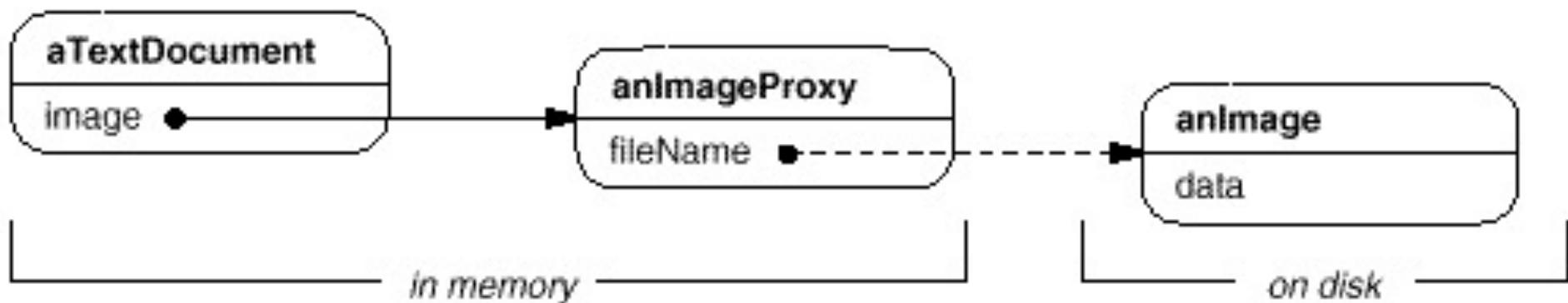
- Entrambi sono “wrapper” (involucri).
- Entrambi si basano su un’interfaccia, ma:
 - Façade la semplifica;
 - Adapter la converte.

Pattern Proxy: intento

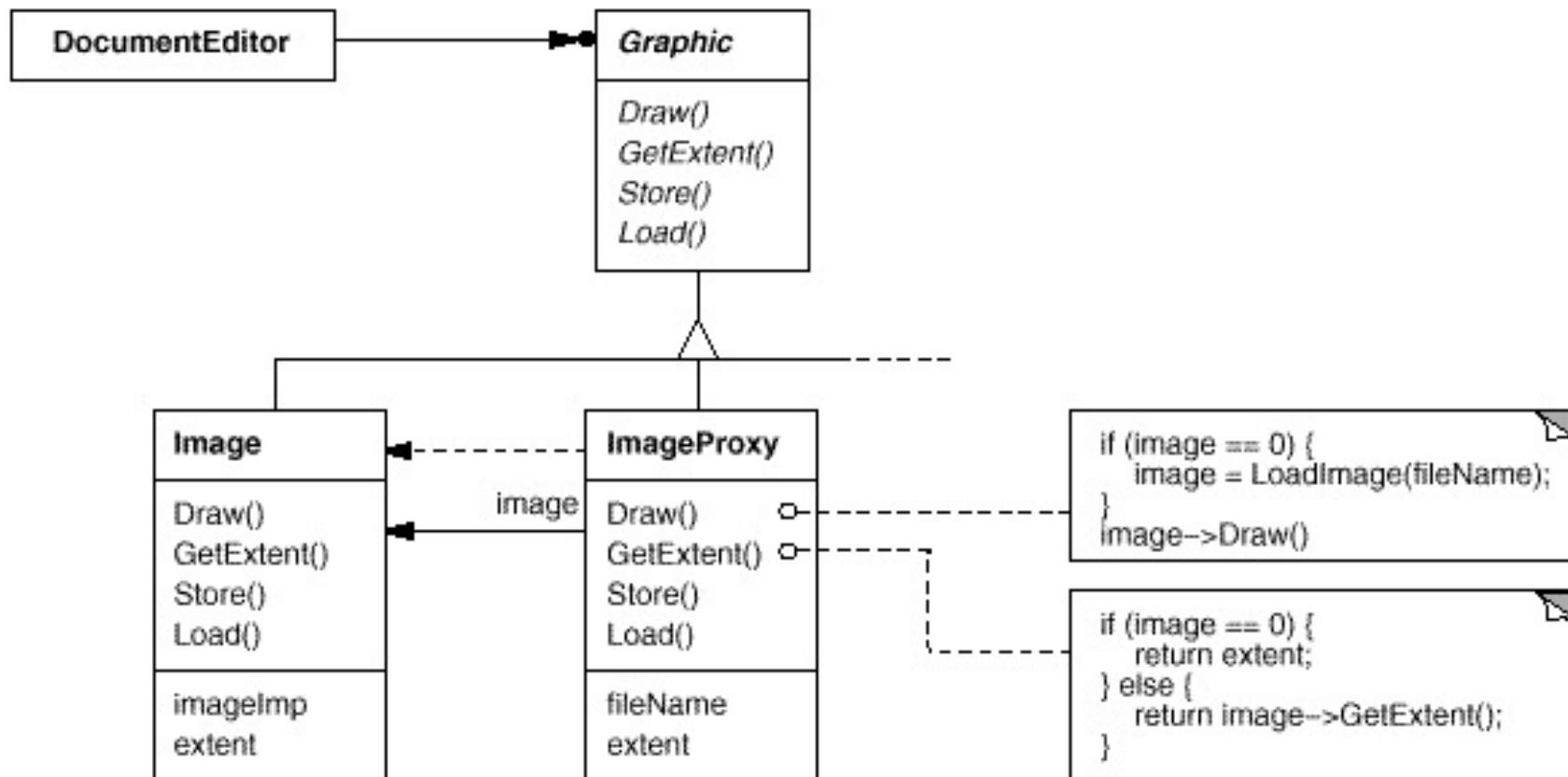
- Fornire un segnaposto surrogato di un altro oggetto per controllarne l'accesso
- Un motivo è differire il costo di creazione e inizializzazione al momento in cui l'oggetto surrogato serve davvero

Pattern Proxy: esempio

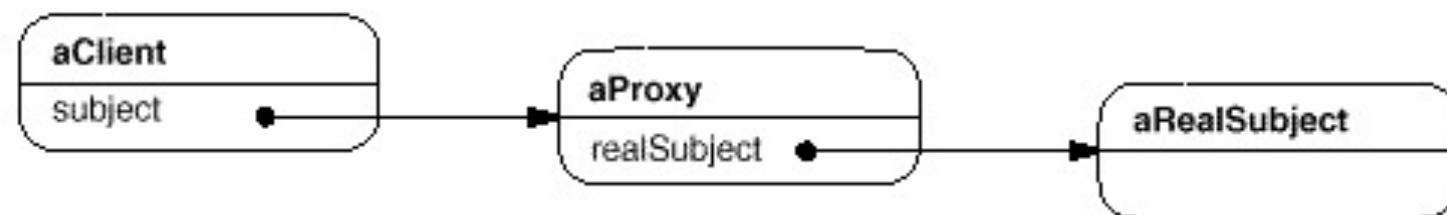
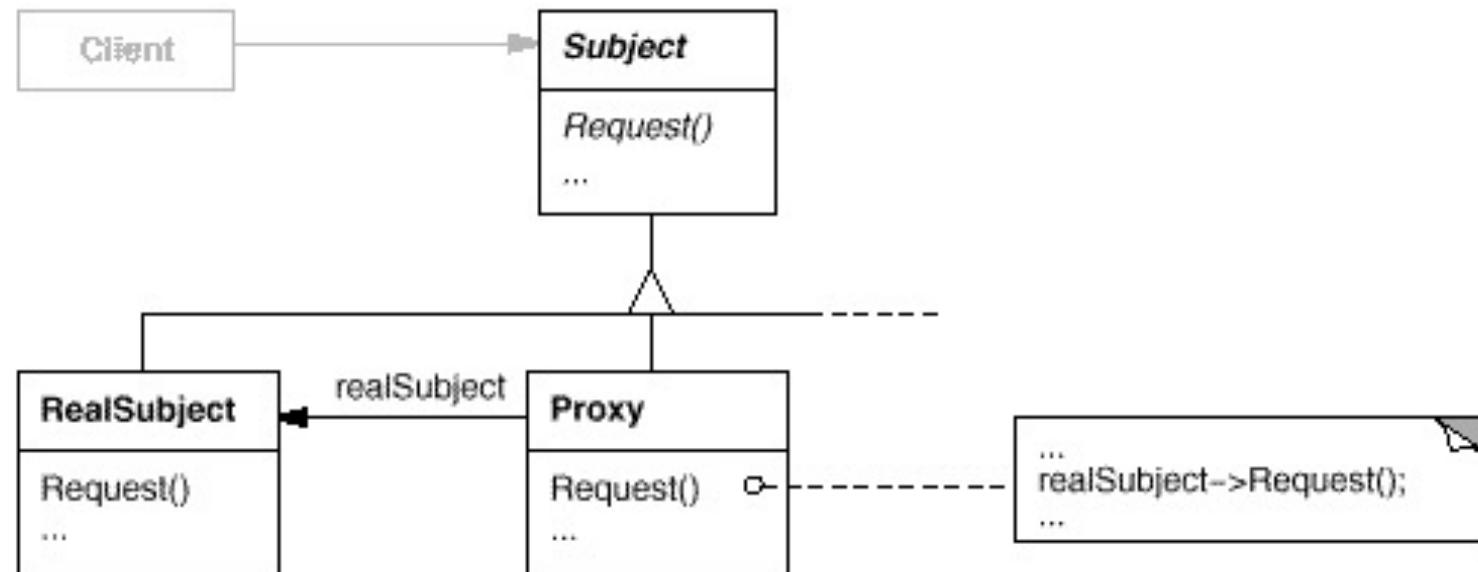
- Si consideri un editor che può inserire elementi grafici in un documento.
- Certi oggetti grafici (es. immagini rasterizzate) sono costosi da creare
- Siccome vogliamo aprire in fretta un documento, vogliamo evitare di creare oggetti pesanti in termini di tempo di creazione al momento dell'apertura del documento



Pattern Proxy: esempio



Pattern Proxy: struttura



A run-time

Pattern Proxy: pattern correlati

- Un **adapter** fornisce un'interfaccia diversa all'oggetto che adatta. Invece, un proxy fornisce la stessa interfaccia del suo surrogato (Tuttavia in alcuni casi un proxy usato per proteggere l'accesso potrebbe rifiutarsi di eseguire un'operazione che il suo surrogato accetterebbe, dunque l'interfaccia del proxy sarebbe diversa, cioè un sottoinsieme di quella del suo surrogato).
- Sebbene i **decorator** abbiano implementazioni simili a quelle dei proxy, i decorator hanno scopo differente. Un decorator aggiunge responsabilità ad un oggetto, invece un proxy controlla l'accesso ad un oggetto.

Design pattern GoF

Design pattern comportamentali

I pattern comportamentali

- I pattern comportamentali riguardano algoritmi e l'assegnamento di responsabilità tra gli oggetti.
- Descrivono non solo i partecipanti (oggetti o classi) ma anche gli schemi di comunicazione da adoperare.
- Tali pattern descrivono flussi di controllo complessi, difficili da descrivere a run-time.
- Spostano l'attenzione dal flusso di controllo e permettono di concentrarsi su come gli oggetti sono interconnessi

Design pattern comportamentali

Behavioral patterns GoF già visti:

- **Command**: isola il codice che effettua un'azione (eventualmente complessa) dal codice che ne richiede l'esecuzione
- **Iterator**: accede gli elementi di un aggregato in sequenza
- **Visitor**: separa un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da aggiungere nuove operazioni senza modificare la struttura stessa
- **Memento**: cattura ed esternalizza lo stato di un oggetto senza violare l'incapsulamento
- **Observer**: gestisce eventi disaccoppiando gli oggetti interessati da quelli che generano gli eventi
- **Strategy**: modifica dinamicamente un algoritmo di un oggetto
- **Template method**: definisce lo scheletro di un algoritmo, differendo alcuni passi alle sue sottoclassi

Altri behavioral patterns GoF:

- **Interpreter**: gestisce la valutazione di frasi in un linguaggio
- **Chain of responsibility**: evita l'accoppiamento tra mittente e ricevente permettendo a più oggetti di esaminare un msg
- **Mediator**: definisce un intermediario che controlla come si comportano alcuni oggetti
- **State**: permette ad un oggetto di modificare il suo comportamento quando cambiano i suoi attributi interni

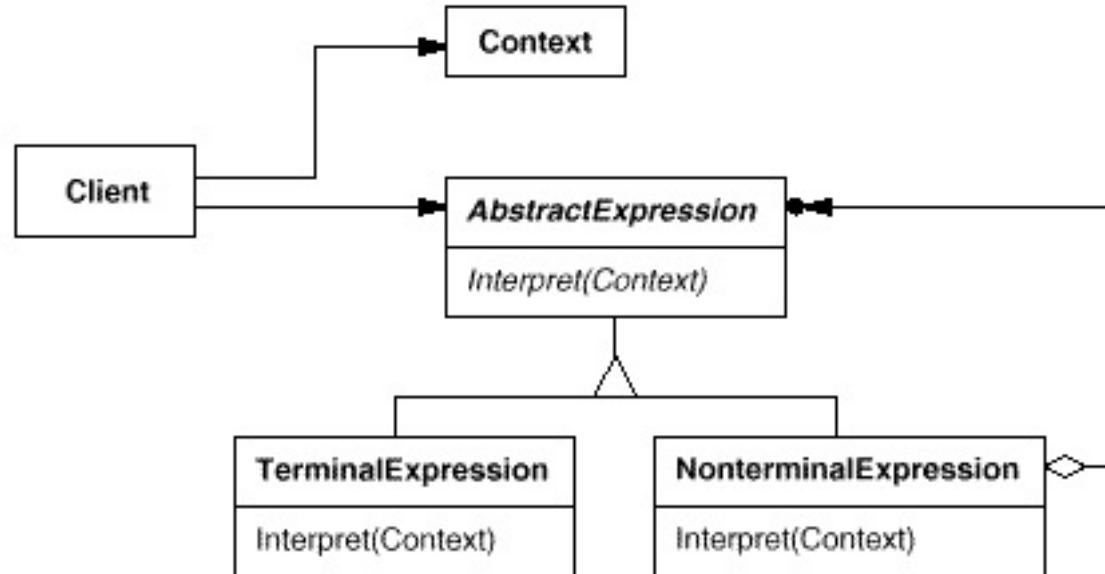
Pattern Interpreter: intento

Dato un linguaggio, definisce una rappresentazione della sua grammatica insieme ad un interprete che utilizza questa rappresentazione per l'interpretazione delle espressioni in quel determinato linguaggio

Esempio: questa è una grammatica per espressioni regolari

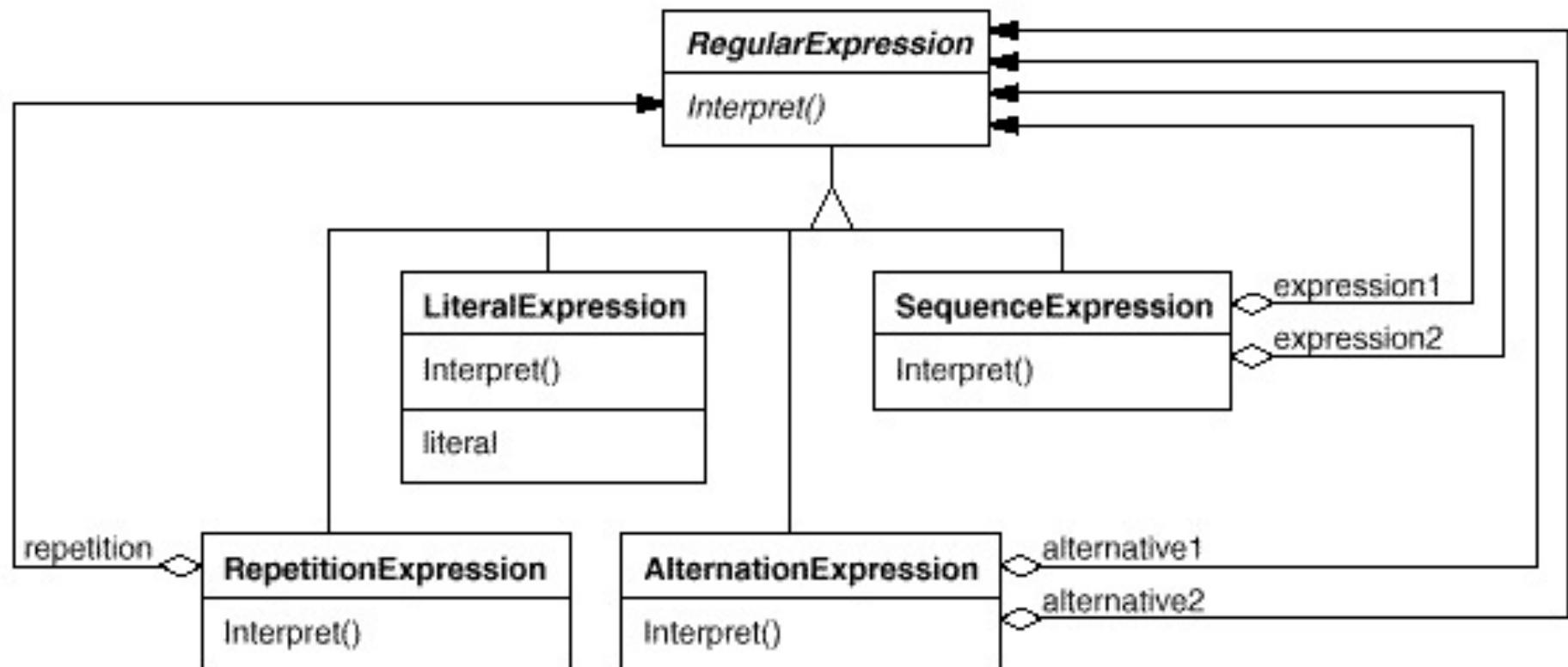
```
expression ::= literal | alternation | sequence | repetition | '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

Pattern Interpreter: struttura



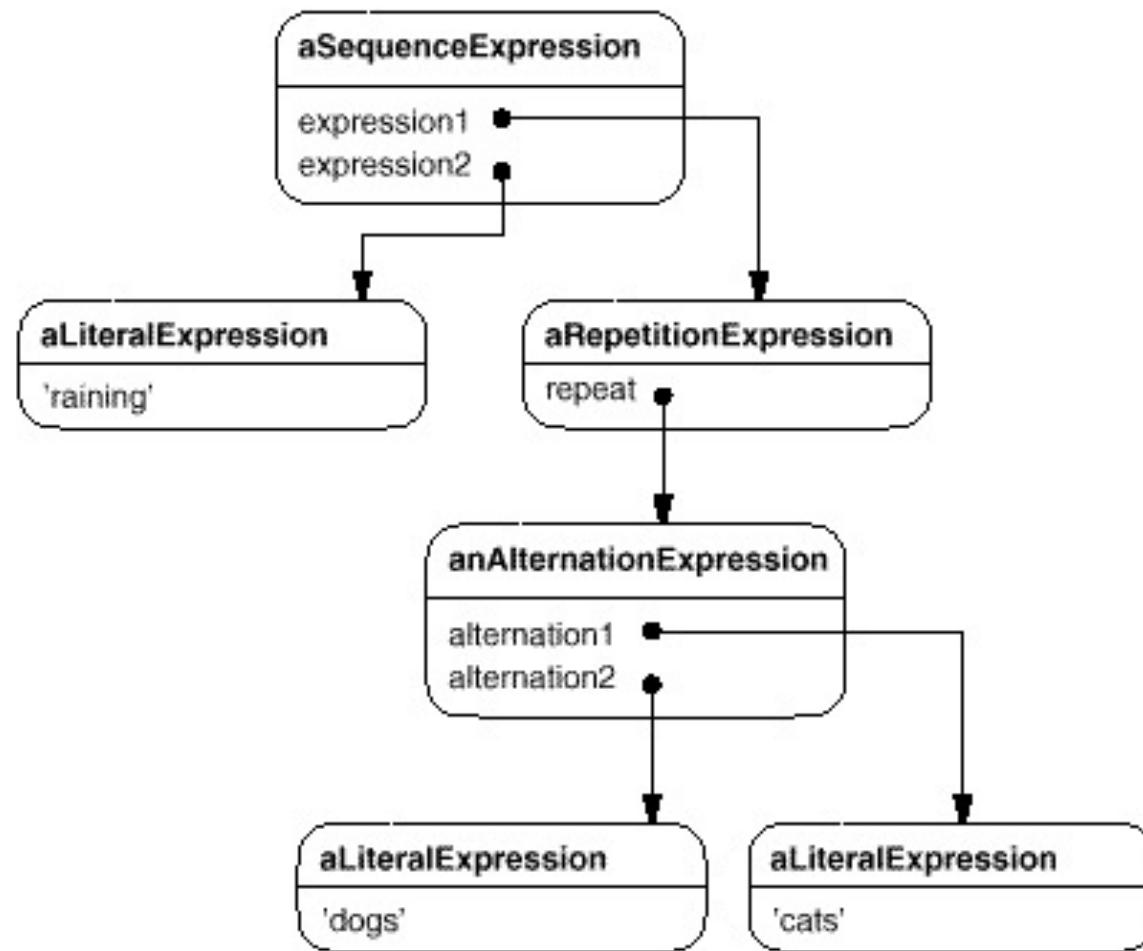
- **AbstractExpression** dichiara un'operazione astratta *Interpret(Context)*
- **TerminalExpression** implementa un'operazione di interpretazione associata coi simboli terminali della grammatica
- **NonterminalExpression** una per ogni regola della grammatica
- **Context** contiene informazioni globali dell'Interprete
- **Client** costruisce un *albero di sintassi astratta*

Pattern Interpreter: esempio/1



Pattern Interpreter: esempio/2

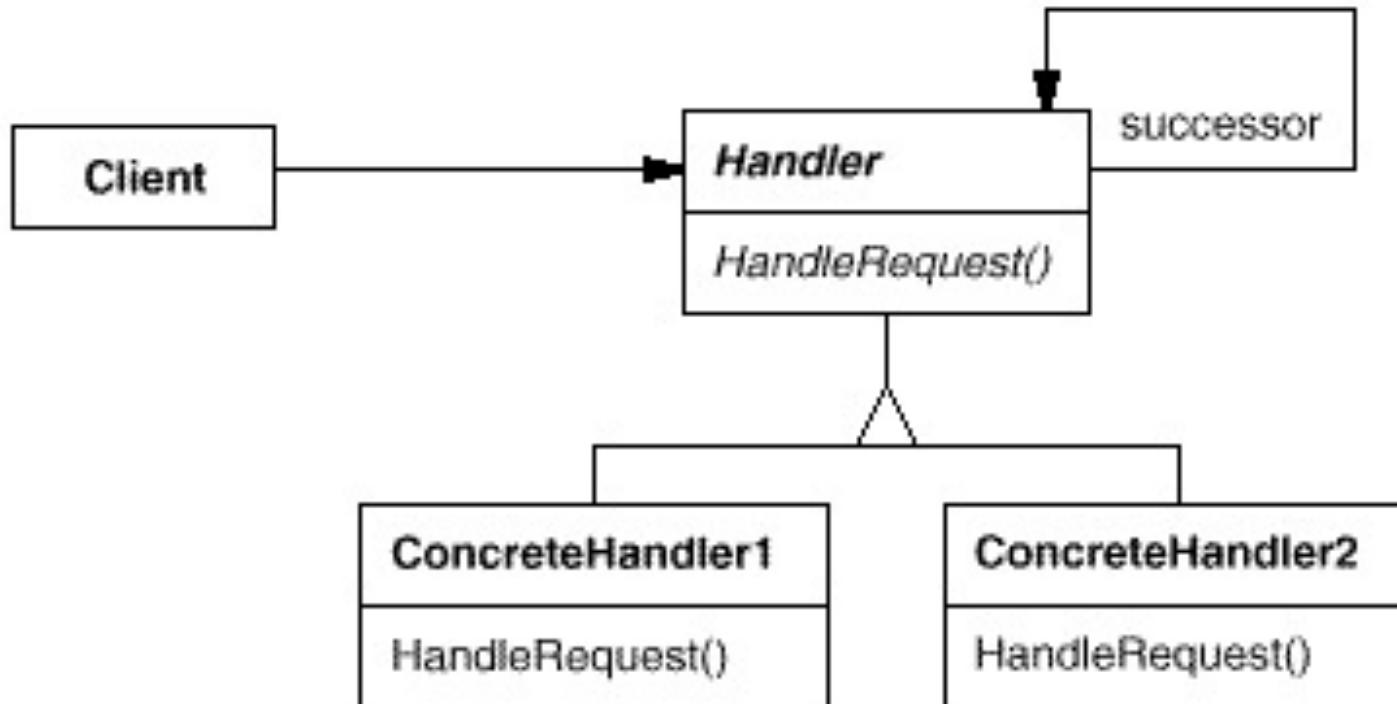
Questo diagramma rappresenta l'espressione regolare
`raining & (dogs | cats) *`



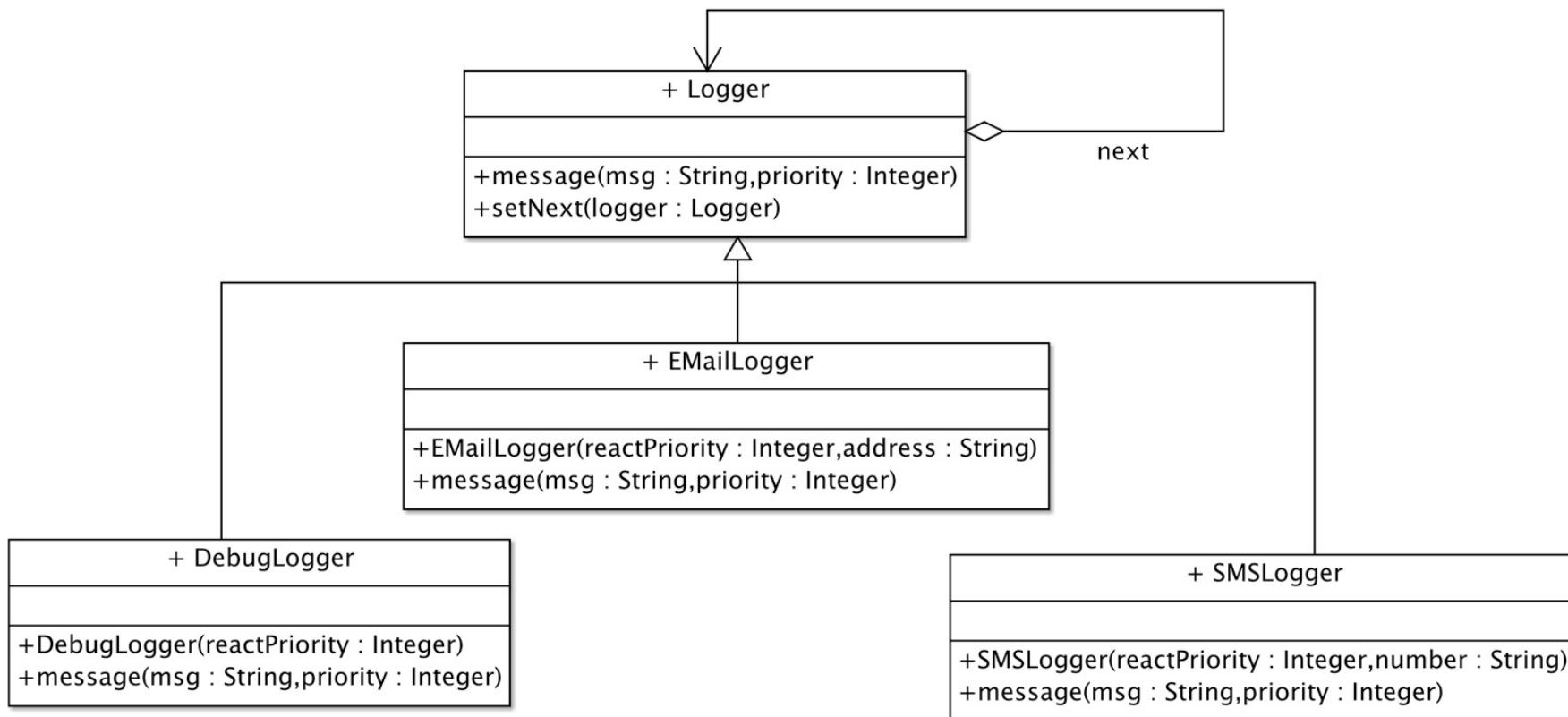
Pattern Chain of responsibility: intento

- Disaccoppia il mittente di una richiesta dal suo ricevente, permettendo a più oggetti di gestire (*handle*) la richiesta.
- Gli oggetti riceventi sono messi in sequenza: la richiesta viene passata lungo la catena fino a trovare un oggetto che la gestisca.
- Esiste anche un meccanismo per aggiungere dinamicamente un oggetto alla catena
- chain of responsibility è la versione oo di un comando if... then... else if... else if... else... endif e in più offre il vantaggio che i blocchi condizione-azione possono essere riconfigurati a runtime.

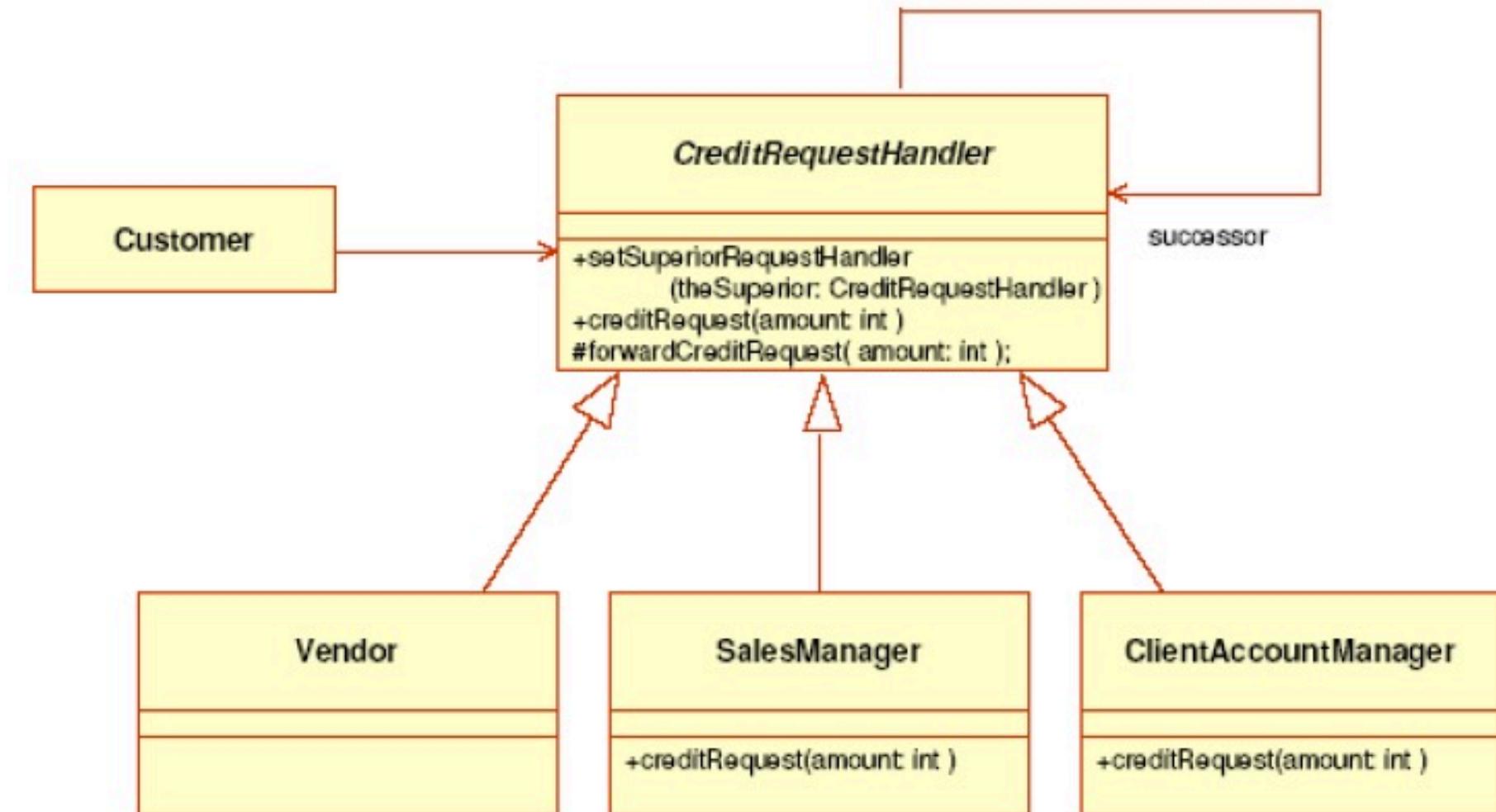
Pattern Chain of Responsibility: struttura



Pattern Chain of responsibility: esempio



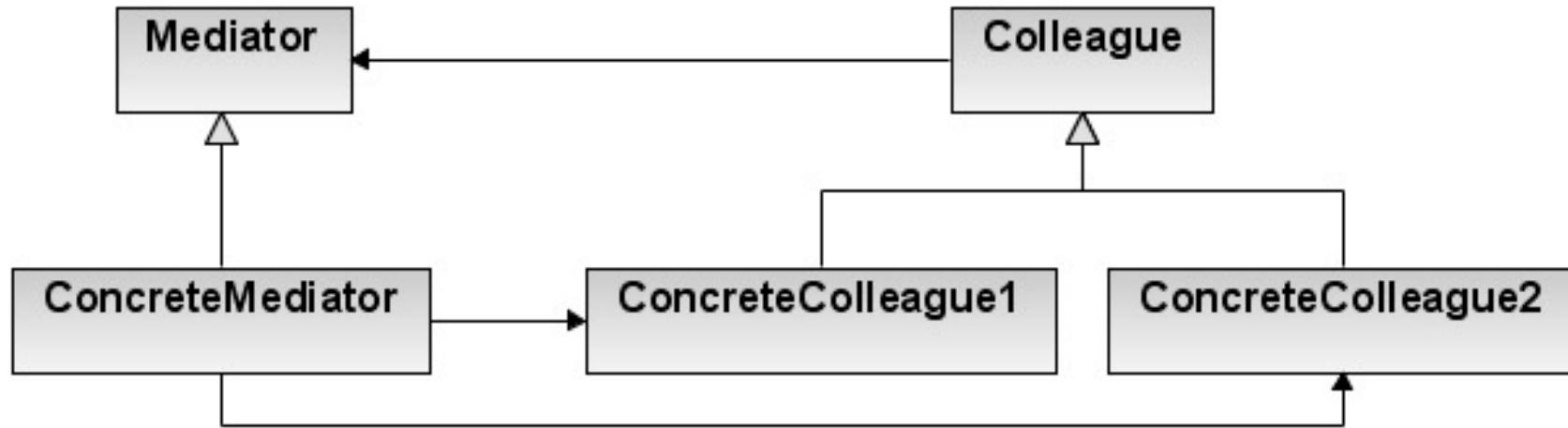
Pattern Chain of Responsibility: esempio



Pattern Mediator: intento

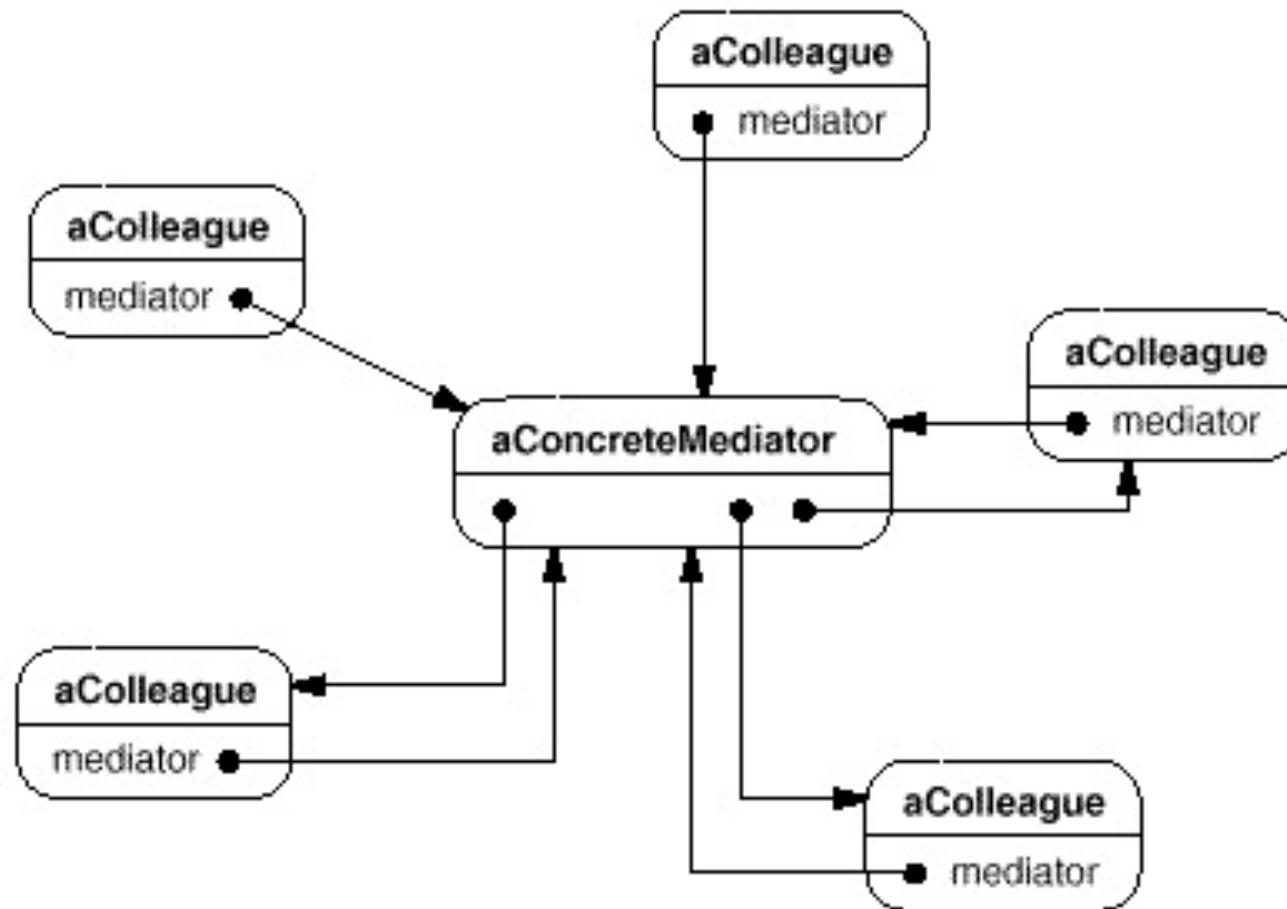
- *Mediator* definisce un oggetto che incapsula i meccanismi di interazione di un insieme di oggetti.
- *Mediator* promuove il disaccoppiamento evitando che gli oggetti interagiscano direttamente, e permette di riprogrammare in modo indipendente la loro interazione

Pattern Mediator: struttura

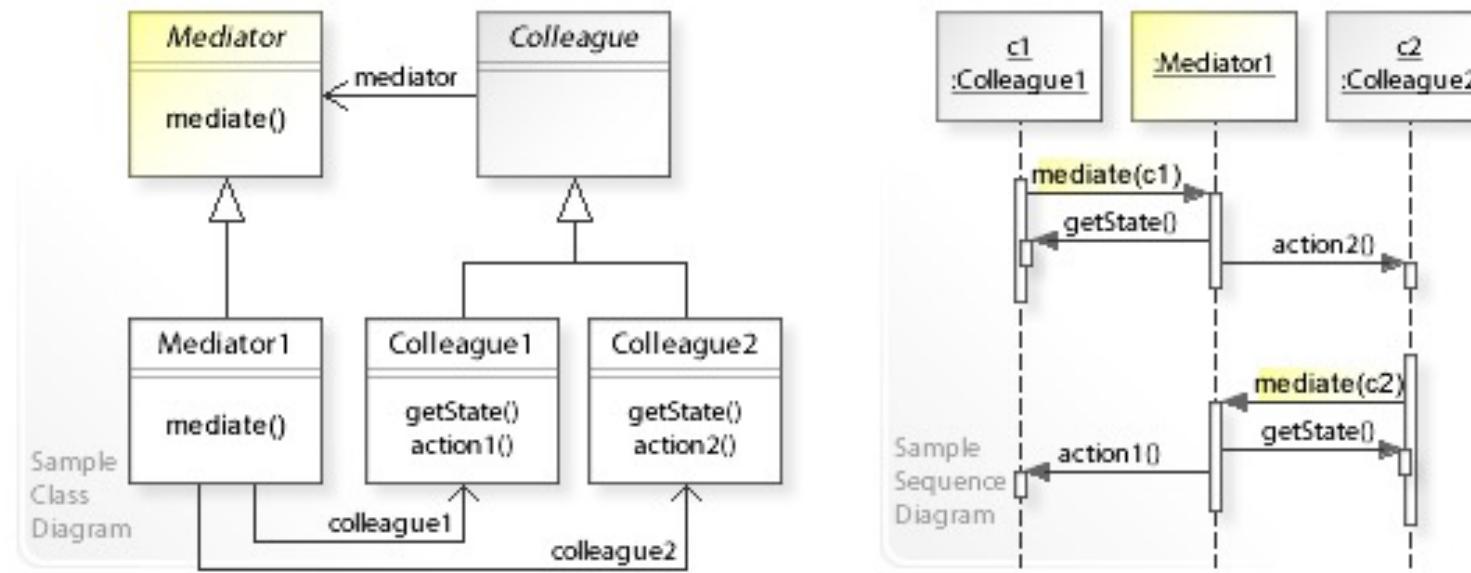


- **Mediator** definisce un interfaccia per comunicare coi colleghi
- **ConcreteMediator**: conosce e gestisce i colleghi concreti
- **Classi collega**: ogni classe conosce comunica col suo mediatore invece di comunicare con un oggetto collega

Pattern Mediator: struttura degli oggetti



Pattern Mediator: comportamento



Pattern Mediator: conseguenze

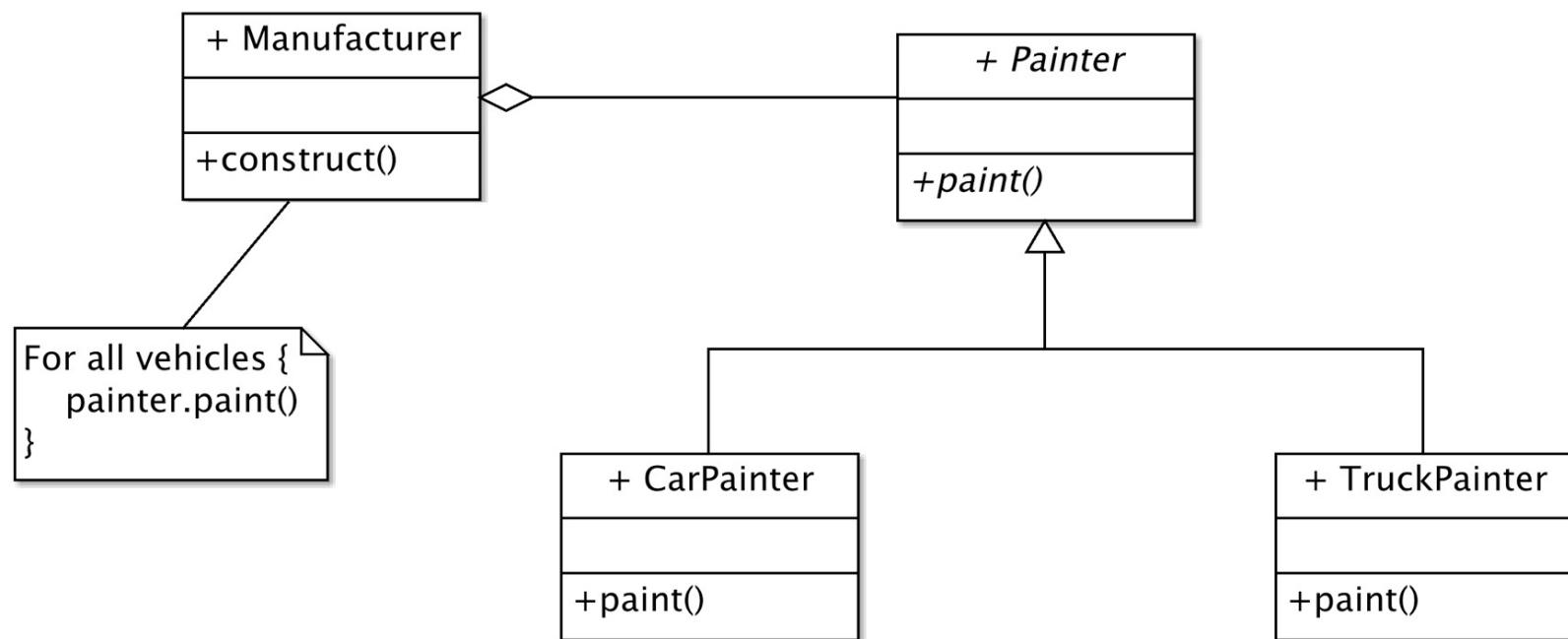
Il pattern Mediator ha i seguenti vantaggi e svantaggi:

- I. **Limita il subclassing.** Un Mediator concentra su di sé il comportamento che altrimenti sarebbe distribuito tra più oggetti.
- II. **Disaccoppia gli oggetti colleghi.**
- III. **Semplifica il protocollo** con cui si usa un oggetto, rimpiazzando interazioni multi-a-molti con interazioni uno-a-molti, che sono più facili da capire ed estendere.
- IV. **Astrae il modo in cui gli oggetti cooperano e si coordinano.**
- V. **Centralizza il controllo.** Il Mediator sostituisce alla complessità dell'interazione la propria complessità: Mediator può diventare un oggetto «monolite» difficile da manutenere.

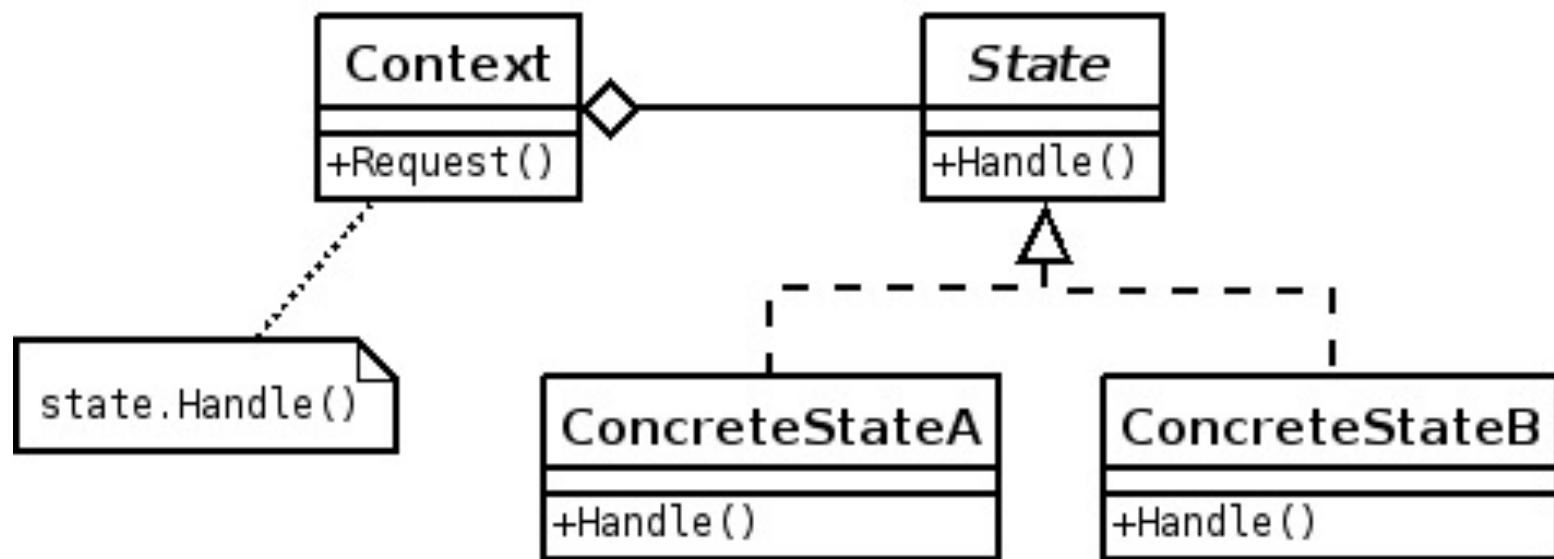
Pattern State: intento

- Permette ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno.
- Sembrerà che l'oggetto modifichi la sua classe

Pattern State: esempio



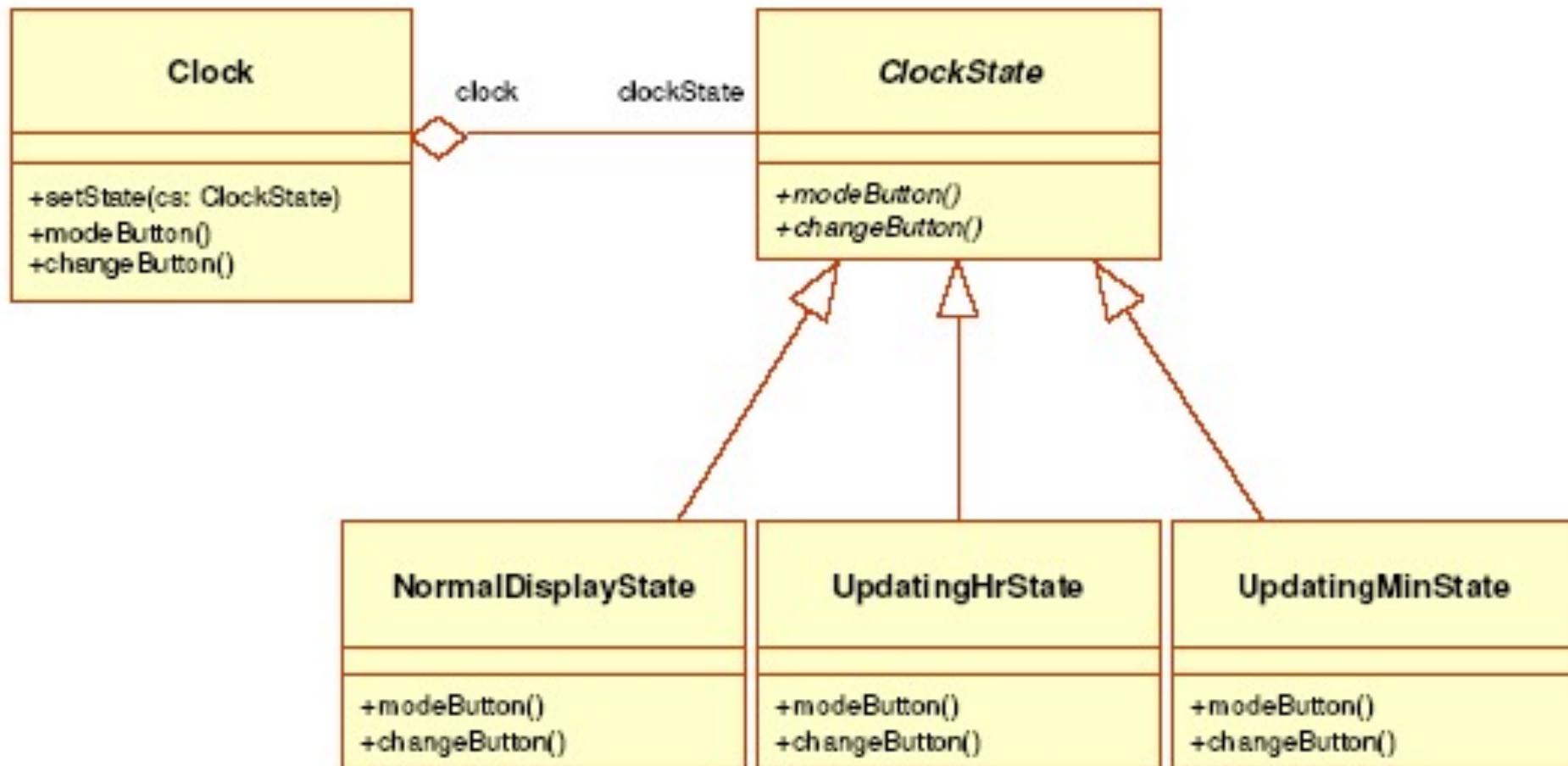
Pattern State: struttura



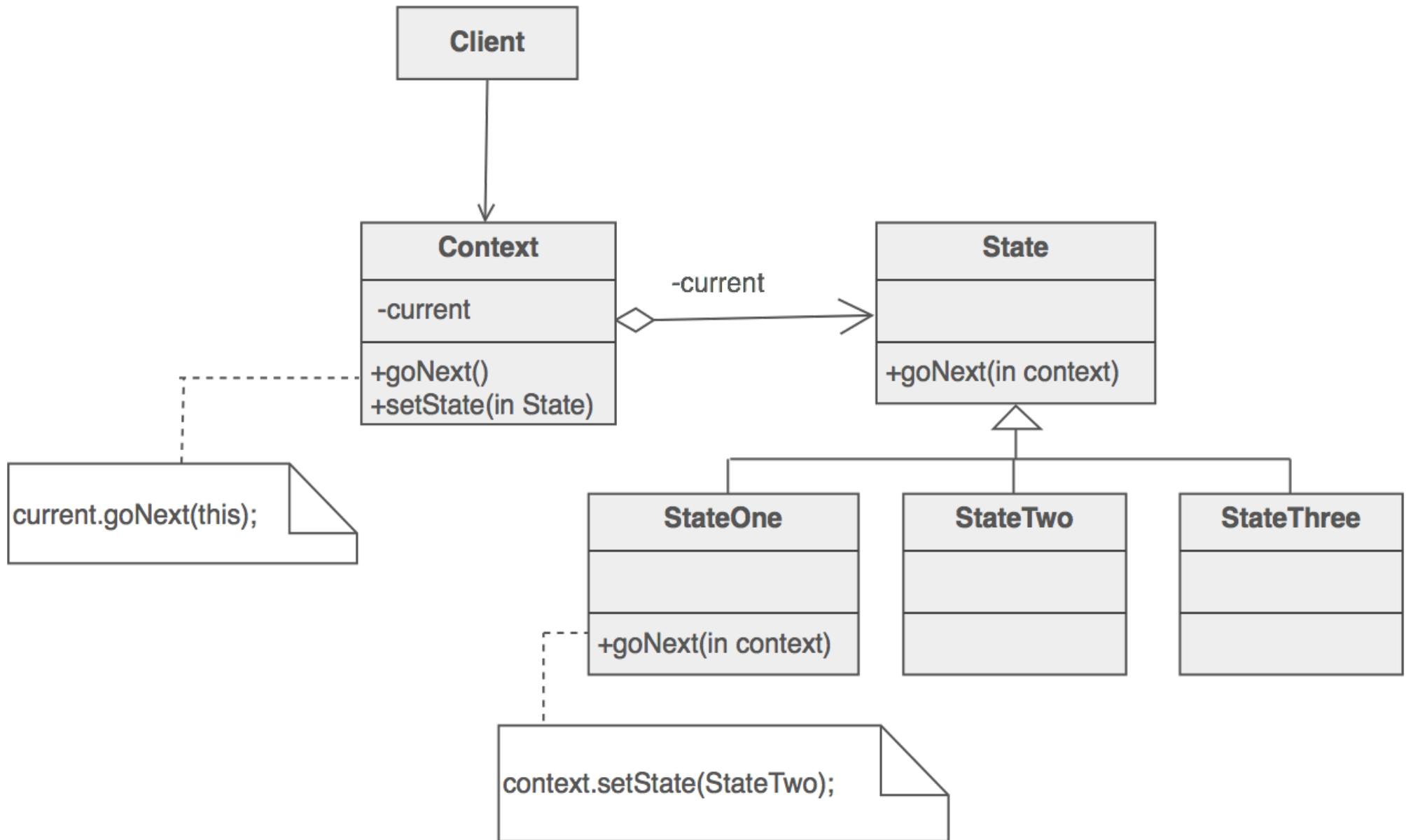
Esercizio

- Modellare il seguente dominio:
 - Un orologio ha due pulsanti: MODE e CHANGE.
 - MODE permette di scegliere tra: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”
 - CHANGE esegue operazioni diverse in base alla modalità:
 - accendere la luce del display, se è in modalità di “visualizzazione normale”
 - incrementare in una unità le ore o i minuti, se è in modalità di “modifica” di ore o di minuti
- Serve un pattern che permetta ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno

Pattern State



Pattern State

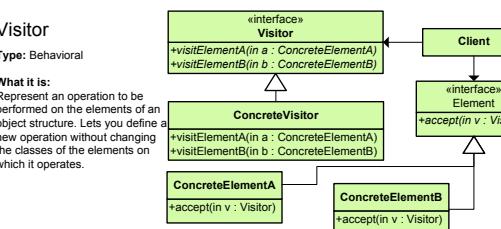
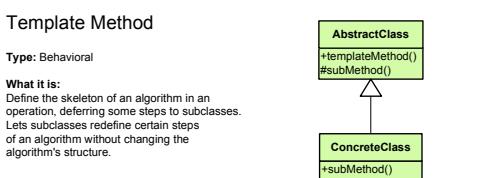
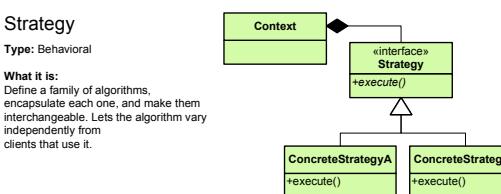
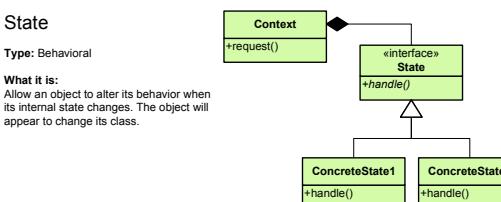
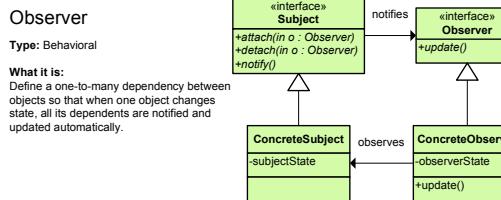
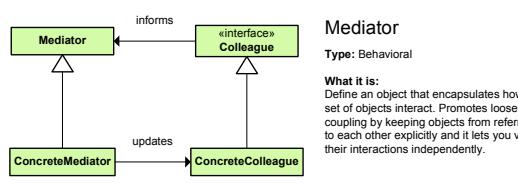
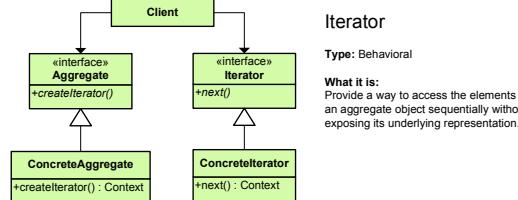
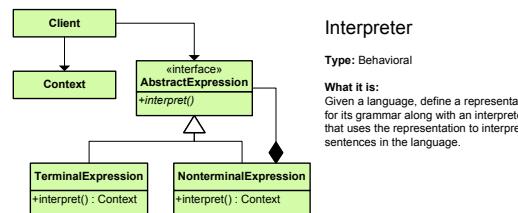
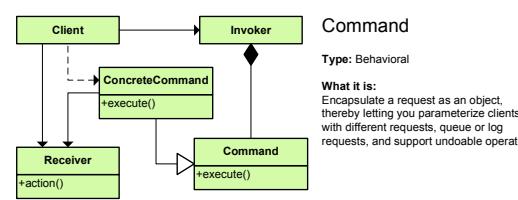
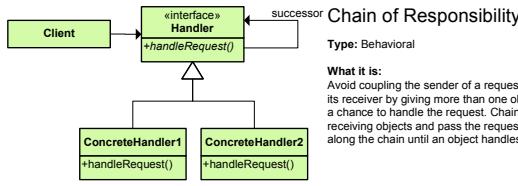
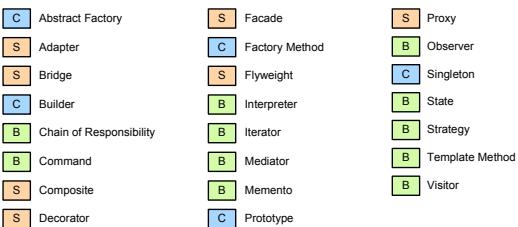


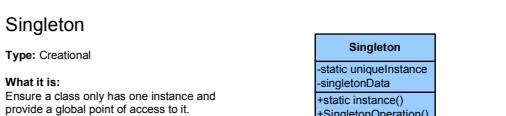
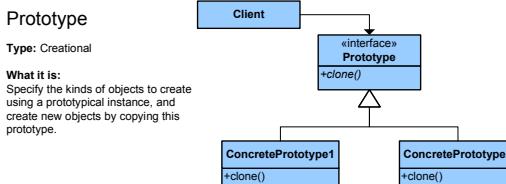
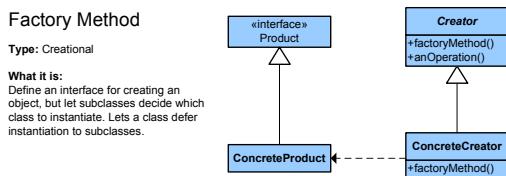
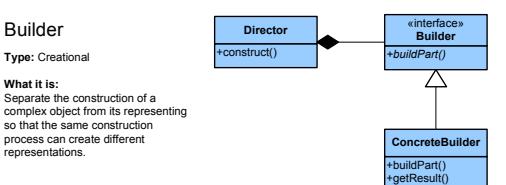
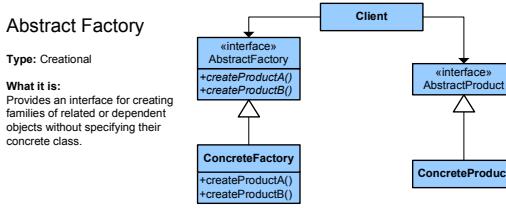
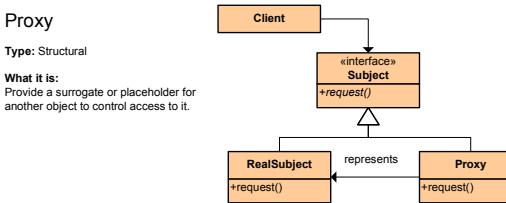
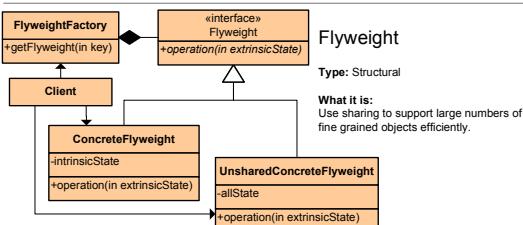
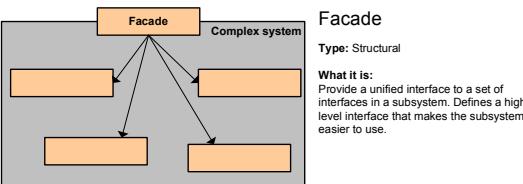
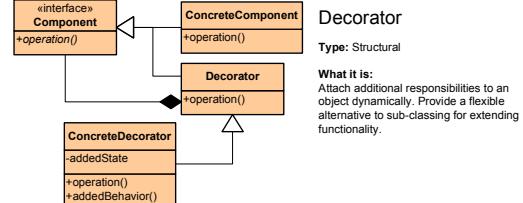
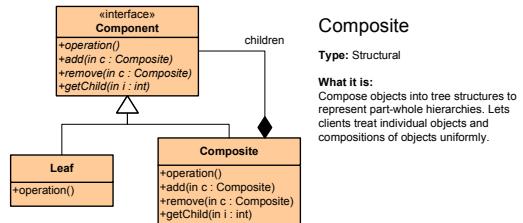
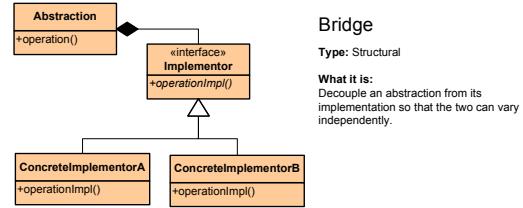
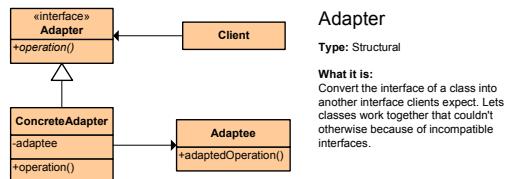
Conclusioni sui design patterns GoF

Intenti dei design patterns di GoF

(i colori corrispondono alle categorie: **creazionali**, **strutturali**, **comportamentali**)

- i. **Abstract factory**: Disaccoppia un client dalle implementazioni di oggetti appartenenti a famiglie diverse
- ii. **Adapter**: Rende interoperabili oggetti con interfacce incompatibili
- iii. **Bridge**: Disaccoppia l'astrazione di un oggetto dalla sua implementazione
- iv. **Builder** Disaccoppia la costruzione di un oggetto dalla sua rappresentazione
- v. **Chain of responsibility** Disaccoppia un client dalla sequenza di oggetti che rispondono alle sue richieste
- vi. **Command** Incapsula una richiesta in un oggetto, e supporta l'undo delle richieste
- vii. **Composite** Permette di trattare allo stesso modo oggetti interi e i loro componenti
- viii. **Decorator** Modifica dinamicamente le responsabilità di un oggetto
- ix. **Façade** Una classe rappresenta l'interfaccia di un sottosistema, disaccoppiandolo dai suoi client
- x. **Factory Method** Crea un oggetto senza specificare la classe dell'oggetto che verrà creato
- xi. **Flyweight** Crea oggetti a grana piccola basati sulla condivisione
- xii. **Interpreter** Valuta le frasi di un linguaggio mappando ogni simbolo in una classe specifica
- xiii. **Iterator** Accede in sequenza gli elementi di un oggetto composito senza conoscere la rappresentazione
- xiv. **Mediator** Semplifica la comunicazione tra più classi, disaccoppiandone le interfacce
- xv. **Memento** Cattura e ripristina lo stato interno di un oggetto, senza violarne l'incapsulamento
- xvi. **Observer** Notifica un evento o cambiamento di stato ad un insieme di oggetti interessati
- xvii. **Prototype** Clona un oggetto a runtime in modo efficiente, per delegazione
- xviii. **Proxy** Crea un oggetto che rappresenta e fornisce accesso ad un altro oggetto
- xix. **Singleton** Crea un oggetto a unica istanza e ne fornisce l'accesso
- xx. **State** Modifica il comportamento di un oggetto quando cambia il suo stato
- xxi. **Strategy** In una classe, incapsula più algoritmi, rendendoli intercambiabili
- xxii. **Template Method** In una superclasse, differisce la specifica di un algoritmo ad una sua sottoclasse
- xxiii. **Visitor** Disaccoppia un algoritmo dalla struttura degli oggetti cui è applicato (principio open/closed)



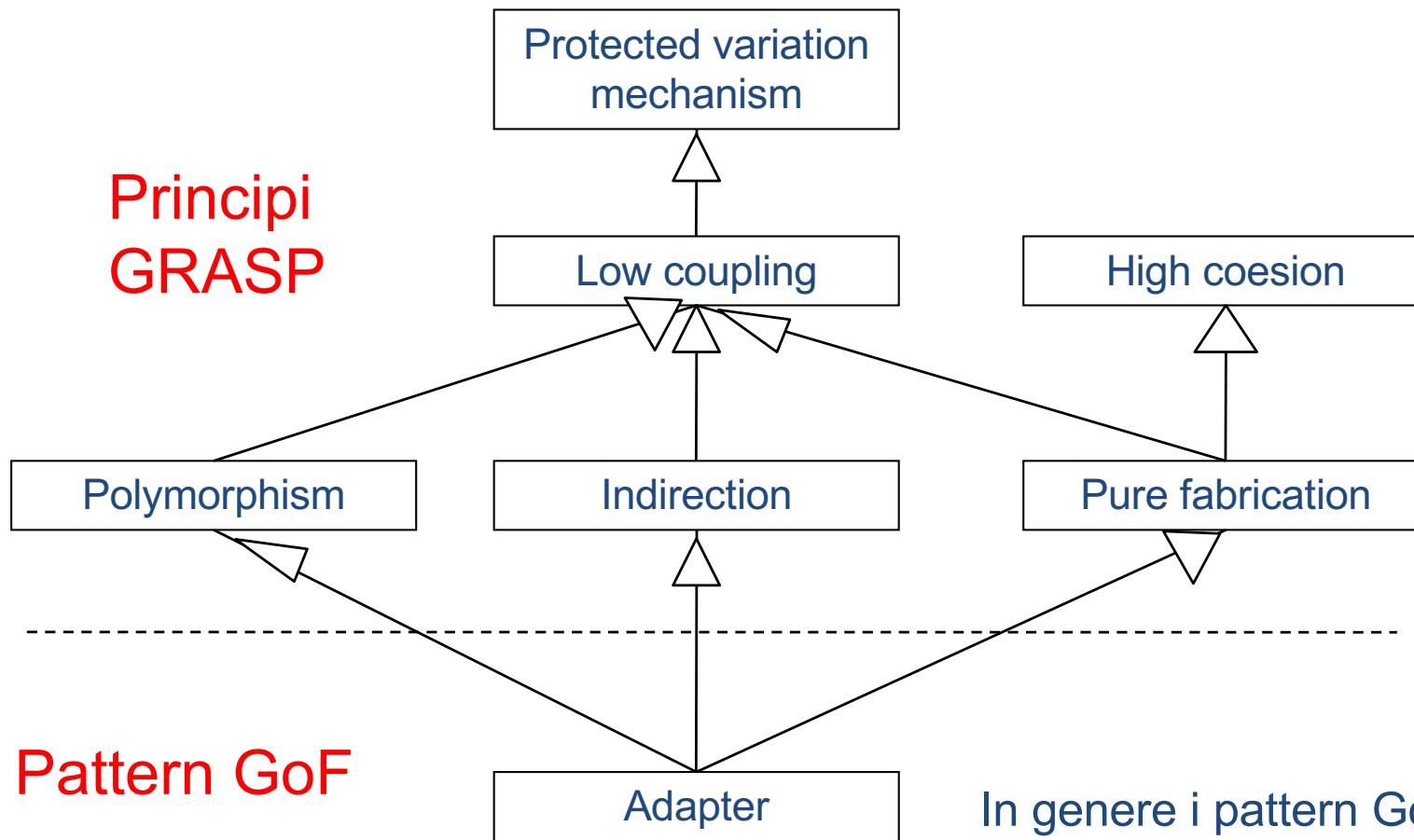


Relazioni tra pattern

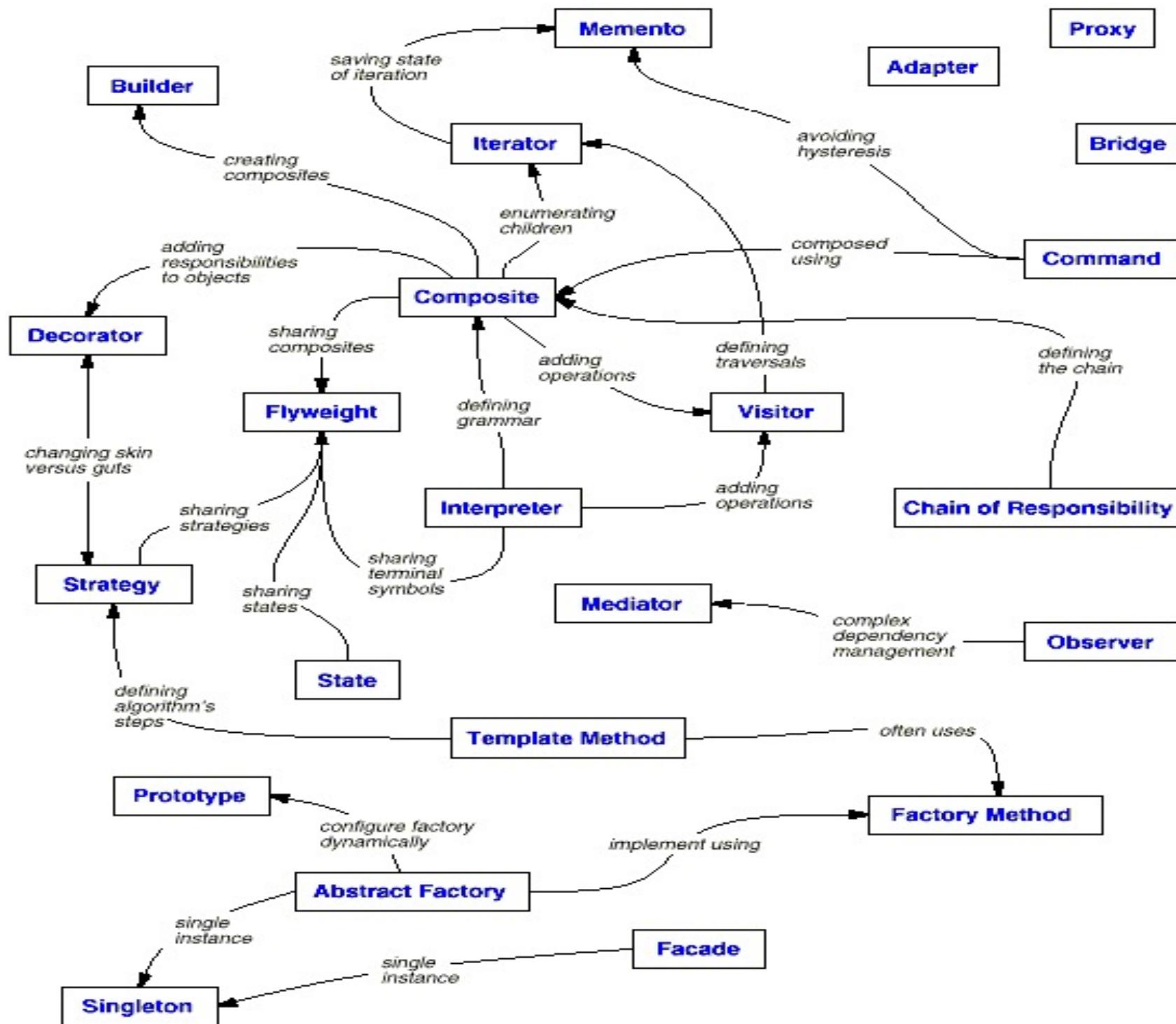
Le prossime diapositive mostrano alcune relazioni tra pattern

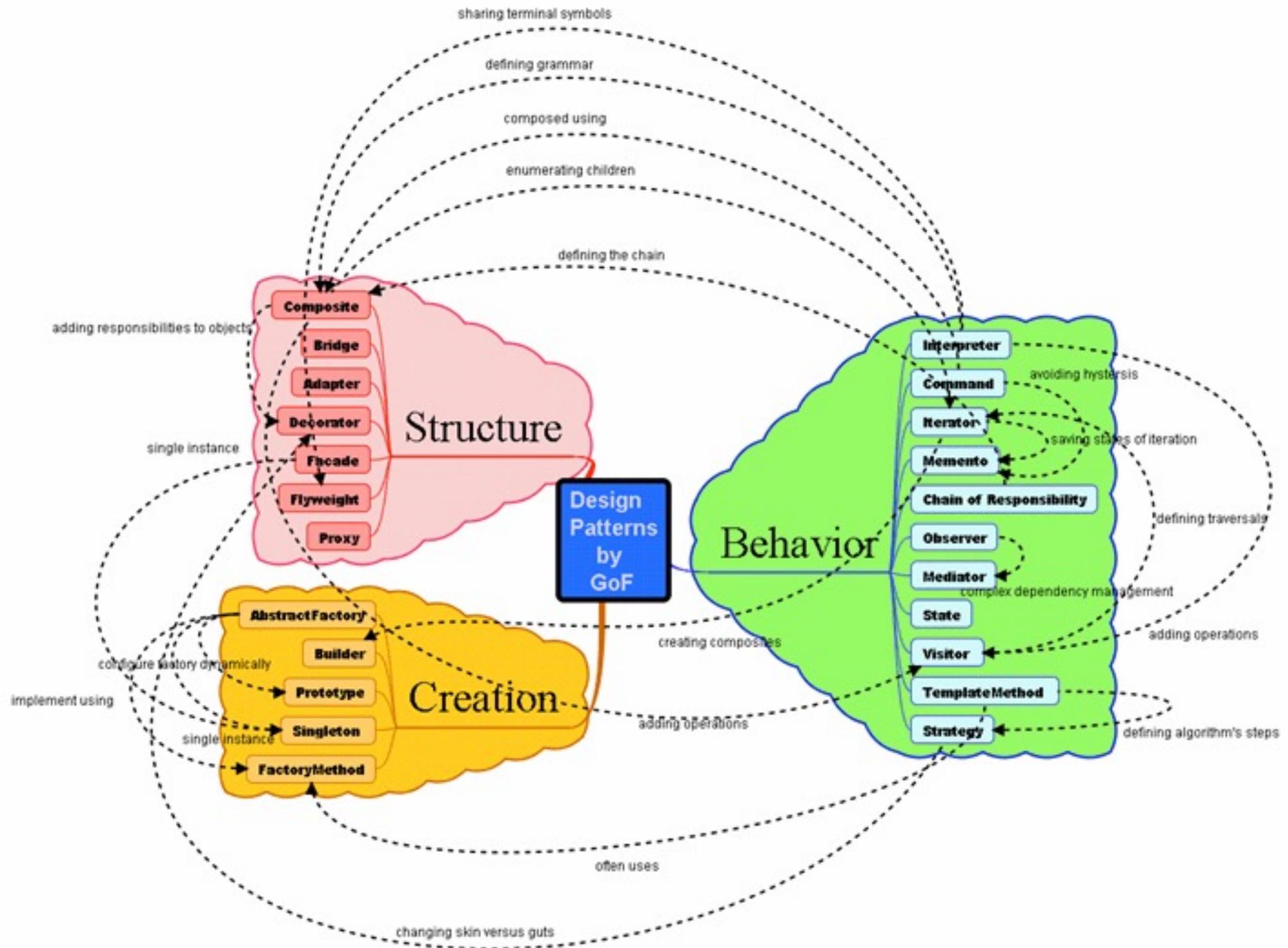
- La prima mostra una relazione tra GRASP e GoF
- La seconda mostra una *mappa concettuale* dei pattern GoF
- La terza mostra una *mappa mentale* (mind map)
 - Fonte: www.stickyminds.com/sitewide.asp?ObjectId=11861&Function=DETAILBROWSE&ObjectType=ART
- Le ultime due mostrano altre relazioni tra pattern

Relazioni tra GRASP e GoF

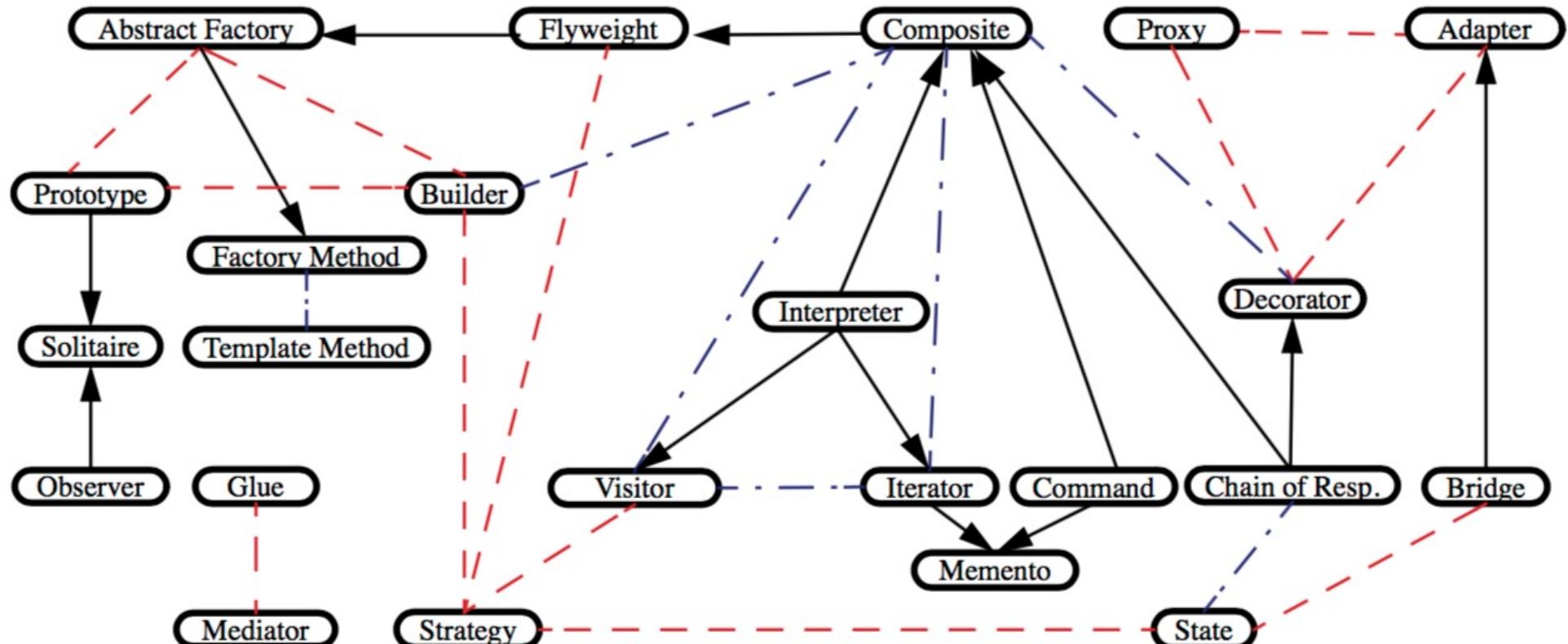


In genere i pattern GoF si
possono motivare in termini di
uno o più GRASP





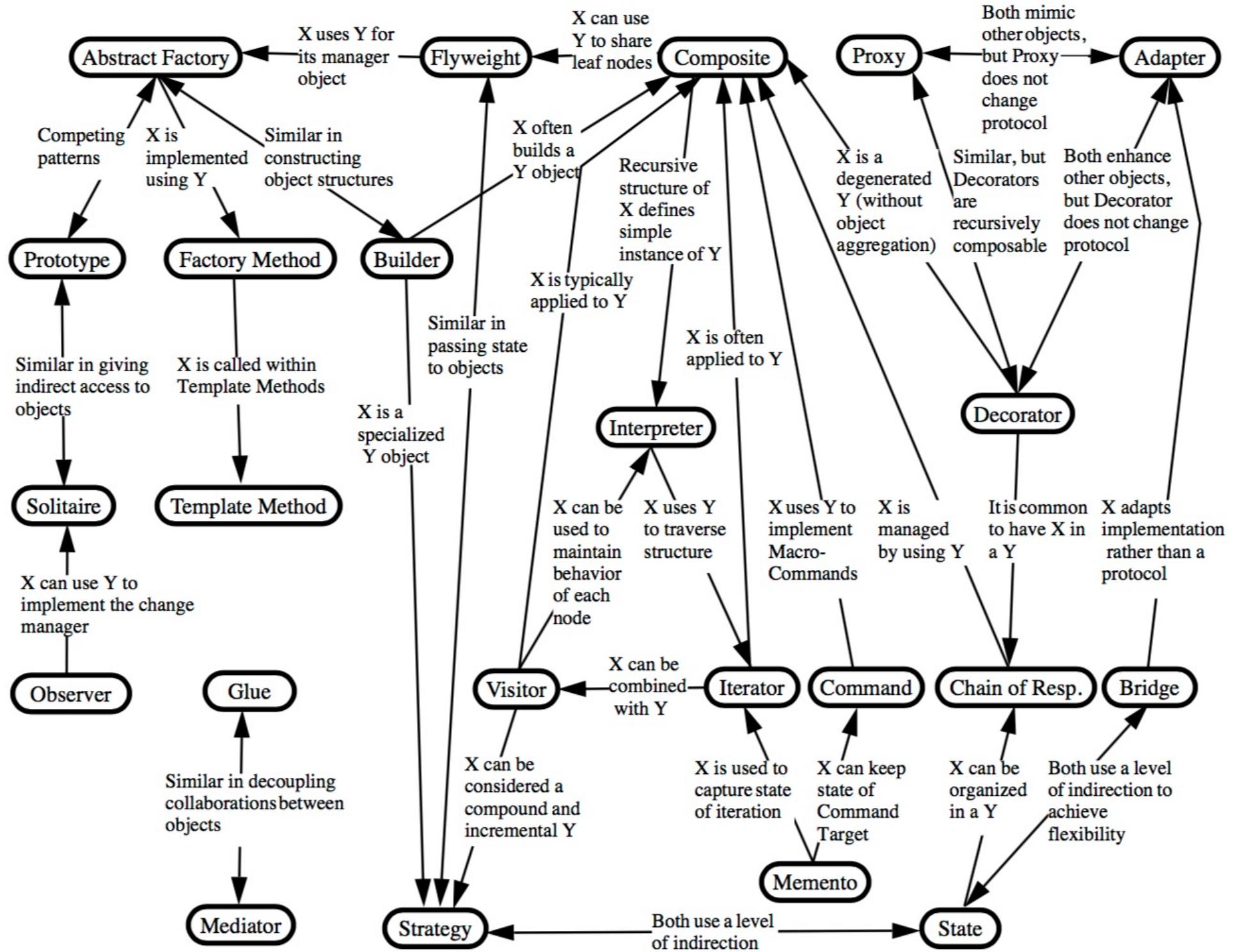
Relazioni tra pattern GoF



X uses Y in its solution

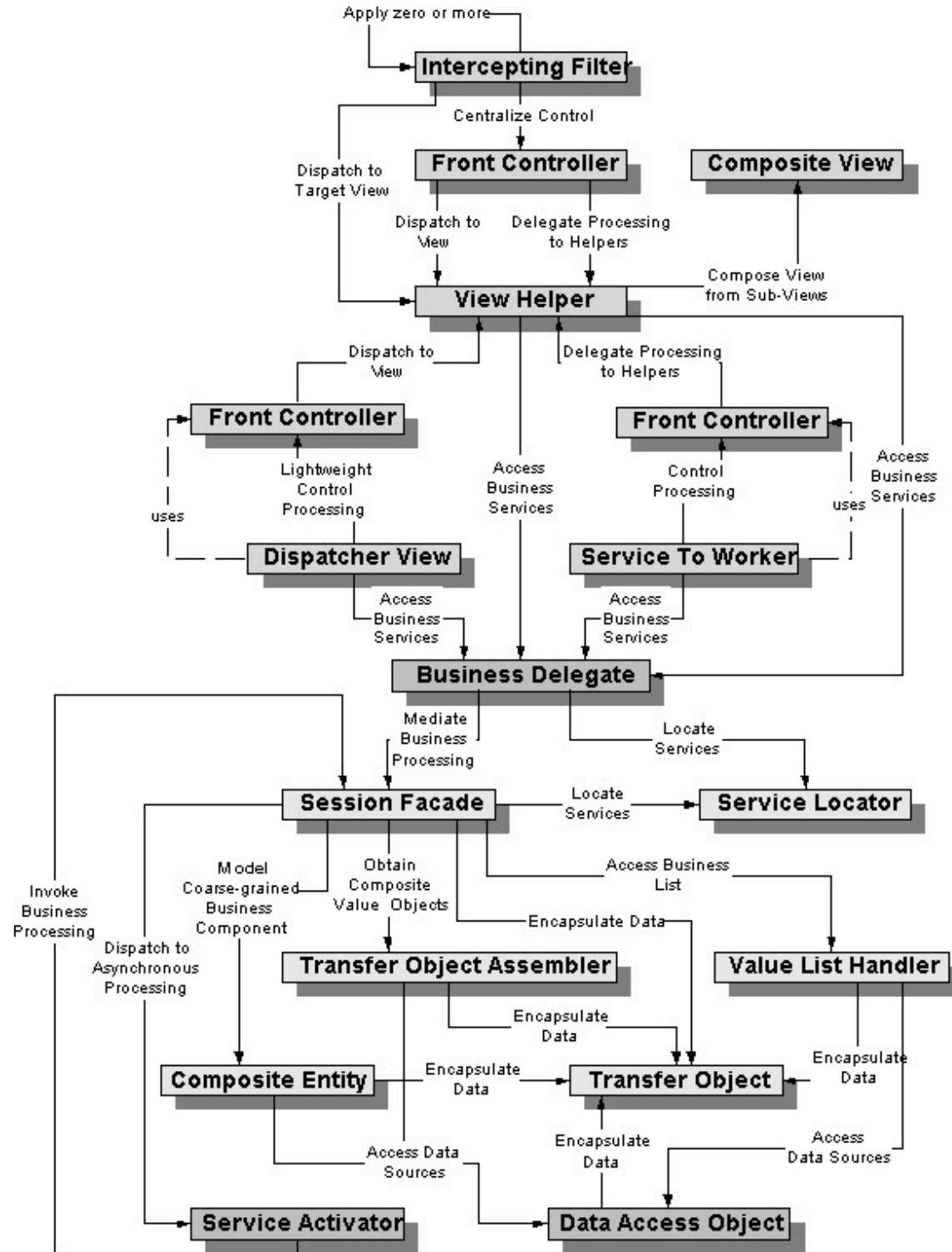
X is similar to Y

X can be combined with Y



Linguaggio dei pattern JavaEE

java.sun.com/blueprints/corej2eepatterns/Patterns/



Conclusioni

I pattern...

- Descrivono astrazioni software riusabili
- Costituiscono un vocabolario comune per i progettisti
- Servono per comunicare sinteticamente principi complessi
- Aiutano a documentare la struttura del sw (architettura)
- Evidenziano le parti critiche di un sistema
- Suggeriscono più di una soluzione

I pattern ...

- **Non** costituiscono una soluzione precisa di problemi progettuali
- **Non** risolvono tutti i problemi progettuali

Non si applicano solo alla progettazione OO, ma anche ad altri domini

Riferimenti

- E.Gamma, R. Helm, R. Johnson, J.M. Vlissides, *Design Patterns: Elementi per il riuso*, Pearson, 2002.
- E.Freeman, *Head First Design Patterns*, 2004
- Holub, Holub on patterns
- Gamma & Beck, Junit - a cook's tour, 1999
<http://junit.sourceforge.net/doc/cookstour/cookstour.htm>
- A.Shvets, *Design patterns explained simply*, 2013
https://sourcemaking.com/design_patterns
- C.Horstmann, *Progettazione del software e Design Pattern in Java*, Apogeo, 2004

Siti utili

- **Tutorial**
 - www.tutorialspoint.com/design_pattern/index.htm
- **Esempi sui design patterns**
 - www.headfirstlabs.com/books/hfdp/sourcemaking.com/design_patterns_sourcemaking.com
 - msdn.microsoft.com/en-us/library/bb190165.aspx
- **Cataloghi on line di pattern**
 - hillside.net
 - www.c2.com/cgi/wiki?WikiPagesAboutWhatArePatterns
 - www.dofactory.com/Patterns/Patterns.aspx
- <http://www.dofactory.com/net/design-patterns>
 - www.vincehuston.org/dp_patterns.architecture.michaelkappel.com
- **Design patterns in Java**
 - www.fluffycat.com/java/patterns.html
- **Antipattern** sourcemaking.com/antipatterns
- medium.com/young-coder/is-it-time-to-get-over-design-patterns-8851864a6834

Domande?

