Design pattern – III

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 11
26/4/2004

Scaletta

- Riassunto
- Analisi dei pattern strutturali
- Intermezzo: Javadoc
- Altri pattern: i pattern creazionali
  8. Factory Method
  9. Abstract Factory
  10. Singleton
  11. Prototype
  12. Builder

Riassunto

- Introduzione ai pattern di progetto
- I 7 pattern strutturali
  1. Adapter (Adattatore)
  2. Façade (Facciata)
  3. Composite (Composto)
  4. Decorator (Decoratore)
  5. Bridge (Ponte)
  6. Proxy (Proxy)
  7. Flyweight (Peso piuma)

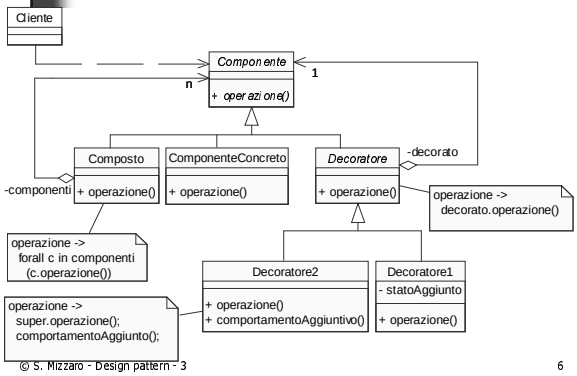
Analisi dei pattern strutturali

- Ci sono similitudini, ma sono ≠
- Adapter e Bridge
  - Entrambi introducono un livello di indirezione e usano la delega,
  - Obiettivi e tempi ≠, spesso usati insieme
- Adapter e Façade
  - Façade come adattatore per tanti oggetti?
  - No, Façade definisce una nuova interfaccia, mentre Adapter usa interfacce preesistenti

Composite e Decorator

- Composite ≠ Decorator (diagrammi simili)
- Complementari, spesso usati insieme
- Dal punto di vista del pattern Decorator, un oggetto `Composto` è un `ComponenteConcreto`
- Dal punto di vista del patter Composite, un oggetto `Decoratore` è una foglia (`Leaf`)

Decorator+Composite



Proxy e Decorator

- Entrambi
  - Usano la composizione di un oggetto
  - Forniscono al cliente un'interfaccia = a quella dell'oggetto
- Proxy (a differenza di Decorator):
  - non aggiunge/rimuove proprietà dinamicamente
    - è l'oggetto a decidere le funzionalità
    - (nel Decorator il decoratore aggiunge funzionalità!)
  - non consente composizione ricorsiva
    - la relazione fra proxy e oggetto è statica, compile time
    - Decorator gestisce aggiunta di funzionalità al run-time

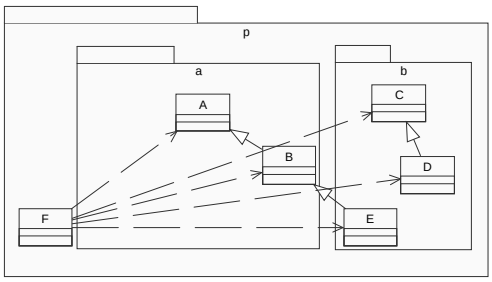
Javadoc

- Generazione automatica della documentazione
- A partire dai commenti `/** ... */`

  `javadoc [<nomefile> | <nomepackage>]*`

  produce la documentazione per *`<nomefile>`* e/o *`<nomepackage>`*
- Tipicamente in HTML (configurabile: doclet)
- Documentazione API

Esempio



Esempio

- Nel package `p.a` (directory `./p/a`) ci sono 2 classi `A` e `B` (sottoclasse di `A`)
- Nel package `p.b` (directory `./p/b`) ci sono 3 classi `C`, `D` (sottoclasse di `C`) ed `E` (sottoclasse di `B`)
- Nel package `p` (directory `./p`) c'è la classe `F` (che usa tutte le altre classi)

Generazione della documentazione

```
>javadoc -verbose -d doc p p.a p.b
```

- Facciamolo!
- Esaminiamola velocemente…
- Aggiungiamo commenti…

Documentazione e OO

- Documentazione utile anche durante la programmazione
  - Per classi, interfacce, metodi, … definite dal programmatore (trovare le definizioni nella gerarchia)
  - Per le API
- Scrivere (e generare) la documentazione insieme al codice!!
  - Un commento per ogni classe, interfaccia, metodo o attributo `public`

I (5) pattern creazionali

- Astrazione sul processo di creazione di istanze
- "Qualcosa di più astratto della `new`"
- Quando faccio una `new`:
  - Devo sapere
    - Il tipo della variabile
    - la classe di cui creare l'istanza e gli argomenti del costruttore
  - `A x = new B(…)`
- Evitano riferimenti espliciti a classi concrete da istanziare

8. Factory Method

- Scopo
  - Delegare l'istanziazione di una classe a una propria sottoclasse
  - La(e) sottoclasse(i) decide(ono) quale istanza creare, quale costruttore chiamare
- Usato tipicamente nei Framework OO

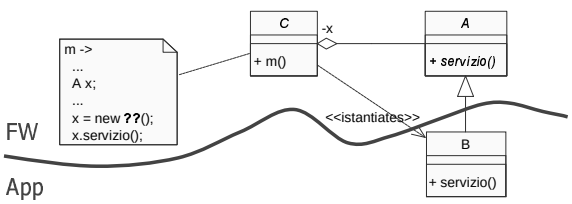
Cos'è un Framework OO

- Insieme di classi
- Differenza da "libreria" software:
  - Libreria: il programmatore di un'applicazione chiama i metodi
  - FW: il programmatore di un'applicazione
    - scrive metodi che vengono chiamati dai metodi del FW
    - "riempie i buchi"
    - "Hollywood principle": don't call us, we'll call you
- Esempi:
  - Gestione eventi AWT/Swing, applet, servlet, …

Problema tipico in un FW

- Il programmatore del FW scrive un metodo in una classe `C` …
- … il metodo deve creare, in un certo punto preciso, un'istanza di una sottoclasse `B` di un'altra classe `A`…
- … ma quale sottoclasse `B` usare è ignoto (perché non deciso dal programmatore del FW ma dal programmatore dell'applicazione!)

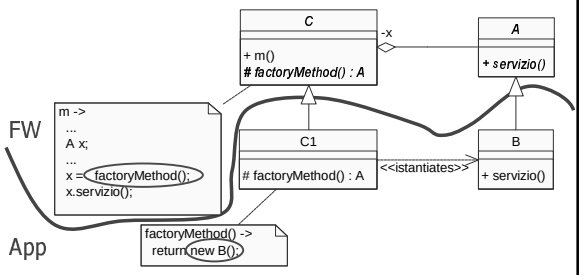
Il problema



Soluzioni?

- Prima (non) soluzione:
  - il programmatore del FW rinuncia e fa fare tutto al programmatore dell'applicazione
- Seconda soluzione:
  - il programmatore del FW scrive quello che può/sa: un'invocazione a un metodo astratto `factoryMethod` che deve creare l'istanza…
  - … e chiede al programmatore dell'applicazione di scrivere, oltre alla sottoclasse `B`, anche una sottoclasse `C1` di `C` che implementa il metodo `factoryMethod`

La soluzione



Commenti

- Invece di avere la conoscenza esplicita sulla creazione in una classe (`C`)…
- …la delego a una sottoclasse (`C1`)
- Motivo: al "tempo di scrittura" della classe (`C`) non so che sottoclasse istanziare

Varianti

- **`factoryMethod`** potrebbe non essere astratto ma fornire un'implementazione di default (eventualmente sovrascritta)
- Le sottoclassi di **`A`** sono note al tempo di scrittura di **`m`**, ma non si sa quale usare

Scopo di Factory Method [GoF]

- Definire un'interfaccia per la creazione di un oggetto (il **`factoryMethod`**)…
- … lasciando alle sottoclassi la decisione sulla classe che deve essere istanziata.
- Consente di deferire l'istanziazione di una classe alle sottoclassi

Scopo di Factory Method [M]

- Come sviluppatori di classi, di solito fornite un costruttore per permetterne l'istanziazione
- A volte il cliente non ha bisogno di (o non deve, o non può) sapere quale classe istanziare
- Con il Factory Method, voi programmatori:
  - definite l'interfaccia per la creazione
  - mantenete il controllo su quale classe istanziare

Cosa non è un Factory Method

- Nelle API Java ci sono molte situazioni in cui la creazione di oggetti avviene attraverso metodi e non costruttori
- Non tutti sono Factory Method!
  - **`toString()`**: crea un'istanza di **`String`**
  - **`java.util.Arrays.asList(Object[])`**:
  - **`javax.swing.BorderFactory.createXXXBorder(…)`**
- (so sempre il tipo dell'istanza creata!)
- Ex.: Iterator

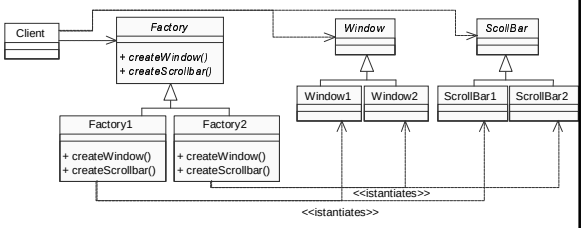
9. Abstract Factory

- Scopo
  - Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o mutuamente dipendenti
  - senza specificare le loro classi concrete

Esempio

- Stiamo progettando un FW per GUI
  - Multipiattaforma
    - Se eseguito su architettura1, **`Window1`** e **`ScrollBar1`**
    - Se eseguito su architettura2, **`Window2`** e **`ScrollBar2`**
  - Però indipendente dalla piattaforma
    - Il **`Client`** (che creerà/istanzierà quelle classi) non deve sapere quali istanzia
    - Bisogna evitare che il **`Client`** accoppi (sbagliando) **`Window1`** e **`ScrollBar2`**
  - Usiamo una "fabbrica astratta"

La fabbrica degli oggetti che ci servono



Cosa fa il `Client`

- Senza la fabbrica:

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

- Con:

```
Factory f = new Factory1();
Window w = f.createWindow();
...
ScrollBar s = f.createScrollBar();
```

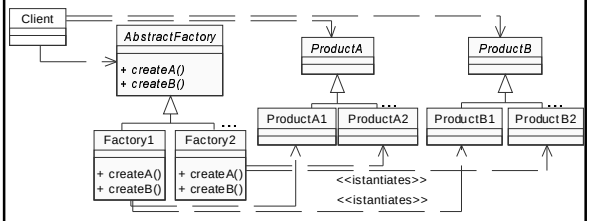

Differenze

- Senza la "fabbrica"
  - **`Client`** deve "conoscere" **`Window1`** e **`ScrollBar1`**
  - **`Client`** crea una **`Window1`**
  - Poi crea una **`ScrollBar1`**
  - **`Client`** ha la responsabilità di accoppiare **`Window1`** e **`ScrollBar1`**
- Con la "fabbrica"
  - **`Client`** chiede una fabbrica "di tipo 1"
  - Poi chiede alla fabbrica una **`Window`**
  - Poi chiede alla fabbrica una **`ScrollBar`**
  - La responsabilità di accoppiare **`Window1`** e **`ScrollBar1`** è delegata alla fabbrica

Commenti

- Separazione delle responsabilità
  - Disaccoppia la responsabilità della creazione (nella fabbrica)…
  - …dalla responsabilità dell'uso (nel cliente)
- Perché "astratta"
  - Il cliente conosce solo classi astratte

Diagramma Abstract Factory



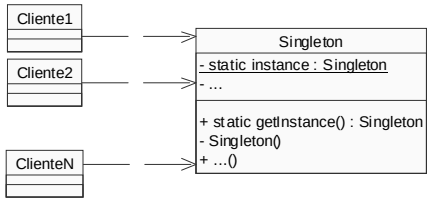
10. Singleton (Singoletto)

- Scopo
  - Assicurare che una classe abbia un'unica istanza
  - Fornire un punto di accesso globale all'unica istanza
- Motivi/Esempi
  - Password
  - S.O.: stampante, file system, …
  - Azienda
- Pattern facile, molto orientato al codice

Come si fa?

- Si delegano alla classe le responsabilità di
  - consentire un'unica istanza
  - consentirne l'accesso
- Impedire creazioni di più istanze
  - Definire un singolo costruttore privato
  - (se ne definite/ereditate altri non privati non va)
- Consentire l'accesso
  - Definire un metodo pubblico che crea o restituisce un riferimento all'unica istanza

Diagramma Singleton



Codice

```
class Singleton {
    private static Singleton instance;
    private int i = 1;
    private Singleton() {}
    public static Singleton getInstance() {
      if (instance==null){
        System.err.println("Istanza creata");
        instance = new Singleton();
      }
      return instance;
    }
    public void set(int x) {i=x;}
    public String toString() {return ""+i;}
}
```


Uso Singleton

```
class ProvaSingleton {
  public static void main (String[] args) {
    Singleton s = Singleton.getInstance();
    System.out.println(s);
    s = Singleton.getInstance();
    System.out.println(s);
    Singleton s1 = Singleton.getInstance();
    s1.set(2);
    System.out.println(s);
  }
}
```

```
>java ProvaSingleton
Istanza creata
1
1
2
```

Commenti

- E se `getInstance` non fosse `static`?
  - Come farebbe il cliente a ottenere un'istanza? Il cliente all'inizio ha solo riferimenti alla classe…
- Quando creare/inizializzare l'istanza unica?
  - Early/Late initialization, solito trade-off
  - Early: potrei non avere le info necessarie…
- Rispetto a usare metodi di classe
  - Più OO
    - Subclassing, polimorfismo, …

Singleton e Multithreading…

```
class Singleton {
    private static Singleton instance;
    private static final Object lock =
                                        Singleton.class;
    private int i = 1;
    private Singleton() {}
    public static Singleton getInstance() {
      syncronized (lock) {
        if (instance==null){
          System.err.println("Creata l'istanza");
          instance = new Singleton();
        }
      }
      return instance;
    }
    public void set(int x) {i=x;}
    public String toString() {return ""+i;}
}
```


Singleton e subclassing

```
class SubSingleton extends Singleton {
  public SubSingleton(int i) {
    ...
  }
}
```

```
>javac SubSingleton.java
SubSingleton.java:2: Singleton() has private access in
  Singleton
    public SubSingleton(int i) {}
                               ^
1 error
>
```

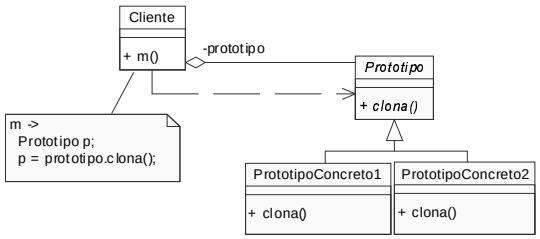

Riconoscere il Singleton

- `java.lang.System`
  `java.lang.Math`
  - No, hanno il costruttore privato, ma sono utility (metodi `static`)
- `System.out`, `System.in`
  - No, sono oggetti, non classi…

11. Prototype

- Scopo
  - Creare una nuova istanza copiandola da un oggetto preso ad esempio
- I pattern creazionali precedenti isolano il cliente di un oggetto dalla chiamata al suo costruttore
  - Metodi che istanziano la classe appropriata
- Prototype: metodo che restituisce una copia dell'istanza

Diagramma Prototype



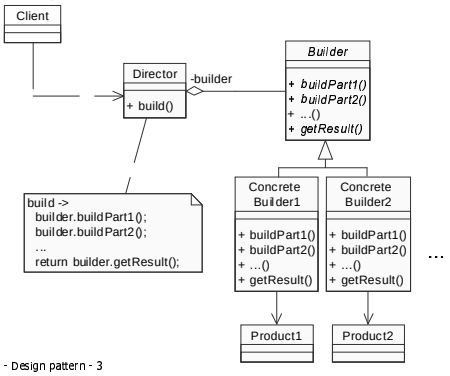
Commenti

- Vantaggi
  - Il Client non "conosce" le classi concrete
    - "Parla" solo alla classe base `Prototipo`
    - `prototipo` può essergli passato come parametro
    - Posso aggiungere altre classi concrete
- Visto di corsa (in Java c'è `Object.clone()`)

12. Builder

- Scopo
  - Costruzione passo-passo di un oggetto complesso…
  - …separando i vari passi del processo di costruzione…
  - …dall'oggetto costruito e dal modo con cui le varie parti sono assemblate
- Mentre il costruttore di una classe specifica la logica della costruzione di un'istanza,
- Builder sposta il codice per la creazione delle istanze al di fuori della classe da istanziare
  - Non più nel costruttore

Diagramma Builder



Funzionamento

- Il `Client` crea l'oggetto `Director` e lo configura per farlo operare con il `Builder` voluto
- `Director` informa il `Builder` ogni volta che una parte del `Product` deve essere costruita
- `Builder` riceve le richieste da `Director` e aggiunge le parti a `Product`
- Alla fine, il `Client` recupera il `Product` creato

Commenti

- I metodi `buildPart` potrebbero essere vuoti (`{}`) anziché astratti
- Rispetto agli altri pattern creazionali
  - L'accento è sul processo di costruzione
  - `Product` è costruito non in un sol colpo, ma passo dopo passo sotto il controllo di `Director`
  - L'interfaccia di `Builder` rispecchia il processo di costruzione
- Builder spesso usato per costruire Composite

Riassunto

- Javadoc
- Analisi dei pattern strutturali [GoF, p. 219]
- Altri pattern: i 5 pattern creazionali
  8. Factory Method
  9. Abstract Factory
  10. Singleton
  11. Prototype
  12. Builder

Esercizio

- Programma di grafica
- "Palette" figure di base
- "Palette" figure create dall'utente
- Gestione gruppi

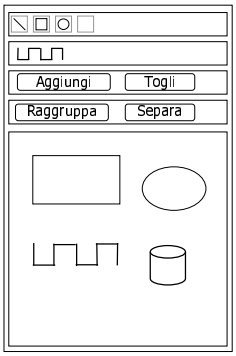


Diagramma (1/2)

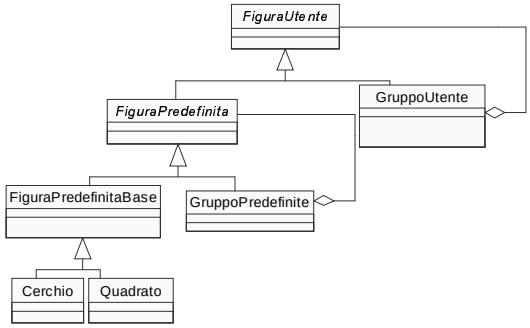


Diagramma (2/2)