GRASP

Introduzione

Obiettivi:

- Definire che cos'è un pattern
- Comprendere quali siano i pattern che contraddistinguono GRASP

Il paragrafo esplicita quali siano i metodi affinchè i principi legati al paradigma orientato agli oggetti, inerenti alla qualità del software, possano essere concretamente attuati. Fino ad ora è stata adottata una definizione della semantica *object design*, in cui si affermano una serie di passaggi, i quali adoperati sequenzialmente danno vita al risultato finale in grado di rispondere alla richiesta dei requisiti funzionali.

Definizione informale
Per ottenere il processo esecutivo sperato occorre innanzitutto identificare i *requisiti funzionali*, artefici della creazione del *domain model* di riferimento, successivamente dovranno essere aggiunti i metodi che caratterizzino le differenti classi della soluzione, e infine definire come le stesse *istanze* debbano comunicare per soddisfare la richiesta.

Tuttavia, questa breve descrizione non è di gran auspicio, dato che in qualsiasi step che la contraddistingue potrebbero sollervarsi problematiche e *design smells*, peggiorando la qualità del software.

Per cui necessita uno strumento in grado di illustrare singole direttive capaci di approcciare un metodo empirico ad una progettazione legata agli oggetti, eliminando ogni grado di incertezza. Ciò è possibile attraverso l'implementazione dei *pattern* del modello *GRASP*.

Responsability

UML definisce una **responsabilità** come un *contratto oppure obbligo di un classificatore*; per cui le *responsabilità* sono relazionate ai vincoli comportamentali di un'istanza. Tipicamente sono suddivise in due macro-aree, le quali si contraddistinguono in:

- *Doing*, responsabilità simili di un oggetto includono certi atteggiamenti come, operare per stessi termini, ossia creare un'istanza o processare un calcolo internamente, inizializzare un'azione per ulteriori oggetti oppure controllare e gestire attività di altre istanze
- *Knowing*, responsabilità simili di un oggetto illustrano una serie di comportamenti, quali, conoscere i dati incapsulati al suo interno, apprendere a quali oggetti sia relazionato oppure comprendere quali elementi possa derivare per proprie azioni

Per cui in breve le due tipologie possono essere riassunte come segue; *doing responsabilities* permettono la realizzazione di computazioni algoritmiche, mentre *knowing responsabilities* prevedono di apprendere le informazioni interne ad ogni istanza di riferimento.

Una responsabilità non può essere paragonata ad un metodo, ma un metodo implementa responsabilità. Quest'ultimo inciso indica una caratterizzazione sottile, la quale può essere

considerata mediante un *activities diagram*. Si prenda come esempio il diagramma posto all'interno del capitolo *design goal*; l'attore pur di apprendere il totale della spesa effettuata, demanda la richiesta alla classe *saleManager*, il quale a sua volta, pur di ottenere l'informazione, reclama il valore alla classe *sale*, contenente l'insieme di attributi e funzioni in grado di rispondere alla domanda.

Nonostante rappresenti una raffigurazione di estrema semplicità, si nota l'affermazione precedente, ossia i metodi implementano responsabilità, delegando la richiesta ad un elenco di istanze, fino a quando non si riscontri l'oggetto capace di soddisfare la domanda.

Un approccio dedicato all'assegnamento di responsabilità è denominato *RDD*, ossia *responsability driven design*. Quindi è un metodo per progettare sistemi software basato sulle responsabilità, in cui esse sono assegnate alle *classi software* durante l'elaborazione. ...

GRASP

GRASP è l'acronimo di *General Responsability Assignment Software Pattern*, ossia descrive i principi fondamentali della progettazione ad oggetti e permette l'assegnazione di responsabilità, detti *patterns*. Spesso è utilizzato per implementare gli aspetti salienti di *RDD* ma garantendo una solida costruzione sui principi cardine. In ingegneria del software il paradigma introdotto è considerato come l'artefice del conseguimento in sistemi software di spessore, poichè permette di analizzare e comprendere gli elementi essenziali nella progettazione ad oggetti, applicando un approccio metodico, razionale e improntato alla massima espressività.

Prima di illustrare i differenti apici su cui è posto, è bene definire l'etimologia dell'espressione *pattern*.

Definizione informale
Un **pattern** è una soluzione generale ad un problema ricorrente.

Come da considerazione descritta dall'architetto Alexander, un qualsiasi *pattern* illustrato considera un problema che occorre frequentemente all'interno di un ambiente, capace di associarne una soluzione, la quale può essere utilizzata con lo stesso metodo per cui sia stata già usata per risolvere correntemente la problematica ripetitiva. Riassumendo quanto detto, tramite il termine pattern, un problema ricorrente di medesima natura può essere risolto dalla stessa soluzione precedente.

Favorendo un'ulteriore chiave di lettura dal teorico Larman, un *pattern*, per maggiore semplicità, rappresenta una descrizione nominativa di un problema e della sua soluzione correlata, indicando quando e come applicarla in nuovi contesti.

Introdotto il concetto su cui fonda l'intero contesto è possibile definire ora quali siano i pattern che contraddistinguono il modello *GRASP*.

Patterns

Di seguito sono riportati i *patterns* principali che contraddistinguono il modello *GRASP*, in cui è attuata una suddivisione tra la definizione del problema e della soluzione correlata.

Information expert

Problema
Come assegnare propriamente responsabilità alle istanze delle classi software.

UML potrebbe definire un numero sempre più elevato di responsabilità tra le classi, e lo stesso processo esecutivo potrebbe richiedere una quantità crescente di vincoli comportamentali. Buona pratica consiste nell'assegnare le *responsabilities* tra *software classes* durante la modellazione dell'interazioni tra oggetti; se eseguito correttamente il sistema tenderà ad essere più semplice da comprendere, mantenendo l'opportunità della qualità interna *reusability*.

Soluzione
La soluzione consiste nella sistematizzazione della classe software che acquisca la responsabilità. L'espressione riportata indica che l'assegnamento di responsability dovrebbe accadere nei confronti di classi già esistenti, scegliendo tra quelle che comprendano il numero maggiori di informazioni necessarie.

Creator

Problema
Come scegliere chi dovrebbe essere il responsabile della creazione di nuove istanze di classi software.

La creazione di oggetti è una delle pratiche più comuni di un linguaggio legato al padarigma degli oggetti. Conseguentemente, sarebbe utile individuare un principio in grado di stabilire una responsabilità affine all'istanze di oggetti. Una correlazione corretta della responsability potrebbe alludere ad una maggiore chiarezza sintattica e alla massimizzazione dell'*incapsulazione* e della qualità *reusability*.

Soluzione
Adoperando una semplificazione semantica, l'assegnazione della responsabilità alla classe B, autrice della creazione di un'istanza della classe A, è corretta se rispettata almeno una delle condizioni seguenti:

- B aggrega, ossia contiene o compone, oggetti di A
- B contiene oggetti di A
- B utilizza fortemente oggetti di A
- B è in possesso di tutti i dati necessari per poter richiamare la funzione della classe software A, banalmente ha disposizione tutti gli attributi sufficienti per il metodo di destinazione

Nel caso più classi dovessero rispettare condizioni elencate, si tende a premiare effettività

che comprendano le prime due caratteristiche illustrate.

Controller

Problema
Come scegliere quale oggetto debba ricevere e coordinare un'operazione del sistema software.

Differenti sono i *patterns* valutativi in cui possono essere presi in considerazione candidati piuttosto simili fra loro, in relazione alle classi poste all'interno del domain model. Il termine *controller* è adottato proprio per esprimere una soluzione al problema; *classi di controllo* potrebbero divenire uno strato posto tra classi di alto livello, quindi affini alla logica algoritmica, e classi di basso livello, per cui in relazione ad una raffigurazione dell'entità usufruibili dall'utente finale. L'utilizzo di uno scudo sovrapposto permette di isolare elementi dinamici, proprio come *user interface*, le quali non devono mai interagire con classi prettamente legate allo sviluppo di metodi basici (*si ricordi l'activity SaleManager all'interno del capitolo design goal, posto in mezzo alle attività UI e sale*).

Concludendo, un controller non rappresenta un'interfaccia finale con cui utenti possano interagire, ma definisce l'architettura comportamentale del sistema software dinnanzi alla ricezione e gestione di eventi.

Soluzione
Come già accenato dall'introduzione del problema, porre responsabilità relative alla gestione di eventi del sistema dovrebbe avvenire nei confronti di interfacce create appositamente per la manipolazione di trasferimenti di dati e richiesta di informazioni. L'assegnazione richiede che siano rispettate almeno una delle seguenti condizioni, quali:

- Una classe simile rappresenta il cosiddetto *root object*, ossia l'elemento che il software in questione provvede ad interpellare con maggiore frequenza, spesso illustrato come il sottosistema che contiene tutte le *variazioni comportamentali di facciata*, ossia la totalità di funzioni che una *GUI* implementata possa richiedere alle *high level classes*, mediante un *controller*
- La classe è denominata tramite il nominativo di riferimento al *caso d'uso* modellato, si ricorda che da un unico *use case* è possibile modellare un *sequence diagram* molto più articolato, in cui è anteposta la nomenclatura *Handler*, *Controller* oppure *Session*. Per cui, attuando una semplificazione, qualora adottato *UseCaseNameController* esso sarà il punto cardine su cui tutti gli eventi del caso d'uso faranno riferimento

Low Coupling

Problema
Come stabilire dinamiche che scoraggino *elevata dipendenza* e *impatto modellativo*, affinchè si possa massimizzare il *riutilizzo*.

Coupling è una misura di come un elemento sia fortemente connesso o meno rispetto a possibili corrisposti. Per elemento si intende un insieme di caratterizzazioni, come classi, sistemi oppure sottosistemi. Per cui un elemento con un *low coupling*, ossia un basso layer di accoppiamento, non è dipendente da un numero crescente di entità, mentre in maniera opposta un elemento è detto *strong coupling* qualora sia artefice di *contesti dipendenti*. Si nota

nuovamente la centralità delle *dipendenze* e delle problematiche derivanti da esse, totalmente contrarie rispetto all'obiettivo del pattern illustrato, tra cui:

- Cambiamenti suscitano modifiche locali delle classi, situazione contraria rispetto all'*OO*, ossia *Open Closed Principle*
- Maggiore complessità nell'intento di isolare elementi software
- Maggiore complessità relativa al *riutilizzo* dato che l'implementazione delle classi ha portato ad un numero elevato di dipendenze, producendo un effetto a cascata

Soluzione
In pratrica, il *livello di accoppiamento* non può essere considerato come un'entità a sè stante da ulteriori principi, tra cui *Information Expert* oppure *High Coesion*. Spesso nelle proposte valutative è considerato come un fattore attuabile solamente con la sequenzialità implementativa della progettazione. **Mettere esempio del libro Register, Payment ...**

High Coesion

Problema
Come rendere elementi comprensivi e maneggevoli, i quali possano essere anche un termine di supporto per il pattern valutativo *Low Coupling*.

Il problema potrebbe essere riassunto semplicemente come mantenere una complessità modellativa costante e manipolabile. Similmente al pattern *Low Coupling*, è introdotta una misura dedicata a tale tipologia di risoluzione, la *coesione*. La *functional cohesion* è la capacità adottata per distinguere il grado di correlazione di una *responsabilità* rispetto ad un'ipotetica associazione ad elementi del dominio, come classi o sistemi. Classi con un livello elevato di responsabilità, non indicano una quantità operativa consistente, ma definisce un elevato layer di coesione. Per cui l'assegnazione di responsabilità non va di pari passo rispetto all'operatività inclusa all'interno dell'elemento del dominio, anzi classi con basso livello di coesione illustrano situazioni sormontate dalla quantità di funzionalità da eseguire. I difetti relativi a quest ultimo passaggio sono descritti come segue:

- Difficile comprensione e manutenzione
- Il principio *reusable* non è attuabile
- Piuttosto sensibili a modifiche, piccole variazioni potrebbero provare enormi cambiamenti

Soluzione
Equivalente a *Low Coupling*, il pattern *High Coesion* non può essere attuato senza considerare l'intero contesto delle responsabilità già adottate e dei principi di riferimento. Occorre quindi applicare una certa granularità all'interno del *domain model*, stanziando classi che abbiano una coerenza a livello comportamentale, ossia siano riportati metodi appartenenti ad uno stesso insieme modellativo. **Mettere esempio ?**

Indirection

Problema

Come assegnare responsabilità affinchè sia possibile ovviare ad un accoppiamento tra due o più elementi, e come attuare il processo opposto al coupling per supportare un riutilizzo potenzialmente maggiore.

In questo contesto gioca un ruolo fondamentale l'aspetto della *pura invenzione*, o meglio definito **pure fabrication**. Non tratta la creazione di elementi senza un rigore modellativo e logico, ma attua un iter di step affinchè sia necessaria una figura che possa sia elaborare funzioni di *coupling* che di *de-coupling*. Dopo aver analizzato tutte le classi presenti a cui applicare tale responsabilità, se non individuate caratteristiche che possano mantenere un livello elevato di coesione e nessuna di esse definisca un *Information Expert*, è necessario introdurre entità, mediatori che possano allineare e disallineare dipendenze tra classi.

Soluzione
Le responsabilità dovrebbero essere assegnate ad un *intermediario*, ossia un mediatore tra differenti componenti del sistema software che non siano direttamente dipendenti.

Protected Variations

Problema
Come progettare elementi affinchè le loro instabilità non siano d'impatto su altri componenti della modellazione.

Il pattern *Protected Variations* è essenzialmente al pari del principio *OCP*, *Open Closed Principle*, il quale afferma che una classe dovrebbe essere aperta a nuove funzionalità ma non al cambiamento di metodi già implementati. Per cui rimanda all'uso di astrazioni modellative qualora vi sia la netta necessità, come le interfacce, affinchè cambiamenti legati a *low level classes* non riguardino metodi legati alla logica algoritmica. A favore di quanto detto, si adopera in questa tematica la *legge di Demetra*; essa formula il pretesto secondo cui una classe dovrebbe interagire, mediante metodi, chiamate alle funzioni interessate oppure alla creazione di istanze, solamente con entità di cui abbia lo stretto collegamento. Semplicimente, vanificando il contesto teorico, la classe dovrebbe invocare procedure di cui abbia tutto il necessario, evitando di gestire dipendenze superflue.

Soluzione
La soluzione prevede l'identificazione precedente di elementi soggetti ad instabilità, modellando interfacce che circondino l'insieme di entità dannose, dove un profondo cambiamento non intralci ulteriori oggetti. Concludendo, una piccola nota aggiuntiva, consiste che il termine *interfaccia* sia adoperato nel suo più ampio senso etimologico, principalmente a livello di astrazione piuttosto che implementativo, come accadrebbe per linguaggi orientati al paradigma degli oggetti, stabilendo una struttura *flessibile* improntata ad adeguarsi a possibili modifiche.