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
Pur di 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 impirico 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, iniziazzare 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 realizzione 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.

Definizione 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'istenze 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.

Definizione Creator

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*).

Soluzione
...