Marco Faella

I pattern Template Method e Factory Method

Lezione n. 12

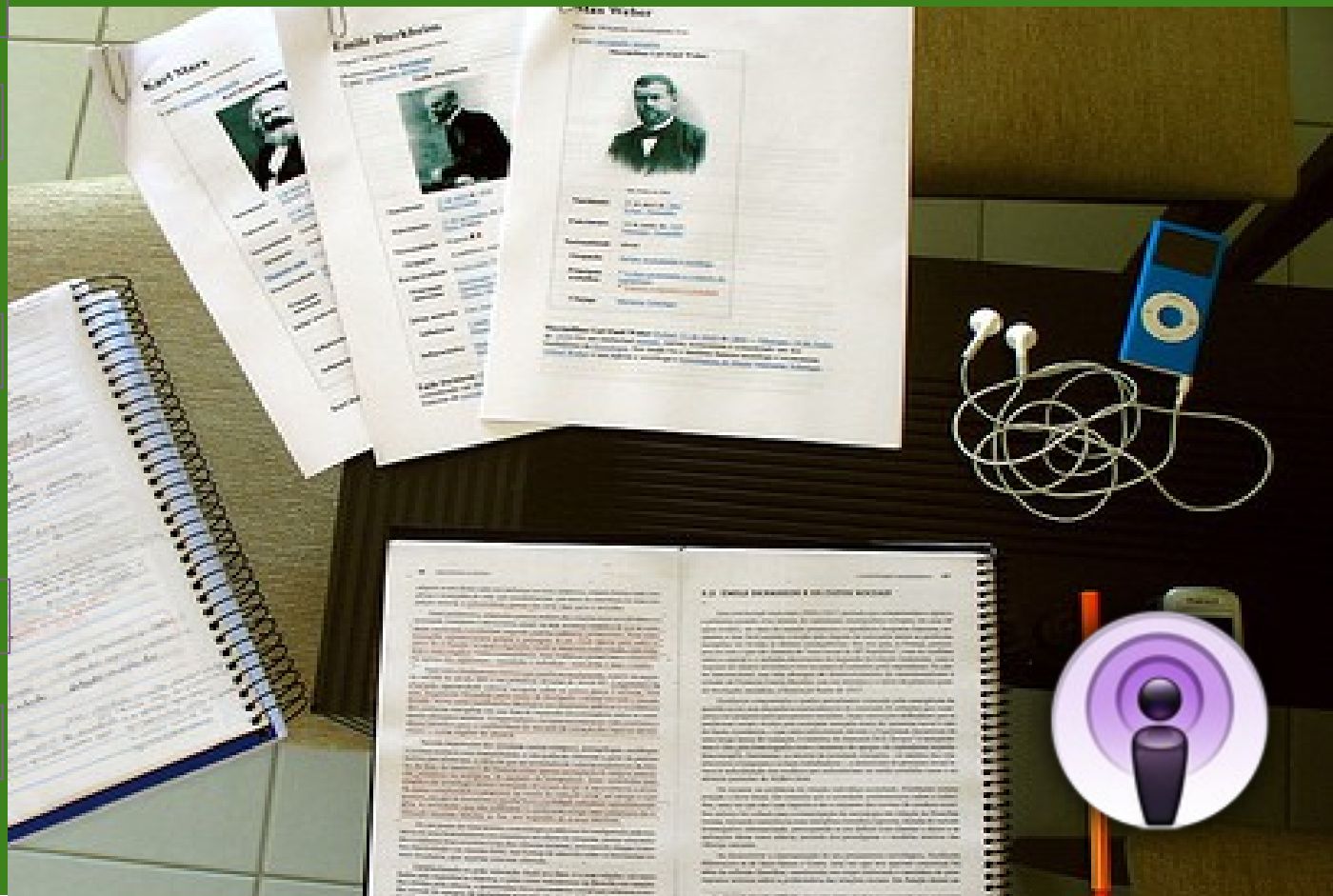
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



Esercizio: corpi celesti

- Cominciamo questa lezione con un esercizio, che ci condurrà ad un nuovo design pattern

Per un programma di astronomia, si implementino le tre classi **Planet**, **Moon** e **Star**, che rappresentano, rispettivamente, un pianeta, una luna (ovvero, un satellite naturale) ed una stella.

A ciascuno di questi corpi celesti è associato un **nome** ed una **massa**.

Quando si crea un oggetto pianeta o luna, bisogna anche specificare il corpo intorno al quale questo orbita.

Le classi devono verificare che i pianeti orbitino intorno a una stella, e le lune intorno a dei pianeti.

Il tentativo di violare queste regole porta ad un'eccezione.

Si fornisca anche una classe (o interfaccia) **Body** che rappresenta un generico corpo celeste, ed è quindi una superclasse di tutte le precedenti.

La classe Body dispone di un metodo *satelliteIterator*, che restituisce un iteratore sull'insieme di satelliti di questo corpo.

La slide seguente mostra un esempio d'uso di queste classi.

Esercizio: corpi celesti

- Le classi descritte nella slide precedente devono poter essere utilizzate nel modo seguente
  - i dati astronomici sono tratti da www.wikipedia.org

```
Body b[] = new Body[5];
b[0] = new Star("Sole", 1.98e30);
// Marte orbita intorno al sole
b[1] = new Planet("Marte", 6.14e23, b[0]);
// Phobos e Deimos orbitano intorno a Marte
b[2] = new Moon("Phobos", 1.07e16, b[1]);
b[3] = new Moon("Deimos", 2.24e15, b[1]);

java.util.Iterator i = b[1].satelliteIterator();

System.out.println(" Lune di Marte: ");
while (i.hasNext()) {
   Moon m = (Moon) i.next();
   System.out.println(m.getName() + " : " + m.getMass());
}

System.out.println(" ed ora...");
b[4] = new Moon("Giove", 1, b[0]);
```

Esercizio: corpi celesti

- Il codice della slide precedente dovrebbe avere un output simile al seguente

```
Lune di Marte:
Phobos : 1.07E16
Deimos : 2.24E15
 ed ora...
Exception in thread "main" planets.Body$InvalidBodyTypeException
    at planets.Body.addSatellite(Body.java:32)
    at planets.Body.<init>(Body.java:17)
    at planets.Moon.<init>(Moon.java:6)
    at PianetiTest.main(PianetiTest.java:24)
```

- Ovvero, l'iteratore restituito dal metodo *satelliteIterator* permette di ottenere, una alla volta, le due lune di Marte
- L'ultima istruzione, che tenta di definire Giove come una luna del sole, fallisce lanciando un'eccezione, perché una **luna** non può orbitare direttamente intorno ad una **stella**

Esercizio: soluzione di massima

- Abbozziamo una soluzione dell'esercizio, con particolare riferimento al controllo necessario ad assicurare che lune e pianeti orbitino intorno a corpi del tipo giusto
- Cercheremo di seguire il principio secondo il quale bisogna raccogliere nelle superclassi (in questo caso, Body) il maggior numero possibile di funzionalità comuni

- Decidiamo quindi di introdurre in Body un metodo di supporto, chiamato *isValidSatellite*, che prende un corpo celeste x come argomento, e restituisce vero se e solo se questo corpo potrebbe orbitare intorno ad x
- Body sarà una classe **astratta** perché non ha senso in questo contesto istanziare un corpo celeste generico
- Questo ci dà l'occasione di lasciare il metodo *isValidSatellite* astratto, in modo che ciascuna sottoclasse lo ridefinisca nel modo appropriato

- Le prossime slide presentano una possibile implementazione della classe Body

Esercizio: la classe Body, prima parte

```
public abstract class Body {
     private double mass;
     private String name;
     private Body parent;
     private List satellites;

     public Body(String name, double mass, Body parent) {
          this.name = name;
          this.mass = mass;
          this.parent = parent;
          satellites = new LinkedList();
          if (parent != null) parent.addSatellite(this);
     }

     /** Controlla che un corpo sia del tipo giusto per essere un satellite di this. */
     protected abstract boolean isValidSatellite(Body other);

     /** Lanciata quando si tenta di far orbitare un corpo intorno
         ad un altro corpo del tipo sbagliato. */
     public static class InvalidBodyTypeException extends RuntimeException { };
```

Esercizio: la classe Body, seconda parte

```
    /** Restituisce un iteratore sui corpi che orbitano intorno a questo corpo. */
    public Iterator satelliteIterator() {
         return satellites.iterator();
    }

    /** Aggiunge un satellite a questo corpo. Chiamato solo dalle sottoclassi. */
    protected void addSatellite(Body other) {
         if ( isValidSatellite(other) )
              satellites.add(other);
         else
              throw new InvalidBodyTypeException();
    }
}
```

Esercizio: la classe Planet

- Riportiamo una possibile implementazione della classe **Planet**
- Grazie ad una accurata pianificazione delle responsabilità, siamo riusciti a ridurre al minimo il codice contenuto nelle sottoclassi come Planet
- L'annotazione "@Override" indica che il metodo che segue intende eseguire l'overriding di uno della superclasse
  - le annotazioni verranno illustrate in una lezione successiva

```
public class Planet extends Body {

    public Planet(String name, double mass, Body parent) {
        super(name, mass, parent);
    }

    @Override
    protected boolean isValidSatellite(Body other) {
        // Satellites of planets must be moons.
        return other instanceof Moon;
    }
}
```

Contesto:

1) Un **algoritmo** è applicabile a *più tipi di dati*
2) L'algoritmo può essere scomposto in **operazioni primitive**. Le operazioni primitive possono essere diverse per ciascun tipo di dato
3) L'ordine di applicazione delle operazioni primitive non dipende dal tipo di dato

Soluzione:

4) Definire una superclasse che abbia un metodo che realizza l'algoritmo e un metodo per ogni operazione primitiva
5) Le operazioni primitive non sono implementate nella superclasse (metodi astratti), oppure sono implementate in modo generico
6) Ridefinire in ogni sottoclasse i metodi che rappresentano operazioni primitive, ma non il metodo che rappresenta l'algoritmo

Diagramma del pattern TEMPLATE METHOD

- Il diagramma a destra illustra graficamente le classi coinvolte nel pattern TEMPLATE METHOD
- Vi si trova la superclasse, con metodi primitivi e metodo composito "algorithm", e una sottoclasse, che ridefinisce appropriatamente i metodi primitivi

- Con riferimento all'esercizio dei corpi celesti:
  - la classe Body rappresenta la superclasse
  - il metodo addSatellite rappresenta algorithm
  - il metodo isValidSatellite è il metodo primitivo
  - le classi Planet, Moon e Star sono le sottoclassi concrete di Body

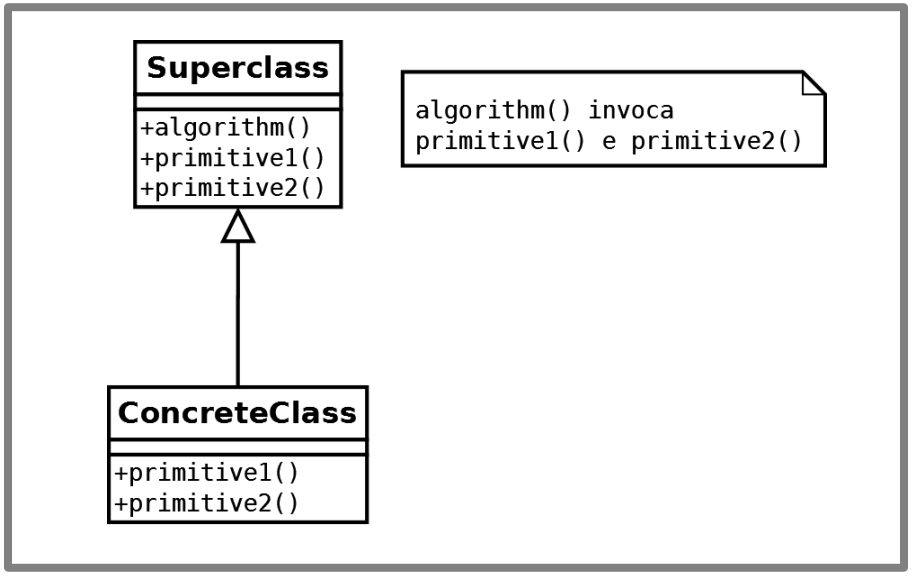


Figura 1: Diagramma UML tipico del pattern TEMPLATE METHOD.

Contesto:

1) Un tipo (*creatore*) **crea** oggetti di un altro tipo (*prodotto*)
2) Le sottoclassi del tipo creatore devono creare prodotti di **tipi diversi**
3) I clienti non hanno bisogno di sapere il tipo esatto dei prodotti

Soluzione:

1) Definire un tipo per un creatore generico
2) Definire un tipo per un prodotto generico
3) Nel tipo creatore generico, definire un metodo (detto appunto *metodo fabbrica*) che restituisce un prodotto generico
4) Ogni sottoclasse concreta del tipo creatore generico realizza il metodo fabbrica in modo che restituisca un prodotto concreto

Diagramma del pattern FACTORY METHOD

- Il diagramma a destra illustra i rapporti tra le classi coinvolte nel pattern FACTORY METHOD

- Le due interfacce **Creator** e **Product** rappresentano il creatore generico e il prodotto generico, rispettivamente
- La relazione di **dipendenza** tra Creator e Product è dovuta semplicemente al fatto che il creatore ha un metodo (il metodo fabbrica, per l'appunto) che restituisce un oggetto di tipo Product
- Al di sotto, troviamo un creatore concreto e un prodotto concreto
- Naturalmente, la relazione di dipendenza si estende anche a loro

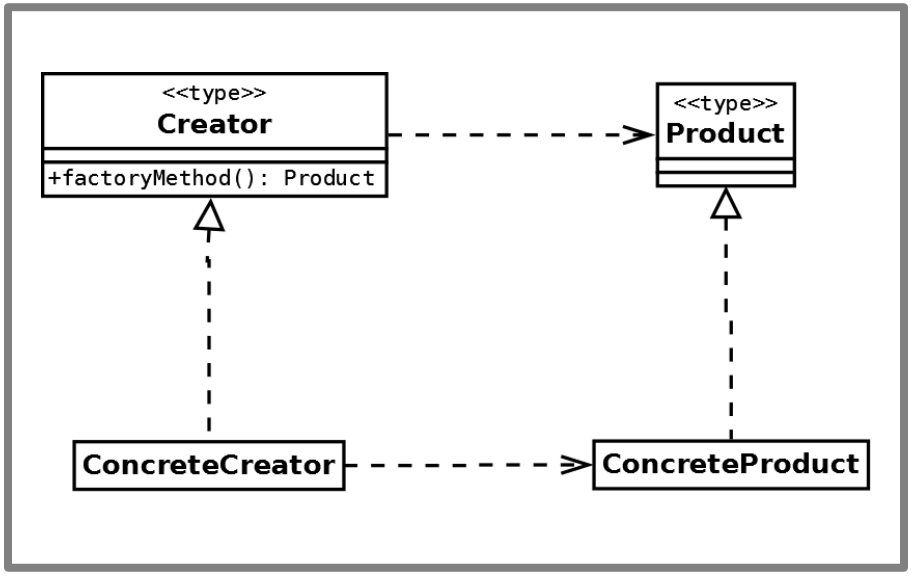


Figura 2: Diagramma UML tipico del pattern FACTORY METHOD.

Applicazione del pattern FACTORY METHOD

- Il diagramma a destra illustra come il pattern FACTORY METHOD si ritrova applicato nella creazione di iteratori da parte delle collezioni come **LinkedList**

- Come si vede, **Iterable** rappresenta il creatore generico e **Iterator** il prodotto generico
- Un produttore generico è rappresentato dalla classe LinkedList, che implementa Iterable
- La classe LinkedList costruisce internamente un iteratore concreto, di una classe che non è documentata nella libreria standard, e che nel diagramma viene chiamata ConcreteIterator
    - è probabile che questa sia una classe interna privata di LinkedList

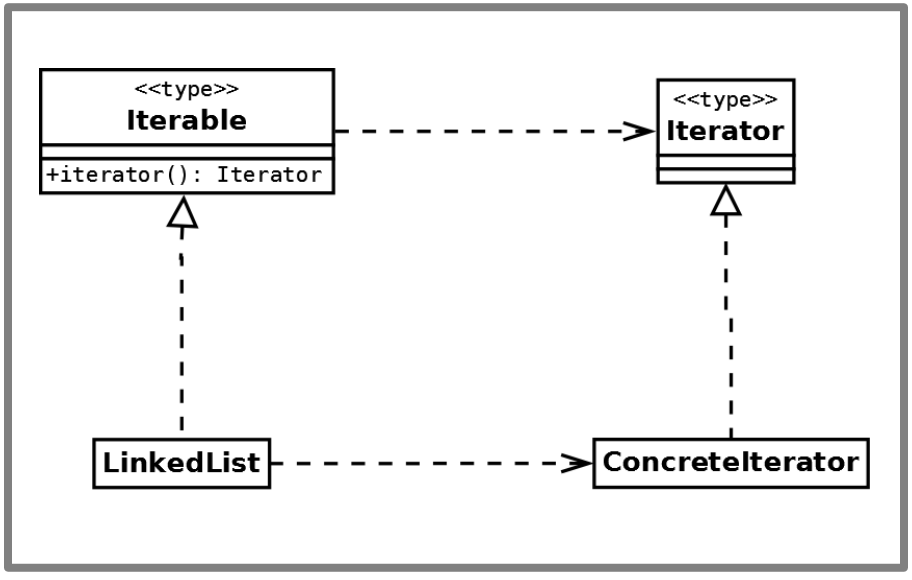


Figura 3: Diagramma dell'applicazione del pattern FACTORY METHOD alla creazione di iteratori da parte di collezioni della libreria standard.

Esempio negativo di FACTORY METHOD

- E' utile considerare esempi di progettazione che assomigliano al pattern FACTORY METHOD, ma che in realtà non gli corrispondono

- Ad esempio, consideriamo il metodo **toString** della classe **Object**
  - Non c'è un'interfaccia che rappresenti il creatore generico, però la classe Object potrebbe farne le veci
  - Allo stesso modo, la classe String potrebbe fungere da prodotto generico
  - Tuttavia, la classe String è final, cioè non è estendibile
  - Quindi, manca uno dei presupposti principali del pattern, ovvero: "Le sottoclassi del tipo creatore devono creare prodotti di tipi diversi"

- **Esercizio**: esaminare la documentazione del metodo *createEtchedBorder* della classe *javax.swing.BorderFactory* e stabilire se si tratta di un esempio del pattern FACTORY METHOD