09 - GRASP

General Responsibility Assignment Software Patterns

Il decidere quali metodi appartengono a chi e come devono interagire gli oggetti ha delle conseguenze, e pertanto queste scelte devono essere fatte con attenzione.

> **Principi e pattern**
>
> La padronanza dell'OOD coinvolge un insieme ampio di principi flessibili (e pattern), con molti gradi di libertà.

La **progettazione guidata dalle responsabilità** (**RDD**) è un modo comune di pensare alla progettazione di oggetti software in termini di responsabilità, ruoli e collaborazioni.

Gli oggetti software sono considerati dotati di responsabilità, un'astrazione di ciò che un oggetto o componente software fa o rappresenta. In UML, la responsabilità è un contratto o un obbligo di un classificatore.

Le responsabilità sono fondamentalmente di due tipi:

- **Di fare**: Fare qualcosa da sé (creare un oggetto, calcolare), chiedere ad altri oggetti di agire, controllare e coordinare attività di altri oggetti.
- **Di conoscere**: Conoscere i propri dati privati incapsulati, conoscere oggetti correlati, conoscere cose che può derivare o calcolare.

Le responsabilità sono assegnate alle classi di oggetti durante la progettazione.
La traduzione in classi e metodi è influenzata dalla granularità delle responsabilità.
Le responsabilità sono implementate da oggetti e metodi che agiscono da soli o collaborano.

> **RDD è una metafora generale:**
>
> si pensi agli oggetti software come persone con responsabilità che collaborano. Questo porta a considerare un progetto OO come una comunità di oggetti che collaborano con responsabilità.

Passi della RDD (iterativi):

1. Identificare le responsabilità, una alla volta.
2. Chiedersi a quale oggetto software assegnare questa responsabilità.

3. Chiedersi come l'oggetto scelto soddisfa la responsabilità (da solo o collaborando, il che può identificare nuove responsabilità).
   Questo processo si basa su criteri appropriati per l'assegnazione di responsabilità, come i pattern GRASP.

> **Pattern GRASP**
>
> Sono un aiuto per apprendere la struttura e dare un nome ai principi.
> Sono uno strumento per acquisire padronanza delle basi dell'OOD e comprendere l'assegnazione di responsabilità nella progettazione a oggetti.

Le decisioni sull'assegnazione delle responsabilità possono essere prese durante la codifica o la modellazione.
Disegnare i diagrammi di interazione diventa l'occasione per considerare tali responsabilità (realizzate come metodi).

> **Pattern**
>
> È una descrizione, con nome, di un problema di progettazione ricorrente e di una sua soluzione ben provata e che può essere applicata a nuovi contesti.
> Molti pattern guidano l'assegnazione di responsabilità agli oggetti.

La nozione di pattern ebbe origine con i *pattern architettonici* di *Christopher Alexander*.
I pattern per il software furono sviluppati da *Ken Beck*.
Nel 1994, il libro "*Design Pattern*" di Gamma, Helm, Johnson e Vlissides (la "Gang of Four", GoF) *descrisse 23 pattern per la programmazione OO*. Questi sono diventati noti come design pattern **GoF** ("Gang of Four"), che sono più "schemi di progettazione avanzata" che "principi" (come GRASP).

> **Riepilogando:**
>
> - La RDD è una metafora per la progettazione degli oggetti: una comunità di oggetti con responsabilità che collaborano.
> - I pattern danno un nome e spiegano le idee della progettazione OO:
>   - GRASP per i pattern di base dell'assegnazione di responsabilità,
>   - GoF per idee di progettazione più avanzate.
> - I pattern possono essere applicati sia durante la modellazione che durante la codifica.
> - UML è per la modellazione visuale per la progettazione OO, dove possono essere applicati sia i pattern GRASP che quelli GoF.

Low Representational Gap (LRG)

Nella fase di progettazione, vale sempre il principio **LRG**, o **salto rappresentazionale basso**, tra il modo in cui si pensa al dominio e una corrispondenza diretta con gli oggetti software.
Bisogna sempre guardare il modello di dominio per trarre ispirazione.

Obiettivi Generali dei Pattern GRASP

Un sistema software ben progettato è facile da comprendere, da mantenere e da estendere. Inoltre, le scelte fatte consentono buone opportunità di riusare i suoi componenti in applicazioni future.

Comprensione, manutenzione, estensione e riuso sono qualità fondamentali in uno sviluppo iterativo, dove il software è continuamente modificato.

Progettazione Modulare

- *Comprensibilità*
- *modificabilità*
- *basso impatto nei cambiamenti*
- *flessibilità*
- *riuso*
- *semplicità*
  Il software deve essere decomposto in elementi software (moduli) coesi e debolmente accoppiati.

In GRASP, i principi della progettazione modulare sono rappresentati dai pattern **High Cohesion** e **Low Coupling**. Questi due pattern sono fondamentali, ma difficili da applicare direttamente perché sono pattern valutativi.

I Nove Pattern GRASP

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection

- Protected Variations

Analizziamo i cinque più importanti.

Creator

Pattern Creator

Nome: Creator (Creatore)

Problema: Chi crea un oggetto A? Ovvero, *chi deve essere responsabile della creazione di una nuova istanza di una classe?*

Soluzione: Assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (più sono vere meglio è):

- B "contiene" o aggrega con una composizione oggetti di tipo A
- B registra A[2]
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione (pertanto B è un Expert rispetto alla creazione di A)

Osservazioni:

Cercare il creatore che abbia realmente bisogno di essere collegato all'oggetto creato (low coupling).
Promuove basso accoppiamento, minori dipendenze di manutenzione e maggiori opportunità di riuso.

Information Expert

Pattern Expert

Nome: Information Expert (Esperto delle Informazioni)

Problema: Qual è un principio di base, generale, per l'assegnazione di responsabilità agli oggetti?

Soluzione: Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla, all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità

Osservazioni:

Analizzare il Modello di Dominio (o il Modello di Progetto per primo se questo è gia sufficientemente sviluppato) per cercare la classe degli oggetti che possiede le informazioni necessarie.

Low Coupling (Accoppiamento Basso)

Pattern Low Coupling

Nome: Low Coupling (Accoppiamento Basso)

Problema: Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?

Soluzione: Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative.

Accoppiamento (coupling)

È la misura di quanto fortemente un elemento è connesso ad altri, di quanto ha coscienza di altri elementi e dipende da altri elementi.

⚠ Problemi dell'alto accoppiamento

- I cambiamenti nelle classi correlate, da cui queste classi dipendono, obbligano cambiamenti locali anche in queste classi.
- Queste classi sono più difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono.
- Sono più difficili da riusare, perchè il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono.

Low Coupling è un principio di valutazione da utilizzare in parallelo ad altri pattern.

Le forme più comuni di accoppiamento da un tipo X a un tipo Y comprendono:

- X ha un attributo di tipo Y o referenzia un'istanza di tipo Y (o una collezione di oggetti Y).
- X richiama operazioni o servizi di Y.
- X crea un oggetto di tipo Y.
- il tipo X ha un metodo che contiene un elemento (parametro, variabile locale, tipo di ritorno) di tipo Y o che referenzia un'istanza di tipo Y.
- X è una sottoclasse, diretta o indiretta, di Y.
- Y è un'interfaccia, e X la implementa.

Le classi generiche e altamente riutilizzabili devono avere un accoppiamento particolarmente basso. Un moderato grado di accoppiamento è normale e necessario.

Una sottoclasse è fortemente accoppiata alla sua superclasse; si consideri attentamente ogni decisione di estendere una superclasse, poichè è una forma di accoppiamento forte. Porzioni di codice duplicato sono fortemente accoppiate tra di loro

✓ **Vantaggi:**

- Una classe o componente con un accoppiamento basso non è influenzata dai cambiamenti nelle altre classi e componenti
- È semplice da capire separatamente dalle altre classi e componenti.
- È conveniente da riusare

High Cohesion

Pattern High Cohesion

Nome:	High Cohesion (Coesione Alta)
Problema:	Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?
Soluzione:	Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative.

🖉 **Coesione (funzionale)**

È la misura di quanto fortemente siano correlate e concentrate le responsabilità di un elemento.

- un elemento con responsabilità altamente correlate che non esegue una quantità di lavoro eccessiva ha una coesione alta.

⚠ **Problemi della coesione bassa**

- Classi difficili da comprendere
- Classi difficili da mantenere
- Classi difficili da riusare
- Classi delicate, continuamente soggette a cambiamenti
  Spesso risulta che queste classi hanno assunto responsabilità che avrebbero dovuto essere delegate ad altri oggetti.

È un principio di valutazione che sceglie tra diverse alternative.

Possiamo definire più tipi di coesione:

- **Coesione di dati**: una classe implementa un tipo di dati (*molto buona*)
- **Coesione funzionale**: gli elementi di una classe svolgono una singola funzione (*buona* o *molto buona*)

- **Coesione temporale**: gli elementi sono raggruppati perchè usati circa nello stesso tempo (es. controller, *a volte buona, a volte meno*)
- **Coesione per pura coincidenza**: bro non ne parliamo, very bad

> ✓ **Vantaggi:**
>
> - maggiore chiarezza e facilità di comprensione del progetto
> - spesso sostiene Low Coupling
> - manutenzione e miglioramenti semplificati
> - maggiore riuso di funzionalità a grana fine e altamente correlate, poichè una classe se coesa può essere usata per uno scopo molto specifico

Controller

Pattern Controller

Nome: Controller (Controllore)

Problema: Qual è il primo oggetto oltre lo strato UI che riceve e coordina ("controlla") un'operazione di sistema?

Soluzione: Assegna la responsabilità a un oggetto che rappresenta una delle seguenti scelte:

1. rappresenta il "*sistema*" complessivo, un "*oggetto radice*", un dispositivo all'interno del quale viene eseguito il software, un punto di accesso al software o un sottosistema principale (variante del *facade controller*)
2. rappresenta uno scenario di un *caso d'uso* all'interno del quale si verifica l'operazione di sistema (un *controller di caso d'uso* o *controller di sessione*)

Si utilizzi la stessa classe controller per tutti gli eventi di sistema nello stesso scenario di caso d'uso.

> 🖉 **Note:**
>
> - Le classi dello strato UI non sono Controller;
> - I Controller coordinano i messaggi legati alle operazioni di sistema, non sono classi di dominio;
> - Posso controllare che gli eventi avvengano in un ordine prestabilito.

Controller è semplicemente un pattern di **delega**.

> ⚠ **Un problema comune:**

Un controller soffre di una coesione bassa, a causa di una eccessiva assegnazione di responsabilità.

Il pattern Controller suggerisce due categorie di controller:

- **facade controller**
- **controller di caso d'uso**

Facade controller

Rappresenta il sistema complessivo, una facciata sopra agli altri strati dell'applicazione. Fornisce un punto di accesso principale per le chiamate dei servizi dello strato UI agli altri strati sottostanti.
Sono adatti quando non ci sono troppi eventi di sistema.

Controller di caso d'uso

Un controller diverso per ogni caso d'uso.
Non è un oggetto di dominio, ma un costrutto artificiale per supportare il sistema.

✓ Vantaggi:

maggiore potenziale di riuso e interfacce inseribili;
opportunità di ragionare sullo stato del caso d'uso.