GRASP Patterns

Assegnare responsabilità

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Responsibility Driven Development (RDD)

Pensare alla progettazione in termini di responsabilità, ruoli, collaborazioni.

Le responsabilità per un oggetto sono tipicamente di due tipi:

- Fare
  - fare qualcosa esso stesso
  - chiedere ad altri oggetti di eseguire azioni
  - controllare e coordinare attività di altri oggetti
- Conoscere
  - conoscere i propri dati incapsulati
  - conoscere gli oggetti correlati
  - conoscere cose che può derivare o calcolare

Responsibility Driven Development (RDD)

Pensare alla progettazione in termini di responsabilità, ruoli, collaborazioni.
Le responsabilità per un oggetto sono tipicamente di due tipi:

- Fare
  - fare qualcosa esso stesso
  - chiedere ad altri oggetti di eseguire azioni
  - controllare e coordinare attività di altri oggetti
- Conoscere
  - conoscere i propri dati incapsulati
  - conoscere gli oggetti correlati
  - conoscere cose che può derivare o calcolare

Collaborazione

Le responsabilità possone essere implementati dal singolo oggetto o attraverso la collaborazione con altri oggetti.

Pizzeria 4.0

Come posso calcolare il totale da pagare? A chi assegnereste questa responsabilità?

RDD - How to

RDD prevede di applicare iterativamente i seguenti passi:

- Si identifichino le responsabilità e le si considerino una alla volta,
- Per ogni responsabilità ci si chieda quale classe dovrebbe farsene carico. È possibile che sia necessario aggiungere una classe,
- Ci si chieda come la classe possa soddisfare la responsabilità e se è ncessario mettere in atto collaborazionie e dunque possibili altre responsabilità.

Pattern GRASP

I pattern GRASP (General Responsibility Assignment Software Patterns) suggeriscono l'assegnamento delle responsabilità in accordo a specifici schemi:

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Pure Fabrication
- Indirection
- Polymorfism
- Protected Variations

Creator

- **Problema**: chi deve essere responsabile della creazione di una nuova istanza di una classe?
- **Soluzione**: assegna a *B* la responsabilità di creare istanza della classe *A* se:
  - *B* "contiene o aggrega con una composizione oggetti di tipo *A*
  - *B* registra oggetti di tipo *A*
  - *B* utilizza strettamente oggetti di tipo *A*
  - *B* possiede i dati per inizializzazione di oggetti di tipo *A*, che saranno passati al momento della creazione
- **Discussione**: attività molto comune, si tratta solo di una linea guida che comunque può prevedere eccezioni
- **Controindicazioni**: in alcuni casi creazione può essere molto complessa e può convenire delegare a classi di supporto
- **Vantaggi**: viene favorito accoppiamento basso. La classe *B* è già in qualche relazione con *A*

Information Expert

- **Problema**: esiste un principio generale per assegnazione di responsabilità agli oggetti?
- **Soluzione**: si assegni la responsabilità alla classe che è “esperta” in relazione alle informazione necessarie per farsi carico della responsabilità
- **Discussione**: LRG (Low Representational Gap) è in qualche modo alla base del principio. In alcuni casi informazioni risultano distribuite su più classi e dunque è necessario organizzare la collaborazione (e.g. totale conto)
- **Controindicazioni**: attenzione a non applicare il pattern per attività trasversali che potrebbero complicare la classe assegnandole troppe responsabilità
  - e.g. persistenza
- **Vantaggi**: favorisce incapsulamento e località delle informazioni. Generalmente risulta in coesione alta

Low Coupling

Accoppiamento

misura di quanto fortemente un elemento è connesso e dipendente da altri elementi. Attenzione non tutte le dipendenze sono uguali

- **Problema**: come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?
- **Soluzione**: assegna le responsabilità in modo che l'accoppiamento rimanga basso. Si usi il principio per valutare le alternative.
- **Controindicazioni**: accoppiamento alto con elementi stabili non è generalmente un problema
- **Vantaggi**: maggiore facilità di comprensione, più facile riuso, minore impatto dei cambiamenti

Low Coupling - discussione

Da considerarsi un principio di valutazione per comparare più possibilità progettuali

Forme comuni di accoppiamento

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

High Coesion

- **Problema**: come mantenere gli oggetti focalizzati, comprensibili, gestibili, sostenendo Low Coupling?
- **Soluzione**: assegna responsabilità in modo che la coesione rimanga alta
- **Discussione**: si tratta anche in questo caso di un principio per valutare più alternative. Ci si riferisce in questo contesto a coesione funzionale (alter tipologie – dati, temporale, casuale). In generale problemi di coesione e accoppiamento si mostrano allo stesso tempo
- **Controindicazioni**: può essere accettata una violazione per classi in relazioni a determinate specializzazioni. Altra possibilità può essere favorire efficienza.
- **Vantaggi**: migliore manutenzione, maggiore comprensione, maggiore riuso

la modularità è la proprietà di un sistema che è stato decomposto in un insieme di moduli coesi e debolmente accoppiati *G. Booch*

Controller

- **Problema**: qua le è il primo oggetto oltre lo strato UI che riceve e coordina un'operazione di sistema?
- **Soluzione**: assegna la responsabilità a una classe che:
  - rappresenta il "sistema" complessivo, un "oggetto radice" - varianti di un "façade controller"
  - rappresenta uno scenario di un caso d'uso, spesso chiamato "<UseCaseName>Handler". Questo caso permette naturalmente la gestione della "sessione"
- **Discussione**: si tratta di un pattern di delega. L'oggetto si occupa di controllare e coordinare le attività. Attenzione da non confondersi con controller di framework MVC. Il C dell'approccio BCE all'identificazione delle classi.
- **Controindicazioni**: controller gonfi
- **Vantaggi**: maggiore riuso e favorisce disaccoppiamento di UI. Permette naturalmente di estendere la discussione sui casi d'uso.

Pure Fabrication

- **Problema**: non volendo violare High Cohesion e Low Coupling, cosa fare quando Information Expert porterebbe ad una soluzione idonea?
- **Soluzione**: si assegni un insieme di funzionalità altamente coeso ad una classe "inventata" per tale scopo.
- **Discussione**: progettazione si basa su due tipi di oggetti:
  - decomposizione rappresentazionale: partono dal dominio e LRG
  - decomposizione comportamentale: sono invenzioni per rappresentare comportamenti e algoritmi
- **Controindicazioni**: rischio di sfociare verso progettazione procedurale
- **Vantaggi**: può favorire riuso

Polymorphism

- **Problema**: come gestire alternative basate sul tipo? Come creare componenti software inseribili?
- **Soluzione**: uso di operazioni polimorfe
- **Discussione**: si possono usare sia meccanismi di ereditarietà che definizione di specifiche interfacce
- **Controindicazioni**: rischio di abusare del meccanismo per prevedere necessità future
- **Vantaggi**: flessibilità e facilità di estensione

Indirection

- **Problema**: Dove assegnare una responsabilità, per evitare l'accoppiamento diretto tra due (o più) elementi? Come disaccoppiare degli oggetti in modo da sostenere un accoppiamento basso e mantenere alto il potenziale di riuso?
- **Soluzione**: Assegna la responsabilità a un oggetto che medi tra altri componenti o servizi, in modo che non ci sia un accoppiamento diretto tra di essi. Si crea indirezione tra i componenti
- **Discussione**: diversi design pattern in qualche modo forniscono meccanismi di indirezione (adapter, façade, observer, proxy)
- **Controindicazioni**: può rendere il codice meno comprensibile
- **Vantaggi**: ridotto accoppiamento

Protected Variations

- **Problema**: come progettare oggetti, sottosistemi, e sistemi in modo tale che le variazioni o l'instabilità in questi elementi non abbiano un impatto indesiderato su altri elementi?
- **Soluzione**: identifica i punti in cui sono previste variazioni o instabilità e poi assegna delle responsabilità per creare un "guscio" stabile attorno a questi punti
- **Discussione**: in realtà è un principio noto di progettazione OO che è generalizzazione di altri principi.
  - interfacce, polimorfismo, indirezioni sono meccanismi che supportano PV
  - Lookup di servizi
  - progettazione riflessiva
  - principio d sostituzione di Liskov
  - legge di Demetra o principio "don't talk to stranger"
- **Controindicazioni**: rischio di generalizzazione dell'evoluzione con incrementata complessità del sistema
- **Vantaggi**: facilità di modificare il sistema. basso accoppiamento, client più isolati, più bassi costi di gestione delle modifiche