Design Pattern 3

Introduzione

Obiettivi:

- Applicare propriamente i pattern del modello 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 dominio 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)

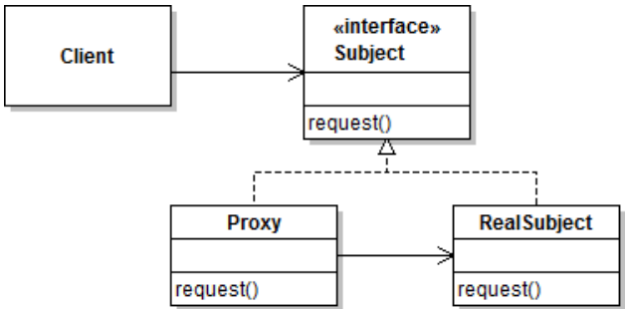
Problema
Come intercettare l'accesso di certi elementi del modello ad oggetti di ulteriori classi soft-

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 *Dipendency 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 software soggetta a controlli pur di verificare il corretto utilizzo di istanze affiliate, caratterizzate dalle da oggetti esterni all'elemento del dominio

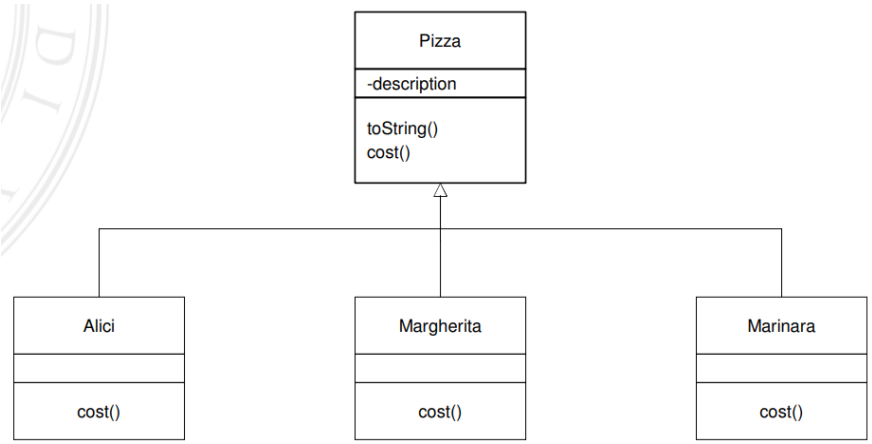
In questo contesto è la classe *Proxy* che promuove lo smistamento di richieste e risposte, affinchè non siano violati i principi *SOLID*, ovviando ad una scarsa qualità del codice, e attuando un meccanismo che possa soddisfare i requisiti imposti dalla classe cliente.

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.

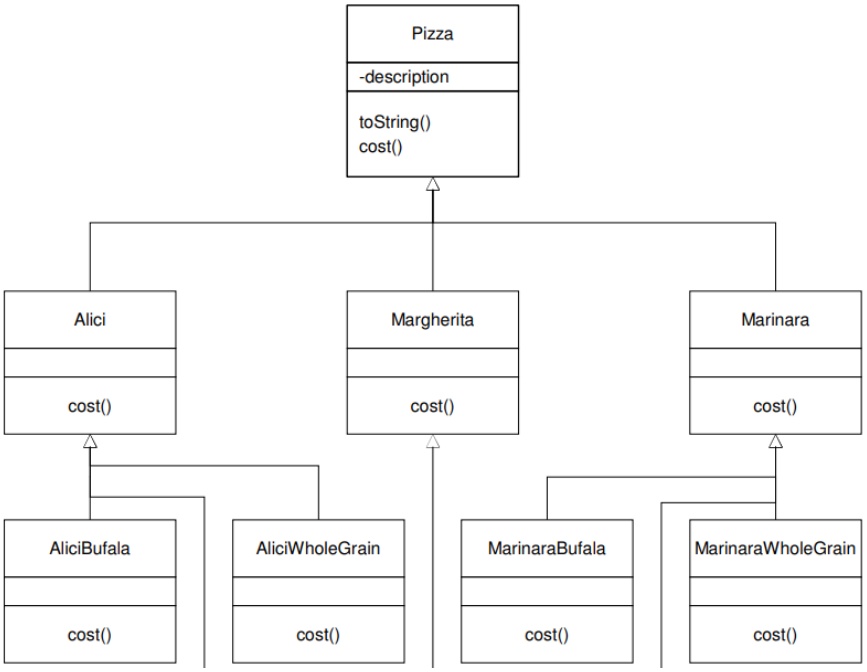
Intermezzo

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

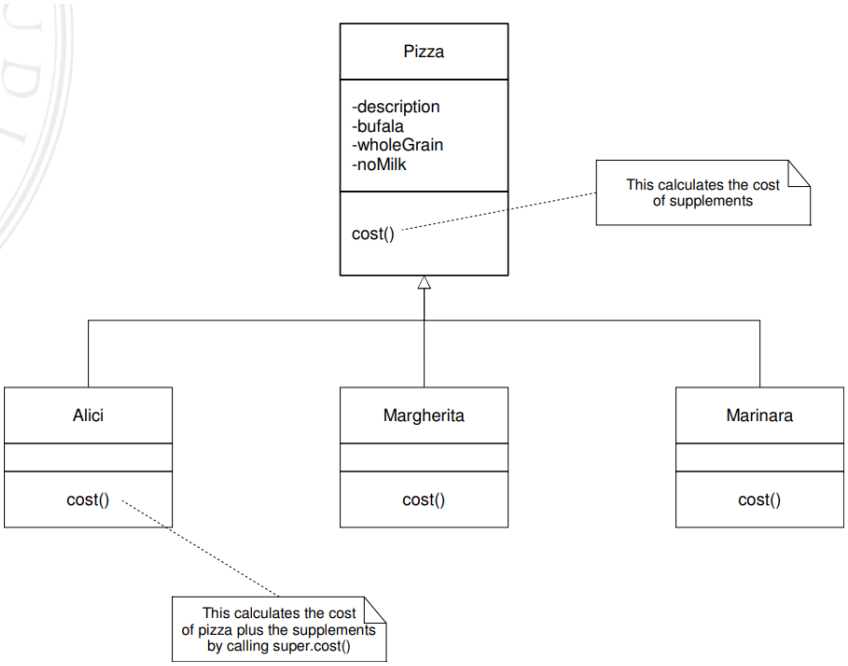


Attuando una semplificazione dell'immagine proposta, affinchè un concetto simile possa essere adoperato per contesti generali, si sottolinea la presenza di una *superclasse* e un totale di tre *sottoclassi*, in cui è applicato il principio del *poliformismo* in maniera tale che classi derivate possano 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:

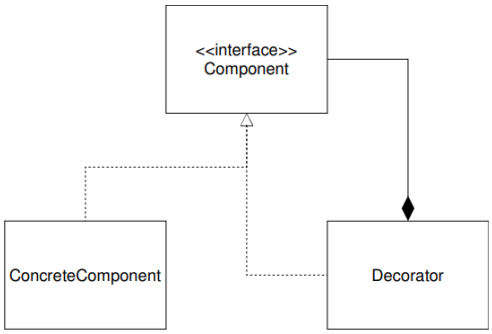
- Le *classi derivate* durante il *lifecycle* della soluzione sono soggette a modifiche, a cui si aggiungono le variazioni che i *supplementi* possano subire; conseguentemente, ogni volta occorre correggere il metodo *cost()*, a seconda dell'aggiunta oppure dell'eliminazione di nuove *combinazioni*
- *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)

Decorator tenta di rimediare alle mancanze precedenti, aggiungendo 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*. 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*, implementa processi comportamentali
- *Decorator*, contenitore
- *Component*, interfaccia comune alle due entità precedenti

...