DESIGN PATTERNS – Parte 2

Polimorfismo
Pure Fabrication
Simple Factory/Concrete Factory
Factory Method
Abstract Factory
Protect Variations + Reflection

POLIMORFISMO

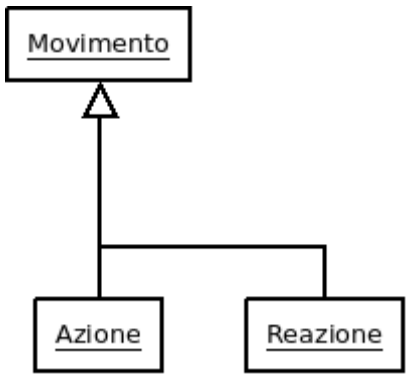
Il polimorfismo è la base della programmazione ad oggetti e si divide principalmente in due utilizzi: l'ereditarietà e l'implementazione.
L'utilizzo di questo pilastro è dettato dai seguenti punti:

1) Ogni comportamento diverso va gestito da un oggetto diverso
2) Tante sotto-classi quanti sono i comportamenti
3) Ogni classe implementa solo i metodi con comportamento diverso

Per comprendere questo concetto fingiamo dunque di avere le seguenti due classi:

```
public class Azione {
     private String descr;

     public void preperazione() { /* … */ }
     public String getDescrizione() { return descr; }
}

public class Reazione {
     private String descr;

     public void contrattacco() { /* … */ }
     public String getDescrizone() { return descr; }
}
```

Come si può vedere le due classi presentano uno stesso metodo comune tra loro, mentre i metodi preparazione e contrattacco sembrano essere comportamenti specifici. Questo ci indica che è possibile utilizzare una super-classe che poi verrà ereditata da Reazione ed Azione.



Che in codice verrà tradotto come:

```
public class Movimento {
     private String descr;

     public String getDescrizione() { return descr; }
}

public class Azione extends Movimento {
     public void preperazione() { /* … */ }
}

public class Reazione extends Movimento {
     public void contrattacco() { /* … */ }
}
```

In questo modo è possibile fare anche eventuali overriding e/o overloading. Questa tecnica ha però dei pro e contro:

PRO: oggetti e metodi più semplici e coesi
PRO/CONTRO: necessita della creazione di una classe per ogni comportamento
CONTRO: vengono a generarsi molte classi. Assicurarsi quindi che sia il comportamento a cambiare e non il valore degli attributi.

Capita a volte però di dover scegliere tra la classe ereditata e l'interfaccia, la sostanziale differenza fra i due costrutti è la seguente:

Interfaccia	Classe
Non instanziabile	Instanziabile (a patto che non sia abstract)
Solo il prototipo dei metodi (metodi astratti)	Implementa i metodi
Solo metodi pubblici	Ha metodi privati
Nessun attributo	Variabili d'istanza

È quindi preferibile usare le interfacce quando si è in dubbio. Le interfacce in più permettono di obbligare l'utente che le utilizza a usare le classi nel modo che vogliamo noi. Prendiamo per esempio il caso della gestione di un database in cui vogliamo salvare dei dati. Questo potrebbe essere fatto attraverso un'interfaccia Persistente:

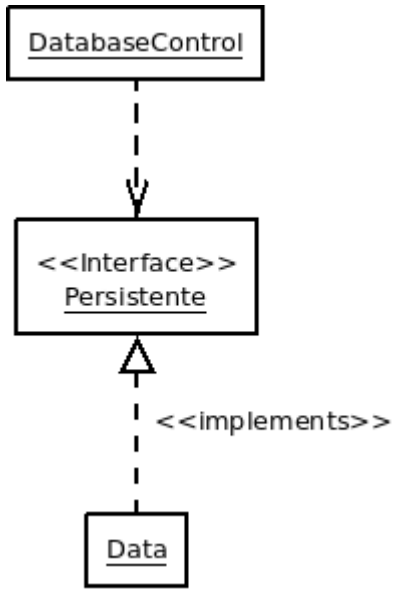
```
public interface Persistente {
     public String[] getProperties();
     public void setProperties(String[] props);
}
```

E con questa interfaccia avremmo anche una classe che gestisce oggetti di

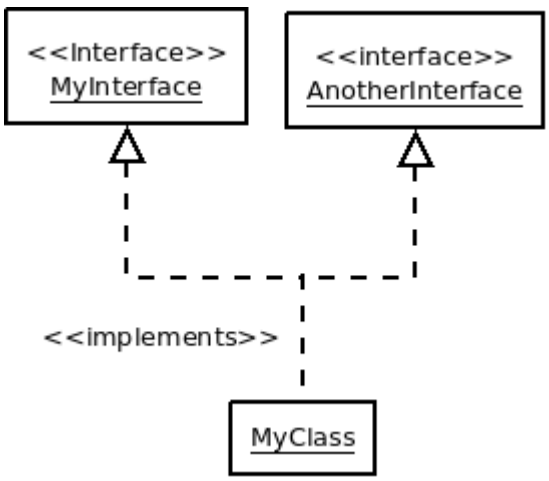
tipo Persistente. Fingiamo quindi che la nostra classe di gestione sia la seguente:

```
public class DatabaseControl{

     public void loadData(int key, Persistente p){
          String[] props = this.loadDataFromDB(key);
          p.setProperties(props);
     }

     public void storeData(int key, Persistente p){
          String[] props = p.getProperties();
          this.storeDataToDB(key, props);
     }

     private String[] loadDataFromDB(int k) { /* … */ }
     private void storeDataToDB(int k) { /* … */ }
}
```

Come possiamo vedere obblighiamo l'utente che vorrà usare la classe Database Control a usare l'interfaccia Persistente. Quindi il nostro diagramma delle classi sarà composto così:



Una classe può comunque implementare tante interfacce quante vuole:



In alcuni linguaggi di programmazione è presente anche l'ereditarietà multipla. Per quei linguaggi dove non è presente è comunque possibile simulare l'ereditarietà multipla attraverso le interfacce.

Per capire ciò immaginiamo di avere una serie di classi (che chiameremo Big, Medium e Small) che avranno i loro metodi (getBig, setBig, getMedium, setMedium e getSmall, setSmall rispettivamente) e una classe Normal che vogliamo che erediti tutte e 3 le classi.

```
public class Big {
     public Object getBig() { /* … */ }
     public void setBig(Object o) { /* … */ }
}

public class Medium {
     public Object getMedium() { /* … */ }
     public void setMedium(Object o) { /* … */ }
}

public class Small {
     public Object getSmall() { /* … */ }
     public void setSmall(Object o) { /* … */ }
}
```

Ora va scelta la classe con più metodi pubblici. Nel nostro caso sono tutte equivalenti, e sceglierne una vale l'altra, quindi sceglieremo Big.

```
public class Normal extends Big {

/******** Metodi ereditati dalla classe Big *******/
     public Object getBig() { /* … */ }
     public void setBig(Object o) { /* … */ }

}
```

Scelta la classe con più metodi pubblici, si fanno delle interfacce che verranno implementate dalle restanti classi con metodi che avranno gli stessi nomi delle classi in questione.

```
public interface SmallInterface {
     public Object getSmall();
     public void setSmall(Object o);
}

public interface MediumInterface{
     public Object getMedium();
     public void setMedium(Object o);
}
```

Queste interfacce ora vanno implementate nelle classe Medium e Small, che all'apparenza non avranno alcuna modifica:

```
public class Medium implements MediumInterface {
     public Object getMedium() { /* … */ }
     public void setMedium(Object o) { /* … */ }
}

public class Small implements SmallInterface {
     public Object getSmall() { /* … */ }
     public void setSmall(Object o) { /* … */ }
}
```

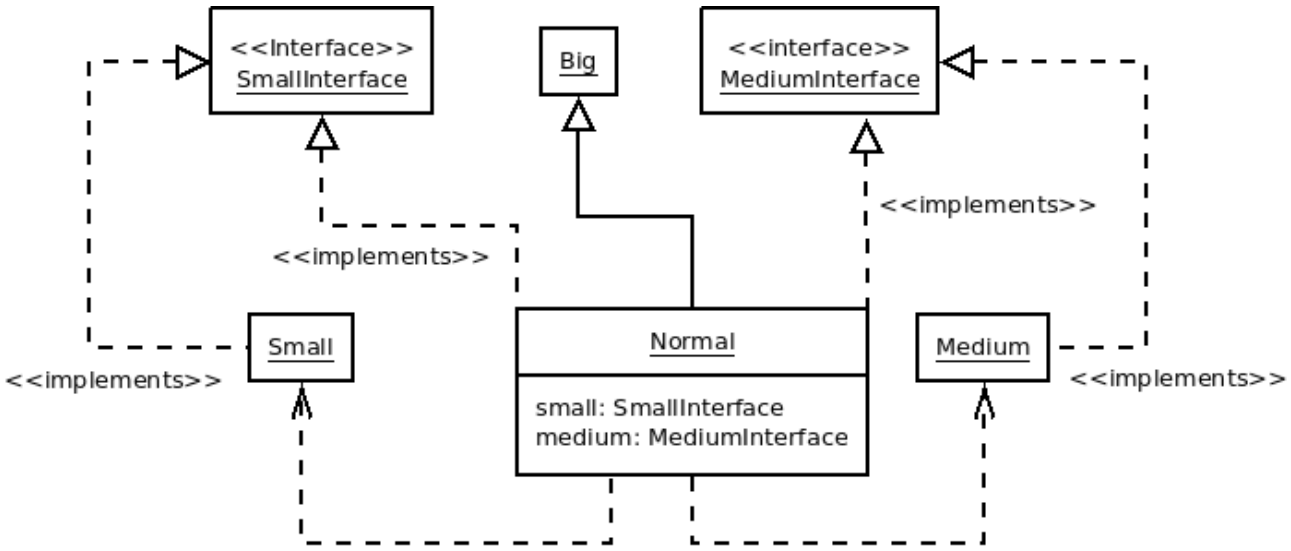
Fatte questo la classe che multi-eredita dovrà avere degli attributi contenente istanze delle classi che dovrebbe ereditare.

```
public class Normal extends Big {
     private SmallInterface small = new Small();
     private MediumInterface medium = new Medium();

/******** Metodi ereditati dalla classe Big *******/
     public Object getBig() { /* … */ }
     public void setBig(Object o) { /* … */ }
}
```

Sempre la classe che multi-eredita ora dovrà implementare le interfacce degli attributi dichiarati precedentemente. Ora i metodi dovranno essere implementati e dovranno quindi fare da wrapper per i metodi omonimi degli attributi. In questa maniera chi userà la classe che multi-eredita sembrerà come se avesse veramente mutli-ereditato invece di aver usato varie interfacce e metodi wrapper.

```
public class Normal extends Big implements SmallInterface,
MediumInterface {
     private SmallInterface small = new Small();
     private MediumInterface medium = new Medium();

/******** Metodi ereditati dalla classe Big *******/
     public Object getBig() { /* … */ }
     public void setBig(Object o) { /* … */ }

/******* Metodi "ereditati" dalla classe Medium ****/
     public Object getMedium() { return mediumn.getMedium(); }
     public void setMedium(Object o) { medium.setMedium(o); }

/******* Metodi "ereditati" dalla classe Small ****/
     public Object getSmall() { return small.getSmall(); }
     public void setSmall(Object o) { small.setSmall(o); }

}
```

L'intero schema delle classi UML sarà quindi il seguente:



PURE FABRICATION

Questo Design Pattern nasce per soddisfare le necessità di dividere la gestione di alcune funzionalità (come la ricerca) che non competono ai comportamenti di una classe.
Per capire ciò partiamo dall'origine del problema. Fingiamo di avere una classe Singleton che ha un attributo Vector result che è il risultato di una qualche ricerca, magari attraverso una query per esempio.

```
public class Singleton {
     private Vector result;

     public void search(String query) { /* … */ }

     public Vector getResult() { return result; }

     public void filterSearch(String query) { /* … */ }

     /* Codice della classe Singleton... */
}
```

Come vediamo questa soluzione porta alcune complicazioni:

1) bassa coesione (non è compito di Singleton di ricercare gli oggetti)
2) ogni oggetto Singleton ha una ricerca interna (e quindi scarso riuso del codice)

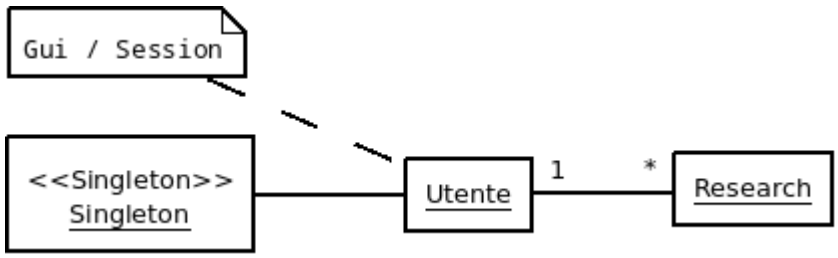
Applicando il desing pattern Pure Fabrication miglioreremo:

1) l'assegnazione delle responsabilità con criteri di Low Coupling e High Coesion

In compenso però avremmo una nuova classe che non rappresenta nessun

oggetto, ma è un puro e semplice artifizio.
In questo modo avremmo una classe Research che fa le ricerche e una classe Singleton che fa le cose che gli competono.

```
public class Research {
     private Vector result;

     public void search(String query) { /* … */ }

     public Vector getResult() { return result; }

     public void filterSearch(String query) { /* … */ }
}

public class Singleton {
     private Research google = new Research();

     /* Codice della classe Singleton … */
 }
```



L'uso del Pure Fabrication non si limita agli algoritmi, ma viene anche usato per la persistenza dei dati e la visualizzazione di essi (per esempio una GUI o una Session per la WebApp).
Purtroppo però il Pure Fabrication ha degli svantaggi:

- necessità di creare una classe in più
- difficoltà nel capire chi è la classe artificiosa
- visione procedurale invece che ad oggetti (raggruppo per funzione invece che per concetto)

SIMPLE FACTORY / CONCRETE FACTORY

Un altro problema che nasce durante la gestione dei dati è quello della loro creazione. Fino ad ora abbiamo visto come maneggiarli, ma chi è per esempio, che dopo averli letti da un database, li deve creare?

Il pattern Information Expert ci potrebbe a fare queste operazioni nell'oggetto stesso, essendo egli quello che ha le informazioni necessarie.

Ma come ora vedremo risulta che non c'è coesione.

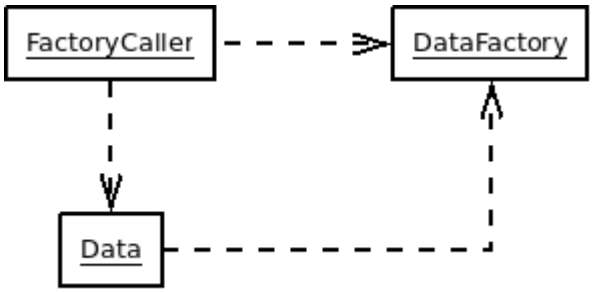
```
public class Data {

     public Data(String[] d) { /* … */ }

     static public Data load(int id)
     {
          String[] data = this.loadData(id);

          return new Data(data);
     }

     static private String[] loadData(int id) { /* … */ }
}
```

Come detto non c'è coesione visto che facciamo fare a Data cose che non sarebbero di sua competenza. Ma non solo, creiamo pure una forte dipendenza tra l'oggetto e la lettura dei dati dal database, e di conseguenza, una forte ripetizione dello stesso codice in più classi.

Come con il Pure Fabrication anche qui creeremo una classe artificiosa per la creazione degli oggetti:

```
public class DataFactory {

     public Data load(int id) { /* … */ }

}
```

La classe che richiama il DataFactory potrebbe aver necessità che i metodi del DataFactory siano statici o che il DataFactory sia un singleton. Questo dipende da cosa si vuole e da una ovvia serie di fattori. Comunque sia, lo schema delle dipendenze tra le classi resta più o meno questo:



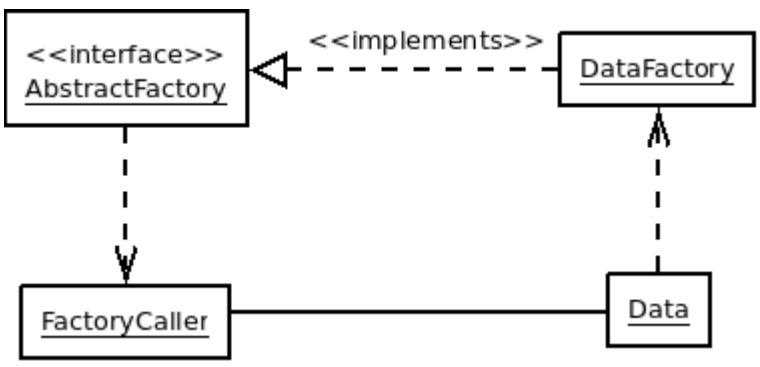
FACTORY METHOD

Il pattern Simple Factory risolve molti problemi però ha ancora un diffetto: è comunque limitato ad un “solo metodo”. Se magari volessimo più metodi che creano il nostro oggetto in mdoi differenti ? Il Factory Method è la nostra soluzione. A differenza del Simple Factory però non serve creare una classe artificiosa e si applica direttamente alla classe di cui si vuole creare l'oggetto. Il costruttore deve essere unico e privato!

```
public class Data {

     private Data() { }

     public static Data loadFromFile() {
          // carica i dati da file
          Data d = new Data();
          // setta i dati
          return d;
     }

     public static Data createNew() {
          return new Data();
     }

     public static Data loadFromDB() {
          // carica i dati da database
          Data d = new Data();
          // setta i dati
          return data;
     }

}
```

ABSTRACT FACTORY

Fino ad ora abbiamo visto che i Factory possono essere fatti attraverso l'uso di una classe artificiale. Ma se invece volessimo farlo con un'interfaccia?
È possibile farlo se però si segue il seguente schema delle classi:

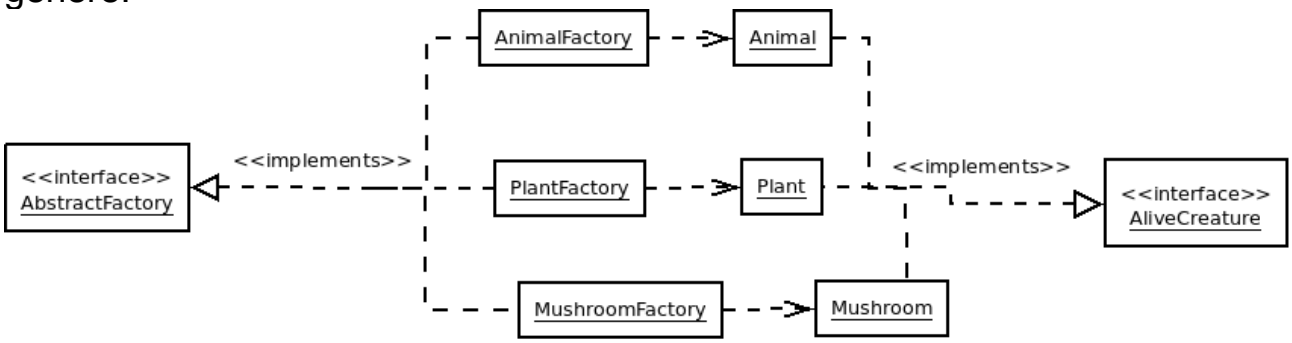


```
public interface AbstractFactory<E> {
     public E createNew();

}

public class DataFactory implements AbstractFactory<Data> {
     public Data createNew() { /* … */ }

}
```

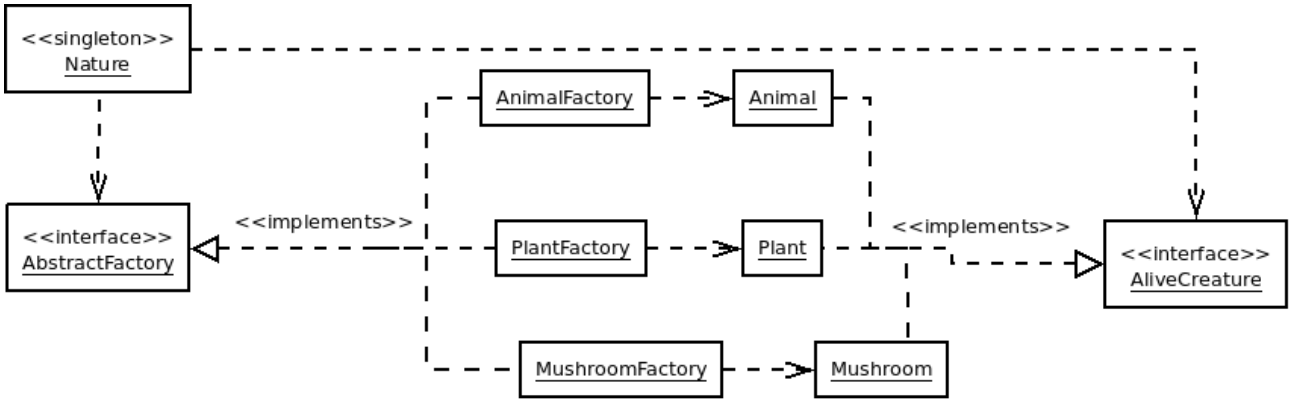
Se si vuole, è possibile implementare più metodi nell'interfaccia AbstractFactory.

PROTECT VARIATIONS + REFLECTION

Sfruttando l'AbstractFactory c'è il rischio che ciò che si vuole proteggere negli oggetti non sia più possibile perché i metodi essendo pubblici sono quindi visualizzabili da tutti quanti. Per esempio potremmo avere una situazione del genere:



In questo caso abbiamo che tutti gli elementi in gioco sono in dipendenza con una classe Singleton chiamata Nature. Tutte queste dipendenze possono portare al fatto che è più facile che sia la classe Nature a essere modificata che il modificare le altre classi in favore del mantenimento dello stato attuale di Nature. Per risolvere questo problema, e quindi anche le dipendenze usiamo il pattern Protect Variations con l'uso dei file di configurazione.



Con l'uso di un file di configurazione ridurremmo tutte le dipendeze a solo due dipendenze. Prima di vedere questi file di configurazione vediamo però i pro e i contro:

PRO: non ci sono dipendenze verso il file perché usiamo classi del pacchetto java.* che difficilmente cambieranno nel tempo e dunque non producono dipendenza.
PRO/CONTRO: tolgo delle dipendenze, ma ne aggiungo verso le classi delle librerie java.
CONTRO: devo assicurarmi che il file esista.

I file usati saranno dei file di configurazione del tipo .properties che seguono le seguenti regole di sintassi:

```
# comment #
! another comment !
Number = 10
string = hello world
testo = ora \
     vado a \
     capo
valore: proprieta'
somma = x + y \= z
divisione = x \: y \= z
altraDivisione = x \\ y \= z
```

Come si può vedere la struttura è quella di un file di configurazione dove abbiamo valore – proprietà, separati dal = o dai :. Qual'ora la stringa contenesse valori usati dal file come caratteri speciali va messo un \ per identificarli come caratteri normali.

Prendendo spunto dall'esempio precedente immaginiamo di avere un file zoo.properties che contiene le seguenti informazioni:

```
numberOfContents = 4
creature0 = panda
cod0 = ac1
area0 = exhibit1
creature1 = lion
cod1 = aft3
area1 = exhibit2
creature2 = cherryTree
cod2 = aj10
area2 = exhibit1
creature3 = amanitaMuscaria
cod3 = ei5
area3 = tropicarium
```

Purtroppo con questi dati ci risulterebbe difficile creare i vari oggetti, visto che non sappiamo a che classi appartengono.. per questo bisogna usare la Reflection, ovvero uno speciale design pattern che ci permette di creare gli oggetti di una certa classe attraverso le informazioni ottenute da essi.
Fingiamo dunque che le nostre classi siano in un package “org.zoo”.
Applicando il Reflection al nostro file zoo.properties risulterà così:

```
numberOfContents = 4
creature0 = panda
cod0 = ac1
area0 = exhibit1
class0 = org.zoo.Animal
creature1 = lion
cod1 = aft3
area1 = exhibit2
class1 = org.zoo.Animal
creature2 = cherryTree
cod2 = aj10
area2 = exhibit1
class2 = org.zoo.Plant
creature3 = amanitaMuscaria
cod3 = ei5
area3 = tropicarium
class3 = org.zoo.Mushroom
```

L'unica cosa che ci manca è vedere come si legge dal file e come si creano le classi di conseguenza:

```
try {
     FileInputStream fis = new FileInputStream(“zoo.properties”);
     Properties props = new Properties();
     props.load(fis);
     fis.close();
     String tempNum=props.getProperty(“numberOfContents”);
     int num = Integer.parseInt(tempNum);
     for (int i=0; i<num; i++) {
          String cln = props.getProperty(“class”+i);
          AliveCreature alive = Class.forName(cln).newInstance();
          alive.loadFromProperties(props,i);
     }
}
catch(Exception e) { }
```

Come si può vedere le proprietà vengono lette attraverso il metodo getProperty che restituisce una stringa che è il valore corrispondente all'etichetta passategli come parametro.
Purtroppo come tutto, anche questo design pattern ha degli svantaggi: più lento, complesso e costoso. Nessun controllo del compilatore sul codice. In compenso come vantaggio è la possibiltià di avere la possibiltià di fare modifiche senza modificare le classi principali.