Programmazione II

16. Design Pattern

https://www.baeldung.com/design-patterns-series

https://refactoring.guru/design-patterns

https://github.com/iluwatar/java-design-patterns

Dragan Ahmetovic

Motivazione

Problema di scalabilità del progetto

Per progetti grandi, che coinvolgono team numerosi, **serve separare le aree di intervento**

- **Contesto** - spesso, diversi software presentano **esigenze simili**
  - App con un'interfaccia utente nei quali si ha una **business logic** e una **logica di presentazione**
  - Problemi molto simili - es: la necessità di eseguire del codice quando avviene qualche evento
- **Esigenze**:
  - Strutturare il codice per permettere a **diversi sviluppatori** di lavorare solo su **parti limitate del codice**
  - Necessità di risolvere **problemi tipici in maniera coerente ed efficace**
  - Facile **testing, riuso e manutenibilità** dei diversi moduli del codice e del software complessivo
- **Soluzione** - `**pattern**` di sviluppo, ovvero modi **ben noti** per risolvere problemi
  - **Pattern architetturali**: modi di **organizzare il codice** secondo una struttura standardizzata
  - **Pattern di design**: modi di **risolvere problemi** tipici in maniera simile

Pattern architetturali

Che cosa sono?

Metodologie per **strutturare il codice** e **separare moduli** (gruppi di classi) secondo i loro **compiti**

- Esigenza di **limitare le dipendenze tra classi**, con coupling solo dove necessario
  - Il pattern definisce i **moduli**, il loro **scopo** e come avviene la **comunicazione tra i moduli**
  - Si tende a **separare moduli** dedicati ad un **compito specifico** (es: accesso ad un database)
- Un caso d'uso comune è l'**organizzazione di software con un'interfaccia utente**
  - Obiettivo: **separazione della business logic e la logica di presentazione**
- Motivazioni:
  - Permettere lo **sviluppo** e la **modifica** delle due parti in maniera più **indipendente** possibile
  - **Sostituzione del layer di presentazione** con interfacce diverse (web, mobile, text, gui)
  - **Minimizzare la conoscenza della business logic** richiesta da chi si occupa dell'**interfaccia**
  - Esempi: Model-View-Controller, Model-View-Presenter, Model-View-ViewModel
- Serve una discreta complessità di progetto e di interfaccia, quindi non li tratteremo

Design pattern

Che cosa sono?

Tecniche **standard** per risolvere in maniera simile i **problemi comuni** in fase di implementazione

- **Metodologia definita** per la risoluzione del problema
  - Struttura delle **componenti**, delle loro **interazioni** e del **flusso di esecuzione** sono ben definite
  - I **passi specifici** devono però essere **adattati caso per caso** (Non sono degli algoritmi)
- **Classificazione** degli design pattern
  - **Creational** pattern - per la **creazione di nuovi oggetti** riutilizzabili
    es: **Singleton** - possibile una sola istanza di classe, accessibile globalmente
  - **Structural** patterns - per la **strutturazione di oggetti e classi**
    es: **Decorator** - possibile **estendere i comportamenti** di una classe per **composizione e delega**
  - **Behavioral** patterns - per la definizione dei **comportamenti degli oggetti**
    es: **Iterator** - possibile **scorrere collezioni**, un item alla volta
    es: **Observer** - possibile **notificare** una serie di **subscriber** quando avviene qualcosa

Singleton

Caratteristiche del Pattern Singleton

- **Caso d'uso**: fornire un'istanza **unica** e **globale** di una data Classe
  - **accesso controllato a una risorsa** (es: database)
- **Come implementare**:
  - **Costruttore privato**
  - **Metodo di costruzione** `static` richiama il costruttore privato
  - Crea un'istanza della classe e la assegna a una **variabile di classe** (`static`)
  - Se la variabile di classe è **già assegnata** invece **restituisce quella**

Singleton

Esempio Singleton

```
1  public class Classe {
2    private static Classe singleton;
3    private String info = "Default message";
4
5    private Classe(){}
6
7    public static Classe getInstance() {
8      if(singleton == null)
9          singleton = new Classe();
10
11     return instance;
12   }
13
14   public String getInfo() {
15     return info;
16   }
17
18   public void setInfo(String info) {
19     this.info = info;
20   }
21
22   public static void main(String[] args) {
23     Classe s1 = Classe.getInstance();
24     System.out.println(s1.getInfo()); //"Default message"
25     Classe s2 = Classe.getInstance();
26     s2.setInfo("New message");
27     System.out.println(s1.getInfo()); //"New message"
28   }
29 }
```

Decorator

Caratteristiche del Pattern Decorator

- **Caso d'uso**: modificare/estendere comportamenti di un oggetto in maniera **dinamica**
  - anche se **già istanziato** (se no bastava una sottoclasse)
  - anche **in più modi** (come faccio se ereditarietà permette un solo genitore?)
- Come implementare:
  - **Interfaccia Comune** che definisce i comportamenti d'interesse
  - **Classe Base** che si **vuole estendere** implementa **Interfaccia Comune**
  - **Classe Decoratore** implementa **Interfaccia Comune** ma in costruzione prende istanza di **Classe Base**
  - Per **composizione Classe Decoratore** fa da wrapper a **Classe Base**
  - Per **delega Classe Decoratore** eredita (possibilmente estendendoli) i comportamenti di **Classe Base**
- Possibile avere **Classe Decoratore astratta**
  - con varie implementazioni concrete, anche impilabili

Decorator

Esempio Decorator - **Interfaccia Comune**

```
interface Bevanda {
  public String effetto();
}
```

Esempio Decorator - **Classe Base**

```
public class Caffe implements Bevanda {
  public String effetto() {
    return "insonnia";
  }
}
```

Esempio Decorator - **Classe Decoratore**

```
public class Zuccherata implements Bevanda {
  Bevanda b;

  public Zuccherata(Bevanda b) {
    this.b = b;
  }

  public String effetto() {
    return b.effetto() + " e diabete";
  }

  public static void main(String[] args) {
    Bevanda c = new Caffe();
    System.out.println(c.effetto()); //insonnia
    caffe = new Zuccherata(c);
    System.out.println(c.effetto()); //insonnia e diabete
  }
}
```

Observer

Caratteristiche del Pattern Observer

- **Caso d'uso**: **notificare** (attivare comportamenti di) diversi oggetti quando avviene un **evento**
  - es: allarme viene segnalato via sms, mail e sirena acustica
  - es: quando viene pubblicato un post, manda notifica push a tutti i follower
- Come implementare:
  - **Interfaccia Listener** definisce il metodo dell'oggetto **Ricevente** che sarà attivato dalla notifica
  - Ciascun **Ricevente** implementa l'**Interfaccia Listener** definendo il proprio metodo **update()**
  - Il **Mittente** mantiene una lista di **subscriber Riceventi**
  - (ci deve essere un meccanismo di subscribe/unsubscribe)
  - Quando avviene l'evento d'interesse, il **Mittente** notificherà il metodo **update()** di ciascun **Ricevente**

Observer

Esempio Observer - Listener

```
interface Listener<T> {
  public void update(T t);
}
```

Esempio Observer - EventListener

```
public class Ricevente implements Listener<String> {

  public void update(String t) {
    System.out.println("Ricevente sente: " + t);
  }
}
```

Esempio Observer - EventGenerator

```
public class Mittente<T> {
 ArrayList<Listener<T>> listeners = new ArrayList<>();

 public void subscribe(Listener<T> l) {
   listeners.add(l);
 }

 public void unsubscribe(Listener<T> l) {
  listeners.remove(l);
 }

 public void notifyAll(T event) {
  for(Listener<T> listener : this.listeners)
   listener.update(event);
  }
}
```

Programmazione II

16. Design Pattern

https://www.baeldung.com/design-patterns-series

https://refactoring.guru/design-patterns

https://github.com/iluwatar/java-design-patterns

Dragan Ahmetovic