Design pattern 2

Introduzione

Obiettivi:

- Applicare propriamente i pattern del modello GoF

Questa sezione rappresenta il naturale conseguimento del documento *Design Pattern 1*, in cui sono elencati ulteriori pattern *comportamentali*, *creazionali* e *strutturali* del *catologo GoF*.

State Pattern

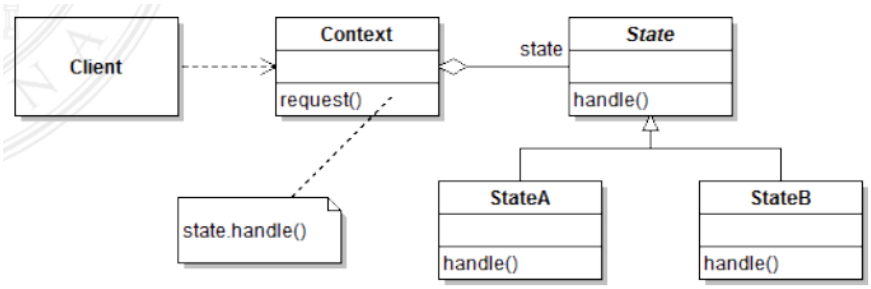
Problema
Come intercambiare il comportamento di un oggetto dipendente da uno stato tramite una soluzione di alta qualità?

Soluzione
Definire un meccanismo che permetta ad un oggetto di variare il proprio comportamento quando lo stato interno dell'oggetto varia.

Il pattern **State** è una soluzione applicata alla necessità di creare un insieme di comportamenti caratteristici strettamente correlati ad uno stato, che possa acquisire l'oggetto della classe in questione. In relazione all'adozione di un meccanismo di composizione simile, sono adottati sei passi principali:

- Creare un layer astratto, quale un'interfaccia, che definisca l'intero contesto a cui una classe possa richiedere di variare il proprio comportamento
- Creare una classe astratta da cui deriveranno le classi concrete, ossia che implementano i *behavior* specifici
- Rappresentare i differenti stati acquisibili dal sistema software modellato come sottoclassi derivanti dalla classe astratta principale
- Implementare il meccanismo comportamentale specifico per ogni singola sottoclasse derivata
- Mantenere un riferimento all'interno dell'interfaccia affinchè sia possibile variare il comportamento in base allo stato corrente dell'istanza
- Concludendo, per variare lo stato del sistema software, modificando il comportamento associato, semplicemente verrà cambiato il riferimento dello *stato corrente*

Caso di studio
Affinchè sia possibile comprendere quando e come applicare al meglio il pattern *State*, di seguito è proposto un esempio grafico che possa raffigurare al meglio quanto detto.



Come da raffigurazione si notano i passaggi elencati precedentemente, in cui la classe *Context* rappresenta il livello di astrazione necessario affinchè sia possibile intercambiare comportamenti di oggetti durante il run time.
Context a sua volta delega alla classe astratta *State* la responsabilità di implementare il comportamento specifico, in cui in questo caso avviene per uno dei due *behavior* esternalizzati, *StateA* e *StateB*; è bene sottolineare la sottile differenza che vige dal pattern *Strategy*, dove nel meccanismo di delega introdotto la scelta del *behavior* è imposta dall'utente finale, si ricorda il processo modellato, in cui è definita l'architettura generale dell'algoritmo affinchè siano classi derivate a specificarne l'implementazione di step specifici, mentre mediante *State* sarà l'interfaccia ad imporre quale metodo debba essere considerato affinchè sia garantito il cambiamento dello stato corrente, il quale gioca un ruolo fondamentale poichè permette intercambiabilità del processo comportamentale dell'istanza in questione.

Infine si considera la casistica in cui debbano essere introdotte nuove funzionalità. Dato che si considera lo stretto legame posto tra *comportamento-stato*, ogni volta che si debba implementare un nuovo *behavior* sarà creata una nuova classe derivata che specificherà la logica logaritmica del metodo, ovviando a *design smells*, come *needless repetition*, e violazioni dei principi *SOLID*.

Factory

Linguaggi legati al paradigma degli oggetti promuovono l'utilizzo di istanze di classi software, affinchè mediante funzionalità incapsulate al loro interno e una corretta manipolazione delle dipendenze, possano dare vita alla soluzione che soddisfi i requisiti funzionali degli utenti finali.

Tuttavia nella stesura del codice spesso non si pone abbastanza importanza sulla creazione di oggetti, ottenuti tramite la semantica *new (...)*, in cui modifiche all'interno del costruttore di classi concrete potrebbero provocare la rottura dell'operatività di *low level classes*. Una prima soluzione potrebbe consistere nel porre il corretto livello di astrazione tra elementi del modello legati alla *business logic* rispetto a classi adoperate da *UI*.

Nonostante la risoluzione, la quale potrebbe risultare conforme rispetto ai principi del modello *SOLID*, è affetta da un'errata gestione delle dipendenze ed espone *LLC* alla determinazione di classi concrete. Affinchè il tema trattato risulti comprensibile, si visualizza di seguito un caso di studio.

Caso di studio

Si pensi alla creazione di un sistema software che debba gestire le singole operazioni di un servizio *InBank*, in cui si vuole relegare una qualsiasi transazione di denaro ad una condizione di estrema persistenza. Per cui ponendo due attori A e B, i quali si scambiano una certa somma di denaro mediante un bonifico bancario, occorre un oggetto che gestisca la transazione di denaro e un'ulteriore istanza che provveda al controllo di persistenza.

FilePersistMgr fpm = new FilePersistMgr();

fpm.makePersistent(b);

Mediante l'ausilio di un file, avviene la scrittura della transazione eseguita dei due conti, dove, tralasciando l'erronea gestione di concorrenza, una soluzione simile non può essere considerata certamente corretta, dato che si prediligie un collegamento diretto tra *low level class*, colei che richiama l'istanza, e *high level class*, ossia la logica logaritmica che permette la scrittura del dato bonifico sul documento. Dato che una *dipendenza* è potenzialmente un percorso che possa diffondere una serie di cambiamenti, si tenta di predire il vincolo comportamentale minimizzando il possibile evento catastrofico che si potrebbe verificare all'interno della soluzione software.

Si passa quindi all'uso di layer astratti, come interfacce, capaci di porre una *schermatura* che consenta ad alcune funzionalità di riportare modifiche solamente ad una delle due parti in gioco, HLC oppure LLC, ovviando alla possibilità che si possa riverbare sull'intera struttura del sistema software adeguato.

PersistMgr fpm = new FilePersistMgr();

fpm.makePersistent(b);

In questa maniera è possibile porre l'interfaccia *PersistMgr* come l'artefice della creazione dell'istanza della classe concreta, nonostante questo non garantisca il livello di correttezza voluto.

Giunto a questo punto, interviene la *pure fabrication*, ossia l'idioma *Factory*, in cui il caso di studio diviene:

PersistMgr fpm = Factory.createPersistMgr();

fpm.makePersistent(b);

Per cui si tende a spostare le responsabilità in ulteriori classi, in cui sarà l'interfaccia *Factory*, in maniera totalmente autonoma, l'artefice della creazione dell'istanza della classe concreta, in cui elementi di basso livello saranno totalmente disinteressati rispetto al processo esecutivo che porti alla realizzazione dell'oggetto desiderato, ponendo quindi il corretto layer di astrazione.

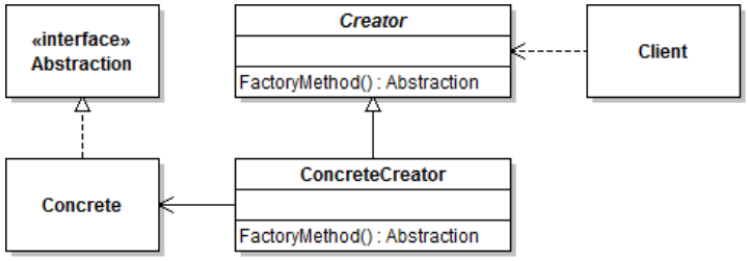
Factory Method Pattern

Problema
Come creare un'istanza di una classe senza che sia specificata l'esatta classe?

Soluzione
Definire un'interfaccia per la creazione di oggetti, la quale permetta alle singole sottoclassi

di decidere quale entità debbe essere stanziata.

Caso di studio
E' bene sottolineare che l'idioma e il pattern della GoF non rappresentino la stessa cosa. In quanto *Factory Method* non indica una classe esterna, come era stato stabilito nel caso di studio precedente, ma raffigura entità relegate a singoli oggetti oppure a certe funzionalità.



Nel contesto raffigurato è solamente il metodo *FactoryMethod()* che si occupa della creazione della classe concreta. Concettualmente è molto simile al pattern *Template Method*, in cui viene implementata la struttura portante dell'algoritmo, tralasciando a sottoclassi la resposanbilità di implementare certi passaggi come preferiscono. Concludendo, in questo modo il metodo illustrato è legato alla logica algoritmica ed è l'unico di carattere astratto, ovviando alla specifica esatta della classe che voglia essere stanziata.

Notification

Il problema relativo alla *notification* può essere espresso come un'avversità pragmatica e contraria alla realizzazione della soluzione. Applicando una semplificazione, spesso questa problematica, piuttosto ricorrente, ritrae componenti soggette a notifiche qualora vari lo stato di elementi correlati. Tuttavia l'intento promuove l'uso di strumenti affinchè l'oggetto notificante non sia dipendente da terze parti. **Chiedere se ci sta mettere la foto oppure esempio da fare generale ...**

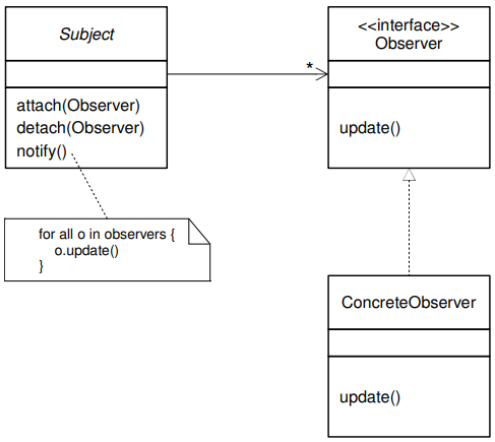
Riassumendo occorre realizzare una soluzione che mantenga in considerazione alcuni principi ritenuti fondamentali, come:

- *Protected Variations*, mettere in pratica la descrizione del principio affinchè cambiamenti conseguiti non varino funzionalità già esistenti
- *Interface Segregation*, definire un'interfaccia più ridotta possibile che possa contenere la dipendenza dell'oggetto notificante rispetto alla classe in ricezione
- *Dipendency Inversion*, dipendere da astrazioni, ossia applicare *abstract layer*, affinchè sia posta una schermatura tra *low level classes* e *high level classes*

L'insieme di queste tematiche è affrontato dal *Observe pattern*, il quale ammette:

Soluzione
Tramite il *pattern* avviene la costruzione di una classe astratta posta tra l'istanza notificante e l'oggetto in ascolto, affinchè tutte le dipendenze della prima citata possano essere automaticamente segnalate oppure aggiornate sul cambiamento di stato.

Caso di studio



Come da raffigurazione, la classe *Subject* rappresenta l'entità che deve comunicare alle proprie dipendenze il cambiamento del proprio stato, mentre *ConcreteObserver* indica la classe concreta in ascolto o in ricezione di modifiche, ossia il vincolo comportamentale prima definito. Ribadito più volte in altri contesti simili, l'uso di un'interfaccia astratta permette di interrompere lo stretto collegamento tra le classi in questione, rispettando soprattutto i tre principi prima elencati.

Piccola nota è relativa ai metodi implementati da *Subject*, in cui *attach()* è implementato per aggiungere un osservatore, *detach()* permette di eliminare una dipendenza dell'elenco di notifica infine *notify()* garantisce l'aggiornamento a tutti gli osservatori.