Il pattern Prototype

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.
“Patterns in Java”, Grand

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Prototype

Classificazione

- Scopo: creazionale
- Raggio d'azione: basato su oggetti

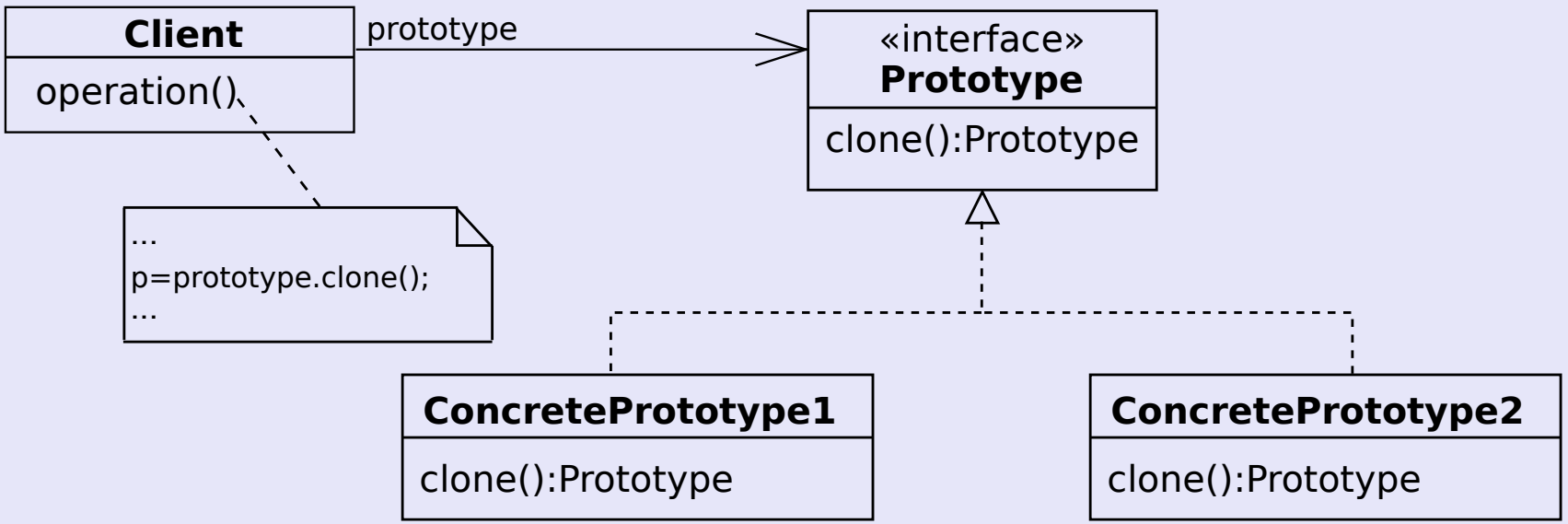
Scopo

- Specificare la tipologia di oggetti da creare utilizzando un'istanza prototipo e creare nuovi oggetti copiando questo prototipo.

Motivazione

- Si pensi ad applicazione che consenta rappresentare degli elementi grafici nel piano cartesiano.
- L'applicazione potrebbe avere una barra di tasti per effettuare varie operazioni.
- Un'azione tipica è quella di creare un nuovo oggetto grafico.
- Per inserire differenti oggetti l'azione da compiere è identica a parte il tipo di oggetto da creare.
- Una soluzione consiste nel configurare l'oggetto responsabile della creazione (il tasto) con un prototipo dell'oggetto da creare.

Struttura



Partecipanti

- **Prototype**: specifica l'interfaccia che consente la clonazione.
- **ConcretePrototype**: implementa l'operazione di clonazione.
- **Client**: crea un nuovo oggetto chiedendo ad un prototipo di clonarsi.

Conseguenze

Ha molte delle conseguenze possedute da **Abstract Factory** e **Builder**: nasconde ai client le classi dei prodotti concreti, riducendo il numero di nomi di classi che devono essere noti al client stesso.

- ☺ *Consente di aggiungere e rimuovere prodotti durante l'esecuzione.*
  I prototipi consentono di aggiungere un nuovo prodotto concreto in un sistema semplicemente registrando una sua nuova istanza prototipale presso il client. I prototipi possono essere installati e rimossi durante l'esecuzione.
- ☺ *Specifica di nuovi oggetti variando i valori.*
  Spesso nuovi comportamenti possono essere definiti componendo degli oggetti in modo diverso senza necessità di introdurre nuove classi. Nuovi "tipi" di prodotto sono definiti registrando come prototipi istanze differenti della stessa classe.
- ☺ *Specifica di nuovi oggetti variando la struttura.*
  Molte applicazioni costruiscono oggetti partendo da parti e sottoparti. Un programma per la progettazione di circuiti, per esempio, può costruire nuovi circuiti partendo da parti di circuito.Tali applicazioni spesso consentono all'utente di istanziare strutture complesse e definite dall'utente, in modo da poter riusare più volte una parte di circuito precedentemente definita.

Conseguenze

☺ *Riduzione del numero di sottoclassi*
Il pattern *Prototype* permette di risolvere un problema di *Factory method* relativo alla dimensione della gerarchia di classi necessarie. Usando un metodo factory è necessario creare sottoclassi per inserire un nuovo prodotto e, se si hanno numerosi prodotti molto simili tra di loro, la definizione di una nuova classe per ognuno può portare a grandi quantità di codice duplicato. Usando i prototipi non sono necessarie né una classe factory, né la gerarchia di classi associata ai prodotti: i nuovi oggetti vengono istanziati e inizializzati variando valori interni e struttura.

☹ *Implementazione dell'operazione clone.*
Il metodo `clone` deve comportarsi come una copia in profondità (*deep copy*), in modo che la copia di un oggetto composto implichi la copia delle sue sottoparti. Poiché molti linguaggi di programmazione utilizzando una copia semplice (*shallow copy*), l'operazione `clone` deve essere ridefinita dal programmatore e ciò può risultare particolarmente complesso in presenza di strutture dati con riferimenti circolari o nel caso alcuni oggetti non permettano la copia.

Prototyope in Java: l'interfaccia `Cloneable`

- `Cloneable` è l'interfaccia (marker) di Java che marca una classe come clonabile.
- L'effetto è che il metodo protected `clone()` di `Object` ritorna una copia campo-per-campo dell'oggetto.
- Se la classe concreta non implementa `Cloneable`, il metodo `clone()` solleva l'eccezione `CloneNotSupportedException`.
- La sottoclasse può rendere il metodo `public` e cambiare il tipo di ritorno (*covarianza*).
- L'oggetto restituito **DEVE** essere quello ottenuto invocando `super.clone()`!

Errata ridefinizione di `clone()`: La classe non implementa `Cloneable`!

```
public class MyWrongClonable {
  @Override public MyWrongClonable clone() {
    try { return (MyWrongClonable) super.clone(); }
    catch (CloneNotSupportedException e) { throw new Error(e);}
  }
}
```

```
MyWrongClonable w = new MyWrongClonable();
MyWrongClonable clone=w.clone(); //si verifica un errore: la classe Object solleva un'eccezione
```

Prototype in Java: l'interfaccia `Cloneable`

- il metodo `clone()` di `Object` clona oggetti in modo "superficiale" (*shallow copy*), ossia i campi dei tipi di base sono clonati perfettamente ma i campi oggetti sono "clonati" copiando solo i riferimenti (*aliasing*).
- Per ottenere una "copia profonda" (*deep copy*), una classe clonabile ridefinisce il metodo `clone()` in modo da copiare in modo profondo anche i campi oggetto
- Una ridefinizione di `clone()` deve, come prima cosa, invocare `super.clone()` (che può sollevare l'eccezione controllata `CloneNotSupportedException`), quindi provvedere a perfezionare l'oggetto restituito da `super.clone()` (previo casting) con la copia dei campi oggetto di interesse.

Shallow Copy

```
public class Point2D implements Cloneable {
  private double x; private double y;
   ...
  @Override public Point2D clone() {
    try { Point2D clone = (Point2D) super.clone();
          return clone; }
    catch (CloneNotSupportedException e) { throw new Error(e);}
  }
}
```

Best practice: Effective Java Item 11

- Una classe non final può ridefinire il metodo `clone()` (protected) di `Object` e comportarsi come la classe Object nei confronti di una sotto classe.
- **Solamente se** la classe è `final` essa *potrebbe* ridefinire `clone()` restituendo un oggetto creato mediante un costruttore senza invocare `super.clone()`.
- Anche una super classe, ridefinendo `clone()`, deve come prima cosa invocare `super.clone()`.
- Ne deriva che una gerarchia di classi ben definita dal punto di vista della clonazione, comporta che ad una richiesta di clone, venga ultimamente invocato il metodo `clone()` di `Object` che realizza la shallow copy.
- Questo è l'**unico** modo per garantire che `clone()` restituisca un oggetto del tipo che ci si attende.

Esempio: framework polinomi

- Si consideri nuovamente il progetto Polinomio Per “creare al volo”, nella classe `PolinomioAstratto`, un polinomio concreto della classe appropriata, anziché utilizzare il Factory Method ci si può avvalere del Prototype.
- Si fa in modo che la classe `PolinomioAstratto` implementi `Cloneable` oltre all’interfaccia `Polinomio`. Tutto ciò assicura che i polinomi eredi di `PolinomioAstratto` saranno clonabili.
- In una classe polinomio concreto si ridefinisce il metodo `clone()` effettuando eventuali “copie profonde”.
- Si *registra* come prototipo un polinomio presso la classe `PolinomioAstratto` un’istanza di una sotto-classe concreta .

Esempio: framework polinomi

```
public abstract class PolinomioAstratto implements Polinomio, Cloneable {
 ...

  protected abstract PolinomioAstratto getPrototype();

  @Override public Polinomio add(Polinomio p) {
     Polinomio somma = getPrototype().clone(); // crea un nuovo polinomio
     for  (Monomio m : this) somma.add(m); // aggiunge ciascun monomio di this al polinomio somma
     for  (Monomio m : p) somma.add(m); // aggiunge ciascun monomio di p al polinomio somma
       return somma;
  }

  @Override public PolinomioAstratto clone() {
     try  { return (PolinomioAstratto)  super.clone(); }
     catch (CloneNotSupportedException e) {
       throw new Error(e);
     }
  }

   ...
}
```

Esempio: framework polinomi

- Per polimorfismo, `PolinomioAstratto` utilizzerà il metodo `clone()` del tipo dinamico di `this` (che potrebbe, ma non necessariamente, essere lo stesso di quello del parametro polinomio).

```
public class PolinomioLL extends PolinomioAstratto {
  private static  PolinomioLL prototype;

  private LinkedList<Monomio> monomi = new LinkedList<>();
   ...
  @Override protected synchronized PolinomioLL getPrototype(){
       if ( prototype==null ) prototype = new PolinomioLL();
      return prototype;
  }

  @Override public PolinomioLL clone() {
     PolinomioLL p = (PolinomioLL) super.clone();
     p.monomi = new LinkedList<Monomio>();
     for  (Monomio m : this)
       p.monomi.add(m);
     return p;
  }
   ...
}
```

Esempio: clonazione di uno stack

```
public class Stack implements Cloneable {
  private Object[] elements; private int size = 0;
  private static final int DEFAULT_INITIAL_CAPACITY = 16;

  public Stack() { this.elements = new Object[DEFAULT_INITIAL_CAPACITY]; }

  public void push(Object e) { ensureCapacity(); elements[size++] = e; }// push

  public Object pop() { if (size == 0) throw new EmptyStackException();
    Object result = elements[--size]; elements[size] = null; // Elimina riferimento obsoleto
    return result;
  }// pop

  // Ensure space for at least one more element.
  private void ensureCapacity() { if (elements.length == size) elements = Arrays.copyOf(elements, 2 * size + 1);}

  @Override public Stack clone() {
      try { Stack result = (Stack) super.clone();
            result.elements = elements.clone(); // clone ricorsiva dell'array
            return result;
      } catch (CloneNotSupportedException e) { throw new Error(e);}
  }// clone
}// Stack
```

In questo caso l'invocazione "ricorsiva" di `clone()` sull'array è sufficiente per effettuare una copia profonda dello stack. Attenzione: Gli elementi contenuti non sono clonati!