DESIGN PATTERNS – Parte 1

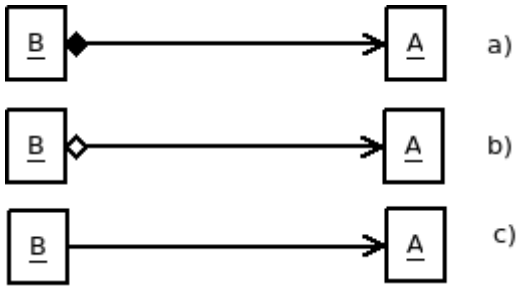
Information Expert
Creator
Null Object
Singleton

INFORMATION EXPERT

Assegno le mansioni adatte all'oggetto che le può portare a termine attraverso l'uso di informazioni dirette che l'oggetto stesso ha.
Per verificare che l'oggetto scelto abbia le informazioni necessarie ci basta suddividere il problema in tanti piccoli sotto-problemi e vedere se essi sono verificabili dall'oggetto scelto.

CREATOR

Il pattern Creator è applicabile ad un oggetto da parte di un altro oggetto se verifica uno di questi tre collegamenti:
a) B contiene A (A è accessibile solo attraverso B. A è creato solo da B)
b) B aggrega A (A è accessibile esternamente)
c) B memorizza A (B possiede un attributo di tipo A)



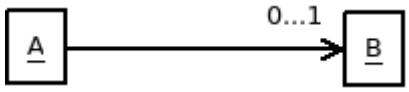
a) impica b) e b) implica c).

NULL OBJECT

Questo Pattern è usato per risolvere i problemi di molteplicità 0...1 tra due classi. Si possono risolvere con vari metodi.

Soluzione 1: utilizzo della costante NULL
Soluzione 2: classe NullObject che estende la classe con la molteplicità
Soluzione 3: interfaccia/classe generale (ha senso se i due oggetti hanno qualcosa in comune tra di loro)
Soluzione 4: classe comune Object (come scegliere NULL ma con più problemi).

Situazione iniziale:



Soluzione 1: utilizzo della constante NULL

Si può definire l'utilizzo della costante NULL in due ambiti differenti.. con un metodo get o con un metodo is.

Esempio col metodo GET

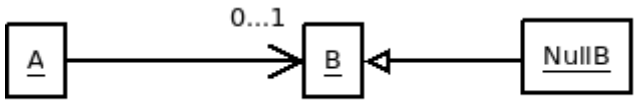
```
public class A
{
     B attribute = null;

     public B getAttribute() { return attribute; }
}
```

Esempio col metodo IS

```
public class A
{
     B attribute = null;

     public boolean isAttributeOf(B t)
     {
          if(attribute == null) return t==null;
          else return attribute.equals(t);
     }
}
```

Differenze:
Il metodo get è più facile da implementare, però causa il dover modificare il codice delle classi che utilizzeranno il metodo getAttribute della classe A.
Il metodo is è meno diretto per l'interpretazione, però non va a causare danni nelle classi che utilizzeranno il metodo isAttributeOf della classe B.

Soluzione 2: classe NullObject



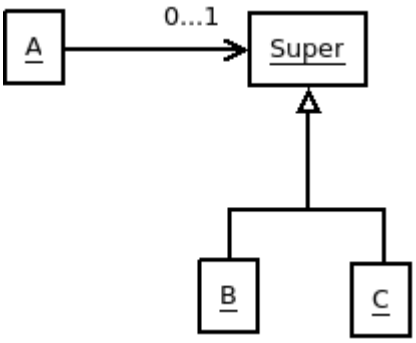
Questo pattern ci permette di creare una classe che è sottoclasse della classe che ha la molteplicità 0...1

Esempio col NullObject:

```
public class NullB extends B
{
     public static NullB istanza = new NullB();
}

public class A
{
     B argument = NullB.istanza;

     public B getArgument() { return argument; }

     public boolean returnArgument(B t)
     {
          if(t.equals(argument)) argument = NullB.istanza;

          return argument.equals(NullB.istanza);
     }
}
```

Soluzione 3: interfaccia / classe generale

È possibile ritorna la classe stessa come rappresentazione del valore nullo. Così però bisogna prima avere una superclasse che rappresenti i due elementi.



Esempio con superclasse:

```
public interface Super {
     static Super getIstanza();
}

public class B implements Super
{ /* … */ }

public class C implements Super
{ /* … */ }
```

```
public class A
{
     Super argument = C.getIstanza();

     public Super getArgument() { return argument; }

     public boolean returnArgument(B t)
     {
          if(t.equals(argument)) argument = C.getIstanza();
          return argument.equals(C.getIstanza());
     }
}
```

SINGLETON

È possibile risolvere il problema dell'unica istanza in due modi... coi metodi static o con il pattern Singleton.

Esempio di classe static:

```
public class StaticClass
{
     private static Type arg1, arg2, arg3;

     public static Type getArg1() { return arg1; }
     public static Type getArg2() { return arg2; }
     public static Type getArg3() { return arg3; }
     public static void setArgs(Type a1, Type a2, Type a3)
     {    arg1 = a1;     arg2 = a2;     arg3 = a3;     }
}
```

Questo tipo di implementazione però ha molti diffetti, per esempio che non segue le regole di OOP. Per questo ci viene in soccorso il pattern Singleton che ci permette di risolvere il problema in una maniera migliore e soprattutto orientata agli oggetti.

Per applicare il Singleton bisogna seguire le seguenti regole:

- Costruttore privato
- Riferimento all'oggetto privato
- Unica istanza costituita dalla classe
  1. Una sola classe può farlo
  2. La classe è l'unica istanza
  3. O lo fa sempre e prima di tutto o solo quando serve
- Metodo getter static e public
- Vari tipi di inizializzazione (preventiva, lazy, etc..)

Esempio di inizializzazione preventiva:

```
public class Singleton
{
     public static final Singleton ref = new Singleton();

     public static Singleton getInstance() { return ref; }

     private Singleton() { /* … */ }
}
```

In alcuni casi però si preferirebbe generare l'istanza solo durante la chiamata del metodo getter e non alla generazione della classe stessa. Per questo si utilizza un implementazione differente detta Lazy.

Esempio di inizializzazione lazy:

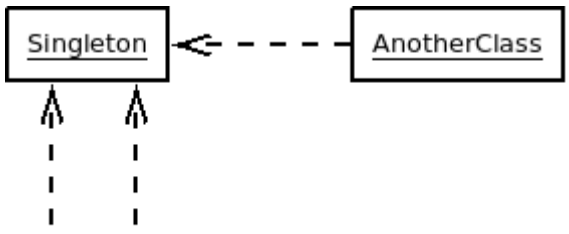
```
public class LazySingleton
{
     public static LazySingleton ref = null;

     public static LazySingleton getInstance()
     {
          if(ref == null) ref = new LazySingleton();

          return ref;
     }

     private LazySingleton() { /* … */ }
}
```

Ora però vediamo come utilizzare il Singleton in una classe esterna:

Esempio:

```
public class AnotherClass
{
      public void method()
      {
           if(Singleton.getInstance().someMethod())
           {
                this.doSomething();
           }
      }

      public void doSomething() { /* …. */ }
}
```

Il Singleton è quindi più flessibile, più facile da passare da 1 a n istanze del metodo Static. La situazione codificata in UML sarà quindi questa...

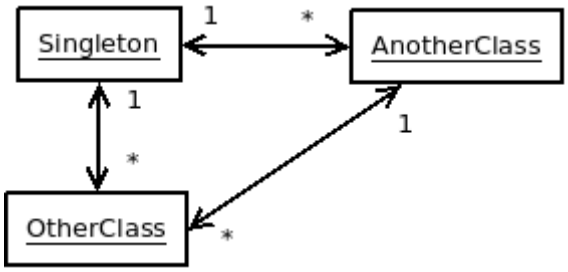


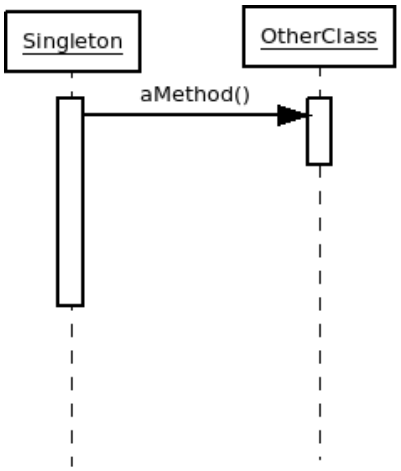
Come si può vedere ci sono troppe dipendenze dovute al fatto che è visibile in tutta l'applicazione. Si può risolvere il problema in due modi:

1) *Metodo 1*: passare un riferimento dell'oggetto a tutti gli oggetti, ovvero quindi aggiungere associazioni
2) *Metodo 2*: sfrutto le associazioni già esistenti
3) *Metodo 3*: attraverso le leggi di Demetra

Metodo 1: aggiungo associazioni

Mettiamo di avere assieme alle due classi precedentemente nominate una terza classe che chiameremo OtherClass.
Il metodo1 consiste nel risolvere il problema passando il Singleton ad ogni classe quando questa o questo viene creata. In questo modo genero delle associazioni.





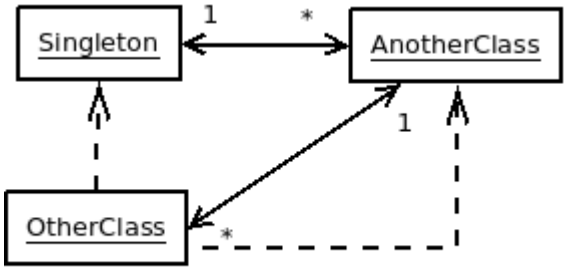
Esempio:

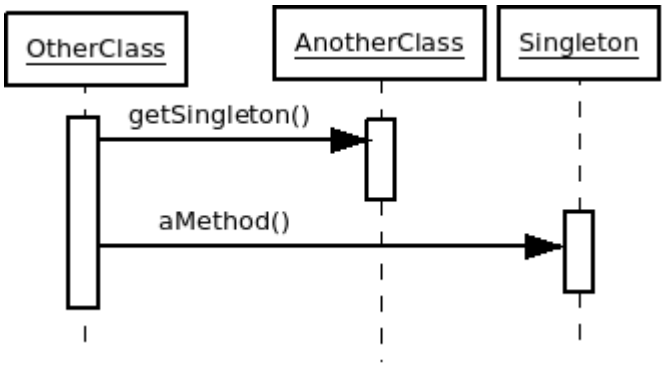
```
public class Singleton
{
     public AnotherClass createAnother()
     {    return new AnotherClass(this);     }

     public void aMethod() { /* … */ }
}

public class AnotherClass
{
     private Singleton single;
     public AnotherClass(Singleton s)
     {
          single = s;
     }

     public void createOther()
     {    return new OtherClass(single, this);    }

     // …
}

public class OtherClass
{
     private Singleton otherSingle;
     private AnotherClass another;
     public OtherClass(Singleton s, AnotherClass a)
     {
          otherSingle = s;
          another = a;
     }
     public void example() {  otherSingle.aMethod(); }
}
```

In questo modo Singleton non è più visibile globalmente ed è accessibile solo in maniera indiretta.

Metodo 2: associazoni già esistenti

Aggiungo in AnotherClass un metodo che restituisca l'attributo Singleton.





Esempio:

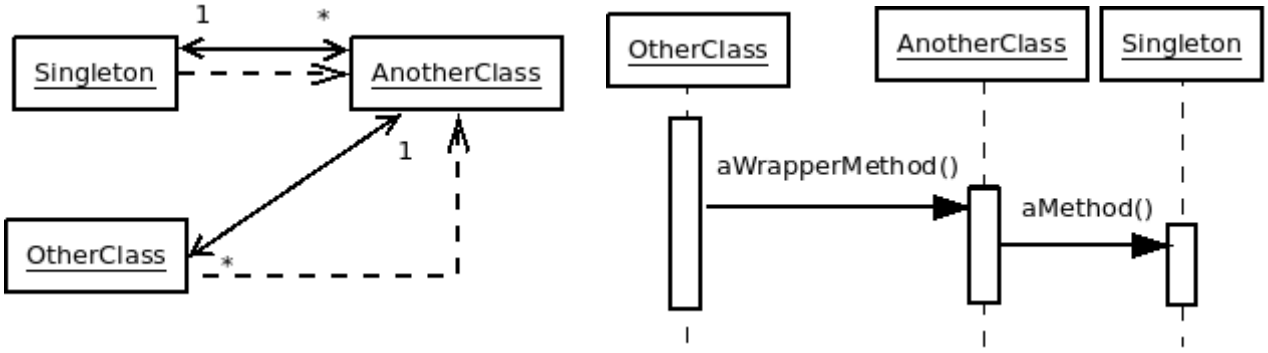
```
public class Singleton
{
     public AnotherClass createAnother()
     {    return new AnotherClass(this);     }

     public void aMethod() { /* … */ }
}

public class AnotherClass
{
     private Singleton single;
     public AnotherClass(Singleton s)
     {
          single = s;
     }

     public void createOther()
     {
          return new OtherClass(this);
     }

     public Singleton getSingleton() { return single; }

     //...
}

public class OtherClass
{
     private AnotherClass another;
     public OtherClass(AnotherClass a){  another = a; }
     public void example() {  another.getSingleton().aMethod(); }
}
```

Questo però rende example un metodo fragine, ovvero un metodo che è da rifare da capo in caso di una qualunque minima modifica.
Per non avere metodi fragili bisognerebbe seguire le Leggi di Demetra, ovvero un metodo è accessibile solo:

– dalle variabili di istanza
– dai parametri
– dagli oggetti che crea
– dei suoi componenti diretti

Metodo 3: con le leggi di Demetra

Le leggi di Demetra se applicate producono molte metodi wrapper, ovvero metodi che richiamano altri metodi.



Esempio:

```
public class Singleton
{
     public AnotherClass createAnother()
     {    return new AnotherClass(this);     }
     public void aMethod() { /* … */ }
}

public class AnotherClass
{
     private Singleton single;
     public AnotherClass(Singleton s){ single = s; }
     public void createOther(){ return new OtherClass(this); }
     public void aWrapperMethod() { single.aMethod(); }

     //...
}

public class OtherClass
{
     private AnotherClass another;
     public OtherClass(AnotherClass a){  another = a; }
     public void example() {  another.aWrapperMethod(); }
}
```