



Java Implementation Patterns

Mauro Gagni



Alcuni Principi

- Aderire allo stile Originale
- Siate Essenziali
 - Semplicità (dell'implementazione)
 - Chiarezza (di scopo)
 - Completezza (delle funzionalità)
 - Consistenza (delle convenzioni)
 - Robustezza (del codice)
- Fallo giusto la prima volta
- Documenta le variazioni dalla consuetudine



Convenzioni

- Formattazione
- Naming
 - Funzioni e variabili
 - Nomi che hanno un significato
 - Nomi Noti (concordati)
 - Non abbreviate nomi
 - Capitalizza solo la prima lettera
 - Costanti Uppercase



Principio Aperto Chiuso

- Le entita' del software (Classi, Moduli, Funzioni, etc..) dovrebbero essere

- Aperte alle estensioni e
- Chiuse alle modifiche



Implementation Patterns

- Classi Base come **final**
- Classi Piccole
- Metodi Piccoli
- Attenzione l'overloading dei Metodi
- Rendete pubblico il necessario
- Usate il polimorfismo al posto di **instanceof**

Implementation



- Type Safety

- Create Static type checking
- Incapsulate Enumerazioni in una classe

- Espressioni e Statements

- Fattorizzate! Scrivete una volta sola il codice
- Usate sempre I block statements nel flusso
 - I.e. `if () {...}` `while () {...}`
- Chiarificate l'ordine delle espressioni con parentesi
- Ricordatevi il `break;` in un case
- Usate sempre `equals()` e mai `==`



Implementation

- Costruzione
 - Costruite gli oggetti in uno stato valido!
 - Non chiamare metodi non **final** da un costruttore
 - Accedete direttamente a variabili interne non ai set-get
 - Create una gerarchia di costruttori
- Gestione delle Eccezioni
 - Non abortite silenziosamente le eccezioni
 - Usate **finally** per rilasciare le risorse

Behaviour Messages



- Scegliere il messaggio
 - Come esegui una di molte alternative?
 - Spedisci un messaggio all'oggetto; lascia la classe dell'oggetto scegliere il messaggio (polimorphism)

if (instance of)

 Call f1

else If ()

 Call f2

else If

 Call f3

Parametri di Default



Come setto i parametri di default?

Fai l'overload della funzione con più parametri

- costruttori
- metodi



Conversione da costruttore

Come rappresenti la conversione di un oggetto da un formato ad un'altro?

Fornisco un costruttore nella classe destinazione che prende come parametro la classe sorgente.

```
public Manager(Impiegato pImpiegato)
{
    nome = pImpiegato.getName();
    salario = pImpiegato.getSalario() * 2;
}
```

- Vantaggi
 - Il processo di conversione rimane isolato nel manager
 - L'impiegato non deve conoscere il costruttore del manager



Metodo di conversione

Come rappresenti la conversione di un oggetto da un formato ad un'altro?

Utilizza la “Conversione da costruttore”.

Altrimenti crea un metodo `asTargetClass()` e fai in modo che ritorni la classe destinazione.

- Evita di chiamare la classe con `new Target(param1, param2...)`
- Usa Conversione da Costruttore `new Target(this);`

Eviti di legarti alla classe destinazione in modo che possa avere più flessibilità e controllo nel sistema.

Metodo di conversione - Note



int to String

- `"" + iValore`
- `new Integer(iValore).toString();`
- `String.valueOf(iValore);`

String to int

- `new Integer(sValore).intValue();`
- `Integer.valueOf(sValore).intValue();`
- `Integer.parseInt(sValore);`

Semplificare la costruzione di oggetti



Come semplificare la costruzione di oggetti?

Crea un metodo che provvede a creare la nuova istanza

```
frase.aggiungi( "Nuova frase" );

public aggiungi( String pFrase )
{
    return new Frase( this, pFrase);
}
```

Utile solo per operazioni molto frequenti



Metodo oggetto

Come codifichi un metodo in cui ci sono piu' linee di codice che condividono argomenti e variabili temporanei?

- Crea una inner class con il nome del metodo.
- Dichiarare una variabile per ogni variabile temporanea utilizzata nel metodo
- Passa quei valori al costruttore della inner class
- Definisci un metodo ad esempio `processa ()`
- Applica "Metodo composto" al metodo originale

Oggetto come parametro



Come codifichi un metodo in cui ci sono piu' linee di codice che condividono argomenti e variabili temporanee?

- Questo metodo è l'inverso di "Metodo oggetto"
- Applica "Metodo composto" al metodo originale
- Crea una inner class chiamata `parametri`.
- Dichiarare una variabile per ogni variabile temporanea utilizzata nel metodo in `parametri`.
- Passa `parametri` a tutti i metodi della composizione
- Usa le variabili temporanee contenuti nella inner class

Oggetto come parametro (2)



```
private CalcualteMonthlyPay()  
{  
    Parameters params = new Parameters();  
  
    params.correction = 1.2;  
    params.rate = 2.3;  
    params.preiodStart = getFirstDay();  
    params.preiodStart = getLastDay();  
  
    calculateBiWeekly( params );  
    calculateTax( params );  
    calculateAdjustments( params );  
    calculateMedicareTax( params );  
    subtractPretaxDeduction( params );  
}
```

Generics



Una delle grosse mancanze di Java sino a... ieri!

- Le collections usano Object come tipo primario, il che Implica:
 - possibilita' di RuntimeException
 - Complessita' del codice
- Di solito cercava di aggirare l'ostacolo con un implementation pattern
 - Si estendeva una collezione
 - Si faceva l'overloading degli operatori di put/add
 - O si aggiungevano operatori specifici,
 - ma nessuno poteva garantire il loro uso da parte del programmatore

Generics 2



Oggi...

```
// Removes 4-letter words from c; elements must be strings
static void exclude(Collection c)
{
    for (Iterator i = c.iterator(); i.hasNext(); )
    {
        if(((String) i.next()).length() == 4)
        {
            i.remove();
        }
    }
}
```

Generics 3



Ora esistono I Generics...

- Nuovo con JDK 1.5

```
// Removes 4-letter words from c
static void exclude(Collection<String> c)
{
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
    {
        if (i.next().length() == 4)
        {
            i.remove();
        }
    }
}
```

Generics 4



Ora esistono I Generics...

- Ora il compilatore e' in grado di controllare i tipi di oggetti
 - Il casting viene fatto direttamente dal compilatore

Generics 5



Come sono cambiate le collections

```
interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

Generics 6



Come sono cambiate le collections

```
interface Map<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

Generics un esempio



Un esempio di runtime horror

```
List ys = new LinkedList();
ys.add("zero");
List yss;
yss = new LinkedList();
yss.add(ys);

String y = (String)
    ((List)yss.iterator().next()).iterator().next();

Integer z = (Integer)ys.iterator().next();
// run-time error!
```

Generics un esempio



Oggi invece...

```
List<String> ys = new LinkedList<String>();  
ys.add("zero");  
List<List<String>> yss;  
yss = new LinkedList<List<String>>();  
yss.add(ys);  
  
String y =  
    yss.iterator().next().iterator().next();  
  
Integer z = ys.iterator().next();  
// compile-time error
```

Enumeration



Come codifichi una lista di costanti di un dominio?

- Di solito

```
public class Coin {  
    public static final int PENNY    = 1;  
    public static final int NICKEL   = 5;  
    public static final int DIME     = 10;  
    public static final int QUARTER  = 25;  
}
```



Enumeration Svantaggi?

Enums Pattern Savantaggi?

- Non e' type safe
- Compila le costanti nelle classi destinazione
- I valori stampati hanno poco significato



Enumeration Pattern

Come codifichi una lista di costanti di un dominio?

- Usa l'enumeration pattern

```
Class Coin {  
    Coin(int value) { this.value = value; }  
    private final int value;  
    public int value() { return value; }  
};
```

```
public color( Coin pCoin ) { ...
```

Enumeration Pattern Svantaggi?



Enums Pattern Savantaggi?

- Forma prolissa (e' un Workaround)
- Non puo' essere usato negli switch

Nuova Enum



Come codifichi una lista di costanti di un dominio?

➤ Nuovo con JDK 1.5

```
enum Suit { clubs, diamonds, hearts, spades };
```

```
for (Suit suit : Suit.values())  
    System.out.println(suit);
```

...

clubs

diamonds

hearts

spades

Nuova Enum (2)



```
enum Coin {  
  
    penny(1), nickel(5), dime(10), quarter(25);  
  
    Coin(int value) { this.value = value; }  
    private final int value;  
    public int value() { return value; }  
  
};
```

Nuova Enum (3)



```
for (Iterator<Coin> i =  
    Arrays.asList(Coin.values()).iterator();  
    i.hasNext(); )  
{  
    Coin coin = i.next();  
    System.out.print(coin + " ");  
}
```

penny nickel dime quarter

Nuova Enum (4)



```
public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.VALUES)
            System.out.println(c + ": \t"
                + c.value() + "¢ \t" + color(c));
    }
    private enum CoinColor { copper, nickel, silver }
    private static CoinColor color(Coin c) {
        switch(c) {
            case penny: return CoinColor.copper;
            case nickel: return CoinColor.nickel;
            case dime:
            case quarter: return CoinColor.silver;
            default: throw new AssertionError("Unknown coin: " +
                c);
        }
    }
}
```

Nuova Enum



```
enum Suit { clubs, diamonds, hearts, spades }
enum Rank { deuce, three, four, five, six, seven, eight, nine, ten,
    jack, queen, king, ace }

List<Card> deck = new ArrayList<Card>();
for ( Suit suit : Suit.VALUES)
    for ( Rank rank : Rank.VALUES )
        deck.add(new Card(suit, rank));

Collections.shuffle( deck );
```

Dispatch Interpretation



Come possiamo far comunicare due oggetti quando uno vuole nascondere la sua implementazione?

- Switch statements negli attributi fanno il lavoro che alcuni oggetti sono troppo pigri per fare
- Se cambia il domino di utilizzo -> la manutenzione diventa difficile, come il debug dato che gli errori accadono a runtime
- Prova a codificare lo switch statement solo una volta

Dispatch Interpretation



Come possiamo far comunicare due oggetti quando uno vuole nascondere la sua implementazione?

- Lo scopo e' fare in modo di accorgersi a compile time di eventuali problemi.
- Lego cosi' le due classi con una interfaccia che definisce il legame
- Se il legame si spezza ce ne accorgiamo subito

Dispatch Interpretation



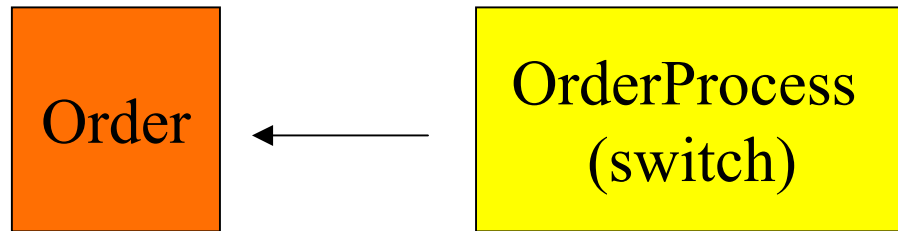
Come possiamo far comunicare due oggetti quando uno vuole nascondere la sua implementazione?

- Incapsula la rappresentazione dentro l'oggetto responsabile.
- Crea una interfaccia che l'oggetto cliente dovrà implementare.
- L'oggetto cliente farà una callback all'oggetto responsabile per uno dei suoi metodi di interfaccia.
- allback all'oggetto respoinsaile per uno dei suoi metodi di interfaccia.

Dispatch Interpretation



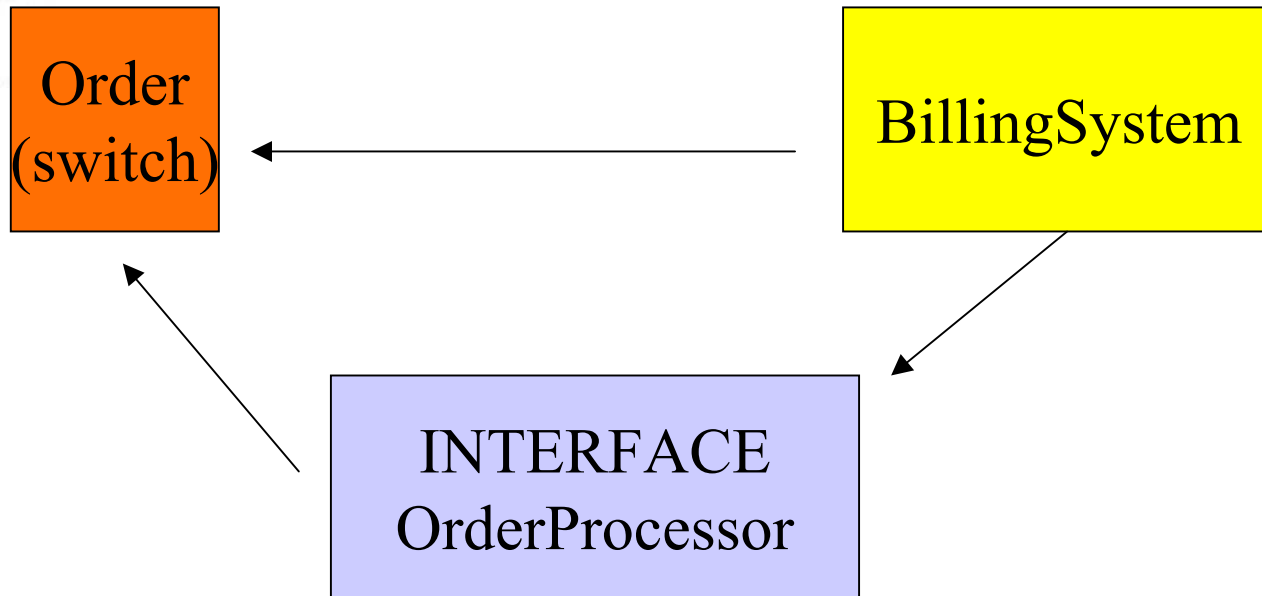
Come possiamo far comunicare due oggetti quando uno vuole nascondere la sua implementazione?



Dispatch Interpretation



Come possiamo far comunicare due oggetti quando uno vuole nascondere la sua implementazione?



Dispatch Interpretation (2)



```
private class Order()  
{  
    private OrderType __orderType;  
  
    Public Order( OrderType pOrderType )  
    {  
        __orderType = pOrderType;  
    }  
    Public OrderType getOrderType()  
    {  
        return __orderType;  
    }  
}
```

```
private class OrderProcess() {  
    Public bill( Order )  
    {  
        switch ( order.getType )  
            case new : call new  
            case cancel : call  
                cancel  
            ...  
    }  
}
```



Dispatch Interpretation (3)

```
Interface OrderProcessor {  
    public processNew( order )  
    public processCancel( order )  
}
```

//order class knows what method to call for its type
//the type is hidden from the Billing System that acts
//as a framework

```
Class Oredr {  
    public processBy( OrderProcessor o ) {  
        switch (order type )  
        ...  
    }  
}
```

Results

1. Il cliente non deve mantenere lo switch perche' ora sono in una classe sola.
2. Si identificano a compile time errori di interfacce non implementate.
3. Si nasconde l'implementazione

```
Class BillingSystem Implements OrderProcessor {  
    public bill( Order pOrder ) {  
        pOrder.processBy( this )  
    }  
}
```



Risorse

www.javaportal.it

www.jia.it

www.jguru.com

java.sun.it

www.javaPortal.it

Grazie!



Mauro Gagni

mauro.gagni@iconbps.it

mauro.gagni@alchemi.it



Elements of Java Style

Allan Vermeulen, et al



Essential Java Style:
Patterns for
Implementation

Jeff Langr