Design Pattern 3

Introduzione

Obiettivi:

- Applicare propriamente i pattern del catalogo GoF

Questa sezione rappresenta il naturale conseguimento del documento *Design Pattern 2*, in cui sono elencati ulteriori pattern *comportamentali*, *creazionali* e *strutturali* del *catologo GoF*.

Singleton (Creational Pattern)

Problema
Come garantire che sia creata una sola istanza di una classe e fare in modo che sia condivisibile dagli elementi del modello?

Singleton è un pattern creazionale adoperato per istanziare oggetti in maniera tale che siano rispettati i principi di qualità del software, cercando di manipolare al meglio le dipendenze tra le classi. La volontà di implementare un meccanismo simile, avviene qualora differenti elementi del modello eseguano lo stesso processo esecutivo, da cui saranno restituiti certi *behaviors* oppure dati in *output*.

Per cui l'intento promuove la creazione di una singola istanza della classe software analizzata, affinchè essa possa essere condivisa tra le entità del sistema elaborato. Tendenzialmente è dichiarata come una variabile *privata*, in quanto lo stesso *costruttore* è *protetto* oppure *privato*, a cui si associa una funzione *pubblica* che incapsula al suo interno l'intero codice di inizializzazione e provvede all'accesso all'instanza in questione.

Soluzione
Creare una *singola* istanza della classe software analizzata e provvedere ad un *unico punto di accesso*.

Generalmente l'utilizzo di *Singleton* avviene solo se soddisfatti tre criteri comportamentali, suddivisi in:

- Sviluppo progettuale secondo il principio *lazy initialization*, ossia posticipare la creazione di un oggetto fino al momento in cui non sia realmente necessario
- Non sia possibile attribuire ad alcun elemento del modello la *responsabilità* della creazione dell'istanza
- Nonostante sia sviluppato un metodo pubblico per rendere accessibile l'oggetto della classe software, non deve essere garantita la possibilità che la funzione sia richiamabile a livello globale

Proxy (Structural Pattern)

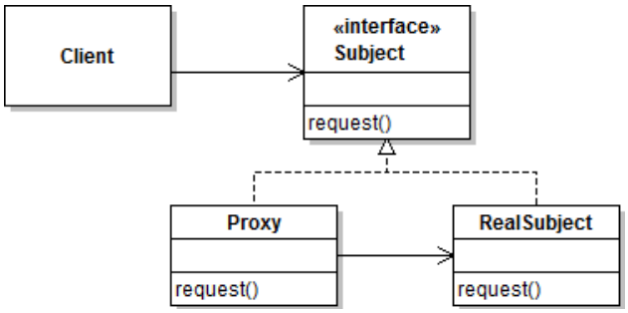
Problema
Come intercettare l'accesso di certi elementi del modello ad oggetti di ulteriori classi soft-

ware?

In alcuni casi potrebbe essere desiderata la funzionalità che provveda ad accertarsi di quali istanze di classi software siano momentaneamente adoperate, oppure per controllare quali referenze, incapsulate all'interno dell'oggetto, siano attuate. Come già ribadito in capitoli precedenti, è possibile porre livelli di astrazione che operano come mediatori tra le entità del design project, caratterizzando la netta separazione tra *high level classes* e *low level classes* di un qualsiasi sistema software sviluppato. Riassumendo quanto detto è necessario introdurre un *intermediario* tra il fornitore di funzionalità, legati alla business logic, e il richiedente, elemento affine all'utente finale.

Soluzione
Realizzare un *segnaposto* per controllare l'accesso agli oggetti interessati.

Caso di studio



La raffigurazione rispetta la soluzione del pattern *Proxy*, in cui si visualizzano un totale di quattro entità. Nell'esempio proposto si contraddistinguono:

- *Client* è una *high level class*, per cui affine all'impiego da parte di utenti finali
- *Subject* indica il livello di astrazione richiesto per soddisfare il *Dependency Inversion Principle*, ossia l'isolamento di classi legate alla logica algoritmica rispetto a *HLC*
- *Proxy*, mediatore che prevede la concretizzazione della struttura *request-response*, ossia rimane in attesa di ricevere *richieste* affinchè possa restituire *risposte inerenti*
- *RealSubject* rappresenta la classe che sviluppa la *business logic*, affinchè sia possibile soddisfare la richiesta implementativa dei requisiti funzionali. Si ricordi che *low level classes* mantengono un livello di astrazione piuttosto elevato dal domain model, in maniera tale che possano eseguire qualsiasi tipologia di funzionalità, indipendentemente dal tipo di oggetto passato come input

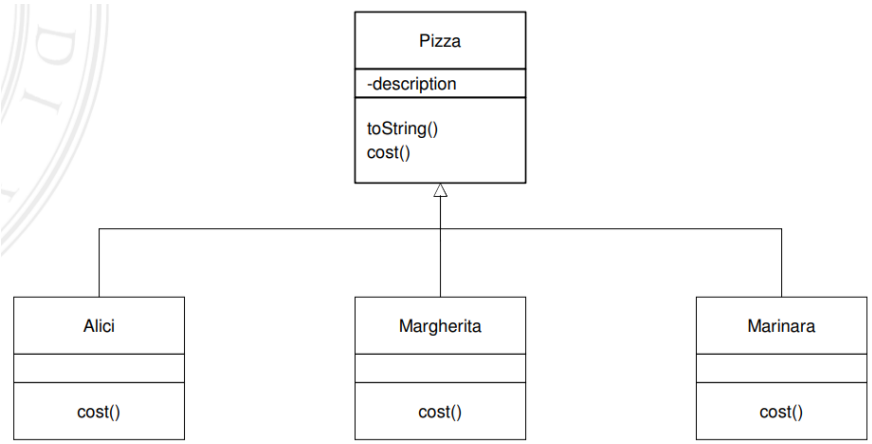
In questo contesto è la classe *Proxy* che promuove lo smistamento di richieste e risposte, affinchè non siano violati i principi *SOLID*, ovviando ad una scarsa qualità del codice, e attuando un meccanismo che possa soddisfare i requisiti imposti dalla classe cliente.

Concludendo, l'interfaccia potrebbe essere sostituita mediante una *pure fabrication*, soddisfando il principio *DIP*, grazie alla realizzazione di una specifica *factory* implementata per la creazione di istanze, in questa tematica di tipologia *Proxy*, le quali saranno conseguentemente restituite.

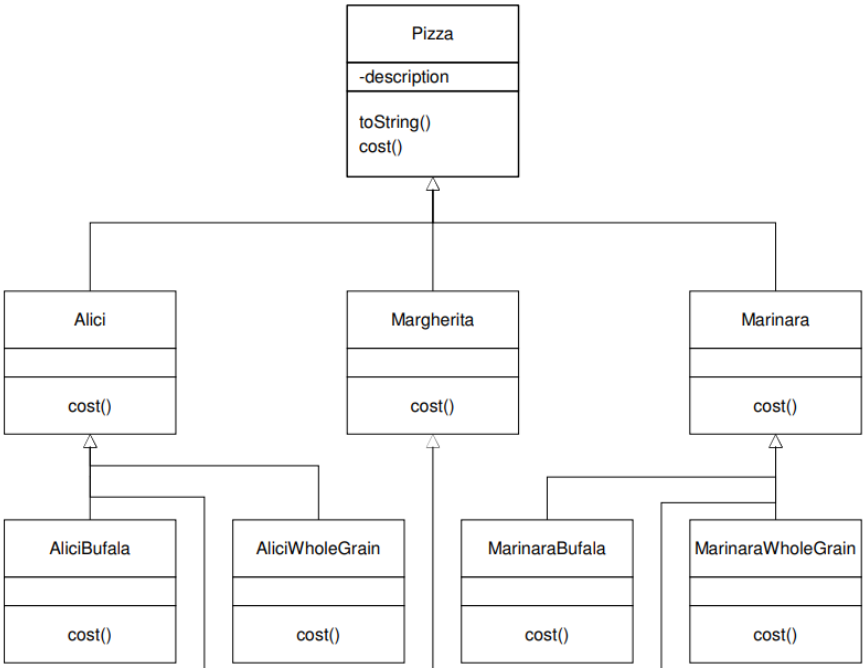
Decorator (Intermezzo)

Di seguito è illustrato un esempio che possa raffigurare certi contesti in cui il solo utilizzo di principi legati al paradigma degli oggetti non sia sufficiente per rispondere a caratteristiche comportamentali, oltre a diminuire la qualità del codice implementato.

Caso di studio

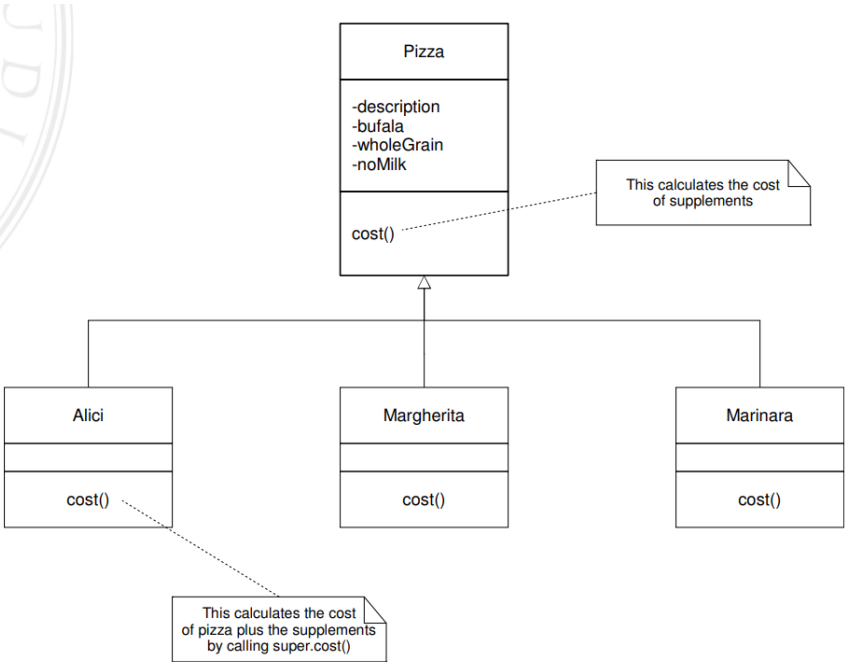


Attuando una semplificazione dell'immagine proposta, affinchè un concetto simile possa essere adoperato per contesti generali, si sottolinea la presenza di una *superclasse* e un totale di tre *sottoclassi*, in cui è applicato il principio del *poliformismo* in maniera tale che classi derivate possano adottare un'implementazione specifica dei *metodi ereditati*.
Tuttavia, tale meccanismo di modellazione aumenta notevolmente la complessità architetturale, a causa del numero crescente di funzionalità che possano essere richieste, dove, in un domain model del genere, ogni *classe figlia* richiede una propria specializzazione, formulando ricorsivamente condizioni di *ereditarietà* e *polimorfismo*.



Come già espresso in precedenza, ogni combinazione diviene una nuova entità da gestire, provocando un numero enorme di classi rappresentative all'interno del modello realizzato, in cui il metodo *cost()*, incapsulato all'interno di sottoclassi finali, richiamerà la funzione per ogni termine antecedente, elaborando una consistente condizione di selezione (*riferimento alla realizzazione di rami if-else annidati, contrari rispetto ad una software quality di spessore*).

Una risoluzione alternativa potrebbe consistere nell'inserimento delle *combinazioni* citate all'interno della *classe padre*, come attributi privati booleani, in cui il metodo *cost()* incapsulato mantiene la spesa per ogni supplemento aggiunto; mentre ognuna delle *classi derivate* ridefiniscono la funzione aggiungendo al prezzo il *costo base*.



Nonostante raffiguri un approccio accettabile per implementare un sistema software con queste caratteristiche, è affetto da due problematiche, quali:

- Le *classi derivate* durante il *lifecycle* della soluzione sono soggette a modifiche, a cui si aggiungono le variazioni che i *supplementi* possano subire; conseguentemente, ogni volta occorre correggere il metodo *cost()*, a seconda dell'aggiunta oppure dell'eliminazione di nuove *combinazioni*
- *Supplementi* sono modellati come un'unica entità, ma, come riportato nel *domain model* di riferimento, nulla vieta che siano richieste *combinazioni duplicate*, effettività che con attributi booleani non è permesso elaborare

Decorator (Structural Pattern)

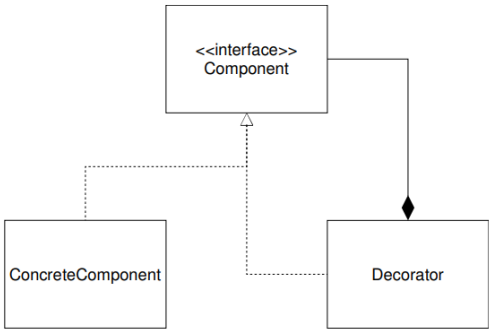
Problema
Come potrebbero essere aggiunti comportamenti oppure stati ad istanze di classi software

durante run-time?

L'intento consiste nell'aggiunta di *behavior* ad oggetti individuali durante l'esecuzione del processo esecutivo, in quanto *ineheritance* risulta essere *statica* ed attuata all'intera classe software di riferimento, per cui non è flessibile a variazioni ricorrenti.

Decorator tenta di rimediare alle mancanze precedenti, immettendo dinamicamente *responsabilità* agli oggetti; inoltre, illustra un'alternativa molto più flessibile rispetto alla gerarchia composita di *sottotipi*, la quale non si addice propriamente all'estensione di *funzionalità*, e nemmeno alla creazione di nuovi metodi.

Rispetto al nuovo termine introdotto, affinchè sia appreso appieno il meccanismo del pattern in questione, è bene considerare *Decorator* come se fosse un *contenitore*. La soluzione include l'*incapsulazione* dell'oggetto originario all'interno dell'entità *decoratrici*, poichè i termini contraddistinti espongono della stessa interfaccia affinchè possano modellare lo stesso *dominio*. Ai fini della descrizione attuata è proposta un'illustrazione grafica che possa chiarire quanto detto.



Si analizzano le componenti nella loro singolarità, in cui:

- *ConcreteComponent*, entità software pertinente all'istanza di oggetti che possano essere successivamente arricchiti
- *Decorator*, classe decoratrice la quale rappresenta un certo comportamento o stato che possa essere aggiunto ad oggetti della classe concreta
- *Component*, interfaccia da cui i due termini precedenti estendono funzionalità affini all'implementazione attuata

L'approccio prevede che i *clienti*, ossia *high level classes*, stabiliscano se vogliano aumentare la caratterizzazione comportamentale di oggetti relativi alla classe concreta, mediante l'utilizzo di decoratori; meccanismo reso possibile dall'interfaccia *Component*, in grado di soddisfare il *Dipendency Inversion Priciple*, ma soprattutto sviluppando i *comportamenti* affinchè siano legati alla stessa realtà modellativa, dato che ereditano attributi e metodi inerenti a *ConcreteComponent*.

Adapter (Structural Pattern)

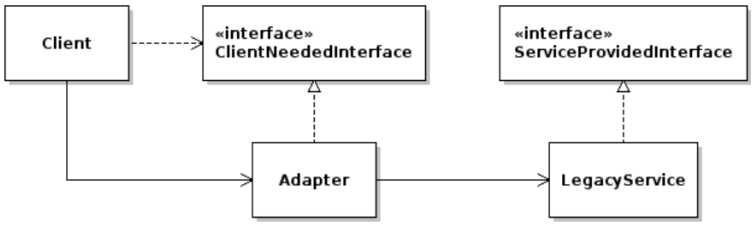
Problema
Come usufruire dei metodi di una classe nonostante non siano conformi all'aspettative del

Client?

L'intento è quello di convertire l'interfaccia di una classe in maniera tale che sia conforme rispetto alla richieste avanzate dal *Client*; per cui si tenta di racchiudere funzionalità di un'entità software esistente all'interno di un *layer astratto*, tipicamente una *interface*. A livello comportamentale è molto simile a pattern illustrati in sezioni precedenti, come *Facade*, in cui l'obiettivo sommatoriamente consiste nella contrapposizione di classi software tra *high level classes* e *low level classes*, affinchè operino come se fossero dei mediatori di *richieste* e *risposte*.

Per cui *adapter* rappresenta un *contenitore* che dispone funzionalità, metodi oppure attributi in modo affine alle richieste di classi *cliente*.

Caso di studio



Come da raffigurazione, la classe *Adapter* è sovrapposta tra l'interfaccia *ClientNeededInterface*, richiamata dall'utente finale, e *LegacyService*, la quale sviluppa logica algoritmica. Al suo interno l'adattatore contiene un'istanza di *LegacyService*, affinchè possa reindirizzare le funzionalità richieste, sia comportamentali che strutturali, all'interfaccia *ClientNeededInterface*, rispettando in questo modo anche il principio *dipendency inversion*.

Concludendo, in questo contesto il pattern realizza un'entità software con un background logico, per cui non è possibile implementare un termine di *facciata*, che provveda solamente ad estrapolare i dati necessari, per cui occorre concretamente sviluppare una nuova realtà che possa contenere metodi e attributi richiesti da *high level classes*.

Bridge (Structural Pattern)

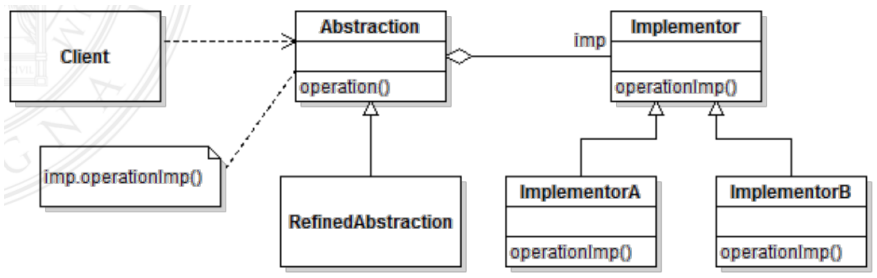
Problema
Come rompere la stretta gerarchia composita di astrazioni imposte dai Client?

Con il termine *tirannia* si definisce una certa tipologia di modellazione che pone *high level classes* ad un livello di maggiore importanza, poichè, in base alle richieste effettuate, *low level classes*, quindi inerenti alla logica algoritmica, si adeguano all'imposizioni implementative del *Client*, rischiando in questa maniera di non attuare un comportamento generale, rendendosi troppo specifiche.

Si ricorda che con il termine *Client* si intendono elementi del sistemi software legati al dominio, mentre le classi *concrete* raffigurano entità non inerenti alla realtà percepita, in cui si tende ad adottare un ambito implementativo generale, affinchè siano utilizzabili per

differenti scopi.

Soluzione
Disaccopiare le astrazioni dalle loro implementazioni in maniera tale che possano variare indipendentemente dalla richieste di high level classes.



In relazione allo schema riportato si adotta una descrizione per ogni singolo elemento, in cui:

- *Client*, high level class, entità legata al dominio
- *Abstraction*, definisce la gerarchia di astrazione modellata all'interno del sistema software, affinchè siano conseguiti i principi del modello *SOLID*
- *Implementor*, classe padre da cui dipendono gli *adattatori* comportamentali sottostanti, si ricalca l'obiettivo del pattern *Adapter*

Mediante una composizione architetturale simile è possibile isolare le due sfere, costituendo un *bridge* posto tra i due livelli di astrazione, che comportano la suddivisione tra la logica algoritmica ed entità di dominio. Si evince anche l'importanza degli *adattatori*, in cui rispetto al livello generale, permettono di racchiudere al loro interno le funzionalità richieste dal *Client*.

Riassumendo, se nella sezione precedente si tentava di adeguare *low level classes* al modello comportamentale oppure strutturale delle *high level classes*, in questo contesto si attua il meccanismo opposto, ossia *elementi del dominio* sono modificati in modo tale che siano conformi alle pretese della *logica algoritmica*.