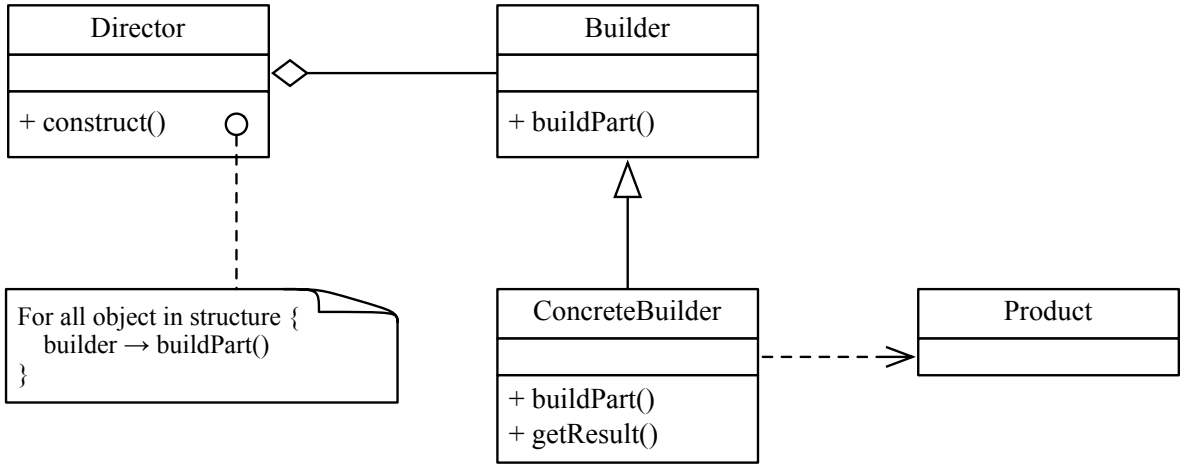
PATTERN CREAZIONALI

Abstract Factory: utilizzata per creare oggetti senza conoscerne l'implementazione concreta.
Procedimento: creo classi astratte e classi concrete (che implementano quelle astratte).
Nel "Client" (main) creo oggetti astratti in modo da svincolarli dalle implementazioni (mi interessa solo cosa sa fare una classe e non come lo fa).

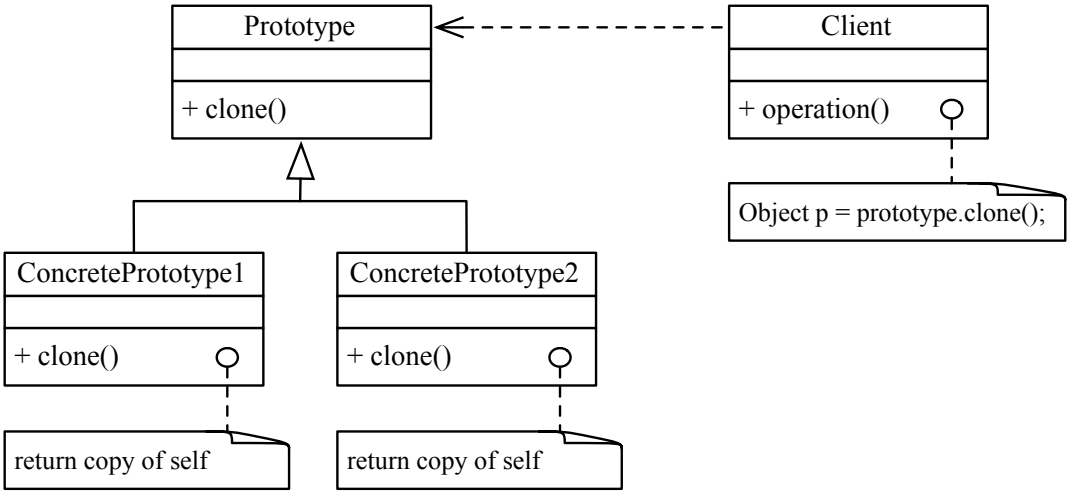
Builder: usato per creare oggetti complessi indipendentemente dalla loro rappresentazione interna. Consente di variare la rappresentazione interna del prodotto, isolare il codice per la costruzione e la rappresentazione e controllare in modo preciso il processo di costruzione.
Schema base:



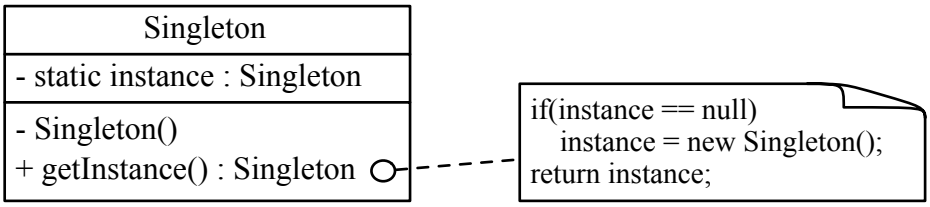
- Builder: specifica l'interfaccia astratta che crea le parti dell'oggetto.
- ConcreteBuilder: costruisce e assembla le parti del prodotto implementando l'interfaccia Builder.
- Director: costruisce un oggetto utilizzando l'interfaccia Builder (quindi indipendentemente dall'implementazione del Builder).
- Product: rappresenta l'oggetto complesso.

Prototype: usato per creare nuovi oggetti, clonando uno iniziale detto prototipo.
Conseguenze nell'usarlo:
- Indipendenza dal metodo d'instanziazione.
- Modularità a run-time.
- Definire nuovi oggetti modificando valori.
- Difficoltà legate alla clonazione.

Schema base:



Singleton: ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.
Schema base:

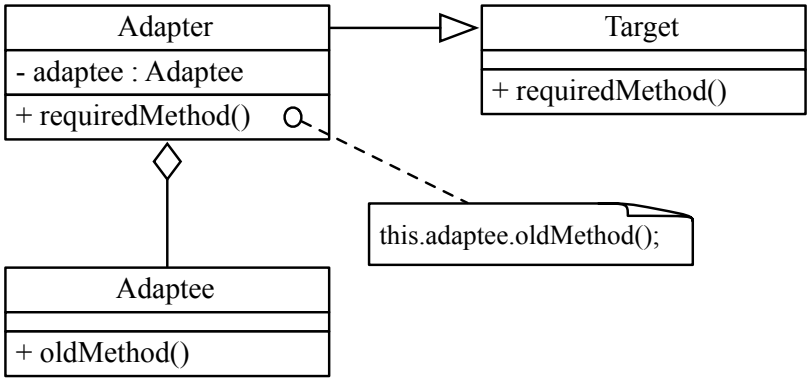


PATTERN STRUTTURALI

Adapter:
Consente ad un oggetto di essere utilizzato mediante un'interfaccia che non implementa.
L'adapter è uno dei pattern più usati:
- consente a oggetti diversi di essere utilizzati insieme anche se le rispettive interfacce non sono compatibili
- aumenta il riuso delle classi perché consente di utilizzarle in situazioni non previste inizialmente senza doverle modificare.

Un adapter viene usato:
- quando si vuole utilizzare una classe che non offre un'interfaccia corretta, senza modificarla
- per creare una classe che utilizza i servizi di un'altra classe di cui non è ancora nota l'interfaccia.

Schema base:

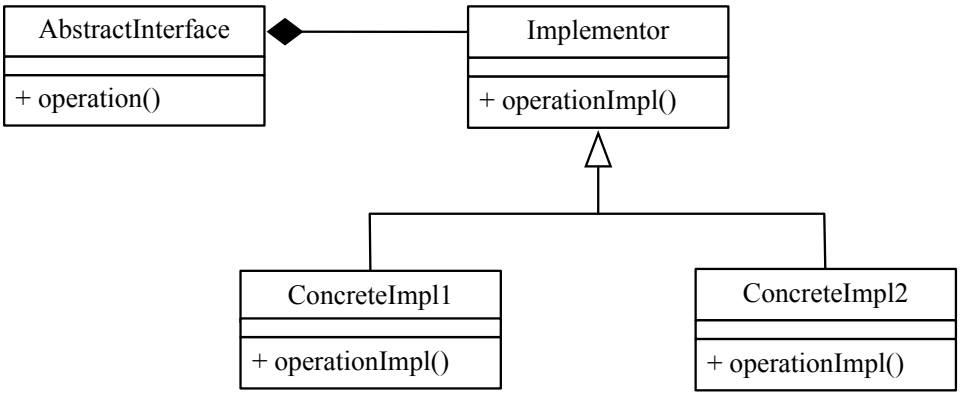


- Adaptee: definisce l'interfaccia che ha bisogno di essere adattata.
- Target: definisce l'interfaccia che usa il Client.
- Client: collabora con gli oggetti in conformità con l'interfaccia Target.
- Adapter: adatta l'interfaccia Adaptee all'interfaccia Target.

Bridge:
Disaccoppia un'astrazione dalla sua implementazione:
- la classe di implementazione e la classe degli utilizzatori possono variare indipendentemente.

Un bridge viene usato quando:
- si vuole evitare di legare un'astrazione ad una sola implementazione
- ogni cambiamento di un'implementazione non è influente sugli utilizzatori
- per nascondere dettagli implementativi

Schema base:



Composite:

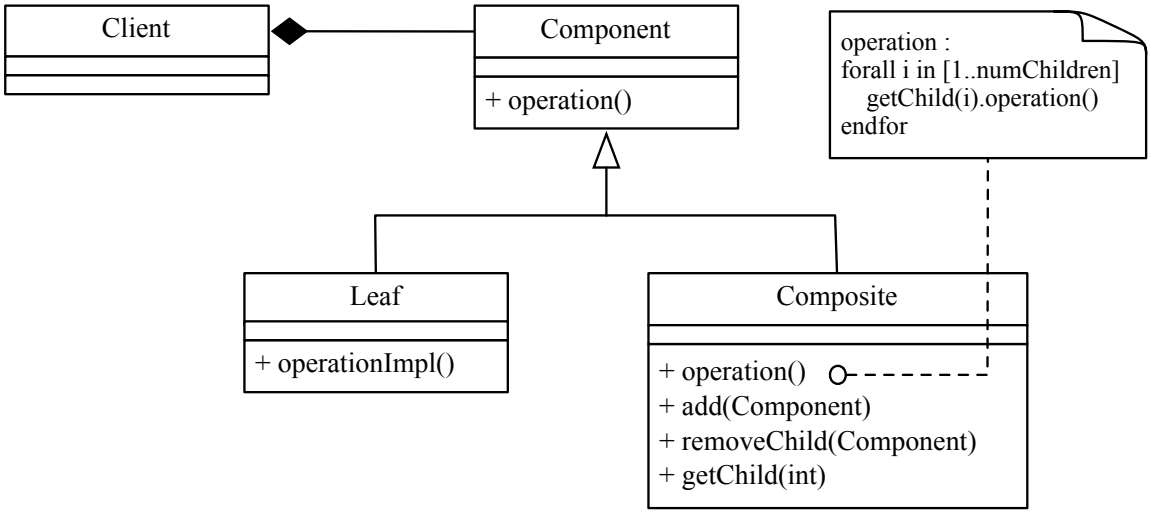
Consente ad oggetti utilizzatori di creare strutture ad albero in modo che sia possibile trattare in modo uniforme:

- oggetti foglia
- oggetti contenitore

Un composite viene usato quando:

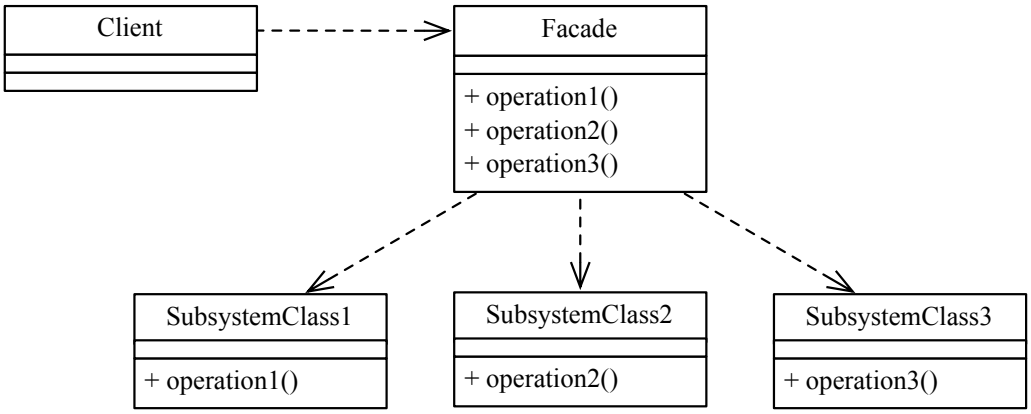
- si vuole rappresentare gerarchie di oggetti
- si vuole consentire di ignorare le differenze tra oggetti foglia e oggetti contenitore

Schema base:



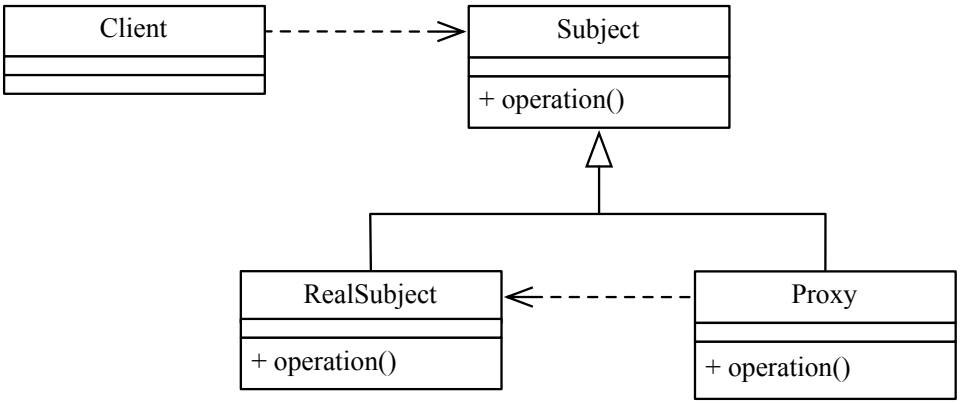
- Client: manipola gli oggetti attraverso l'interfaccia Component.
- Component: dichiara l'interfaccia per gli oggetti nella composizione, per l'accesso e la manipolazione di questi, imposta un comportamento di default per l'interfaccia comune a tutte le classi e può definire un'interfaccia per l'accesso al padre del componente e la implementa se è appropriata.
- Composite: definisce il comportamento per i componenti aventi figli, salva i figli e implementa le operazioni ad essi connesse nell'interfaccia Component.
- Leaf: definisce il comportamento degli oggetti primitivi, cioè delle foglie.

Façade:
Offre un'interfaccia uniforme ad un insieme di interfacce correlate (prendo due o più interfacce e le uso per crearne una nuova).
Una Façade viene utilizzata quando:
- si vuole offrire un'interfaccia uniforme ad un sotto-sistema complesso
- si vuole disaccoppiare un sotto-sistema dai suoi utilizzatori, riducendo le interdipendenze.



Proxy:
Un proxy è una classe che funziona come interfaccia per qualcos'altro.
I proxy hanno diversi usi:
- per controllare l'accesso alle risorse (per implementare politiche di sicurezza)
- per implementare una gestione sofisticata di un oggetto (accesso sincronizzato, persistenza)
- per implementare oggetti remoti (il proxy e l'oggetto reale sono su due host diversi).

Schema base:



- Subject: entrambi (RealSubject e Proxy) implementano l'interfaccia Subject. Questo permette al Client di trattare il Proxy come se fosse il RealSubject.
- RealSubject: è generalmente l'oggetto che fa più lavoro concreto. Il Proxy controlla l'accesso ad esso.
- Proxy: il Proxy tiene un riferimento al Subject così può inoltrare richieste al Subject quando è necessario.
(il Proxy spesso istanzia o gestisce la creazione del RealSubject).

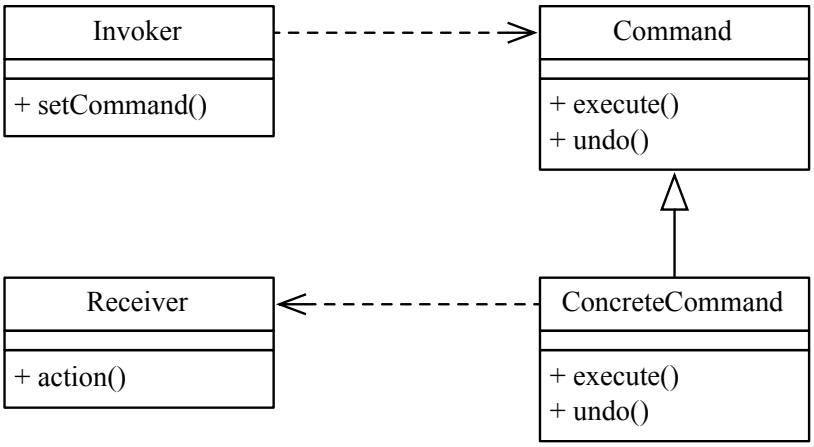
PATTERN COMPORTAMENTALI

Command:

Incapsula una richiesta in un oggetto:
- consente a molte sorgenti di generare richieste e le gestisce secondo una politica
- consente ad un oggetto di modificare il proprio comportamento in base alle richieste

Un command viene usato quando:
- si vuole consentire di creare una richiesta in molti modi diversi
- si vuole consentire di fare undo delle richieste

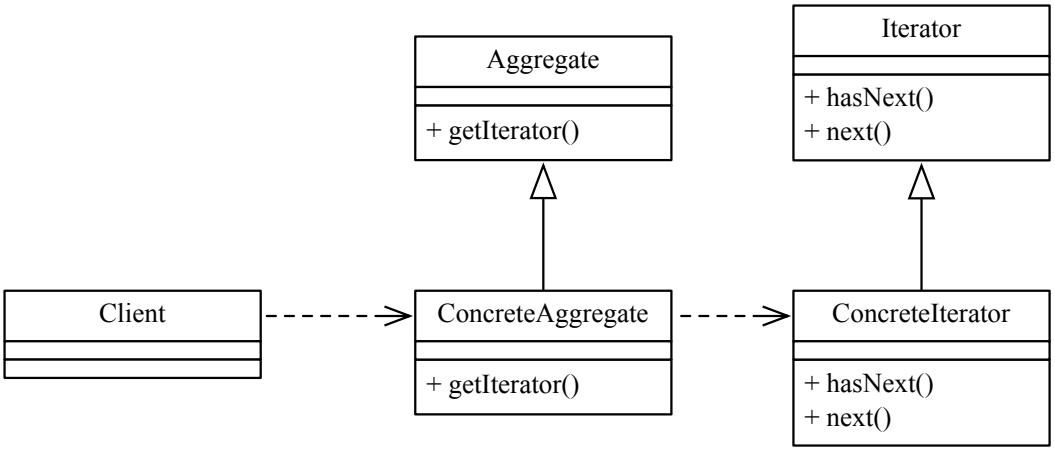
Schema base:



- Receiver: sa come eseguire il lavoro necessario per effettuare la richiesta. Qualsiasi classe può essere un ricevente.
- Command: dichiara un'interfaccia per tutti i comandi. Il metodo execute() è quello che permette di svolgere un azione.
- ConcreteCommand: definisce un legame tra azione e Receiver. L'Invoker chiederà l'esecuzione di execute() del ConcreteCommand il quale chiamerà una o più azioni del Receiver.
- Invoker: ha un Command al quale chiede qualche volta di eseguire i suoi metodi (in particolare, il metodo execute()).

Schema - Command (esteso):
- Inizialmente ho "ElettronicDevice" e la sua implementazione "Television". Decido di applicare il pattern command quindi devo: creare una interfaccia "Command" con una implementazione almeno per ogni metodo di "ElettronicDevice" (che sono 4), poi creo l'invoker (DeviceButton) che è la classe che utilizza i metodi generali del "Command".
- Dopo aver fatto ciò per la Television, decido di creare un nuovo "ElectronicDevice", la "Radio". Si osserva che mi basta creare (in azzurro nello schema) l'implementazione, aggiungerla e basta (le altre classi, escluso il Client, non vengono toccate). Aggiungendo la "Radio" dovrei creare anche per lei i vari "Command" ma noto che in realtà posso usare quelli della TV anche per la "Radio" dato che essi utilizzano un "ElettronicDevice" (e anche la "Radio" è un "ElettronicDevice").
(dato che nei Command della TV ho usato un "ElettronicDevice" e non una "Television", sarebbe stato meglio cambiare i nomi delle classi togliendo la parola "TV", dato che sono generiche).
- Decido di creare anche un novo comando che spegne tutti i dispositivi "TurnItAllOff". Anche in questo caso, mi basta creare un'implementazione di "Command" e aggiungerla alle altre senza bisogno di toccare le altre classi.

Iterator:
Implementa un meccanismo di accesso ad oggetti aggregati in modo sequenziale:
- un iteratore naviga sequenzialmente un oggetto aggregato.
Un iteratore viene utilizzato:
- per accedere al contenuto di un oggetto aggregato in modo indipendente da come i componenti sono memorizzati nell'oggetto aggregato
- quando si vuole offrire diverse politiche di attraversamento di un oggetto aggregato.
Schema base:



- Aggregate: fornisce un'interfaccia comune per gli aggregati, a portata di mano per il cliente; si disaccoppia il cliente dalla collezione di oggetti.
- ConcreteAgregate: ha una collezione di oggetti e implementa il metodo che restituisce un iteratore per la sua collezione.
(ogni ConcreteAggregate è responsabile dell'istanziazione di un iteratore concreto che può iterare sulla sua collezione di oggetti).
- Iterator: fornisce l'interfaccia che tutti gli iteratori devono implementare e una serie di metodi per attraversare gli elementi di una collezione.
- ConcreteIterator: deve gestire la posizione corrente dell'iterazione.