Marco Faella

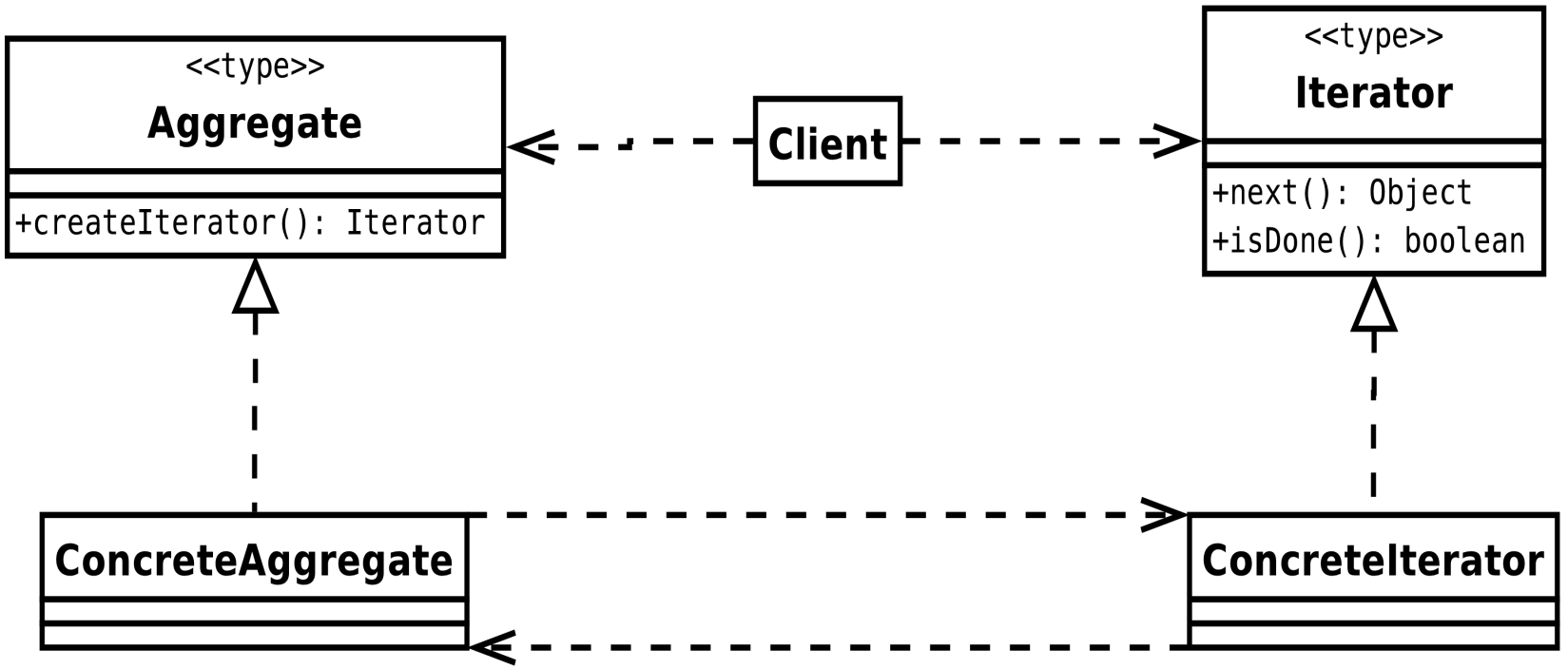
Alcuni Design Pattern in Java

basato su “Progettazione del Software e Design Pattern in Java”,
di Cay Horstmann

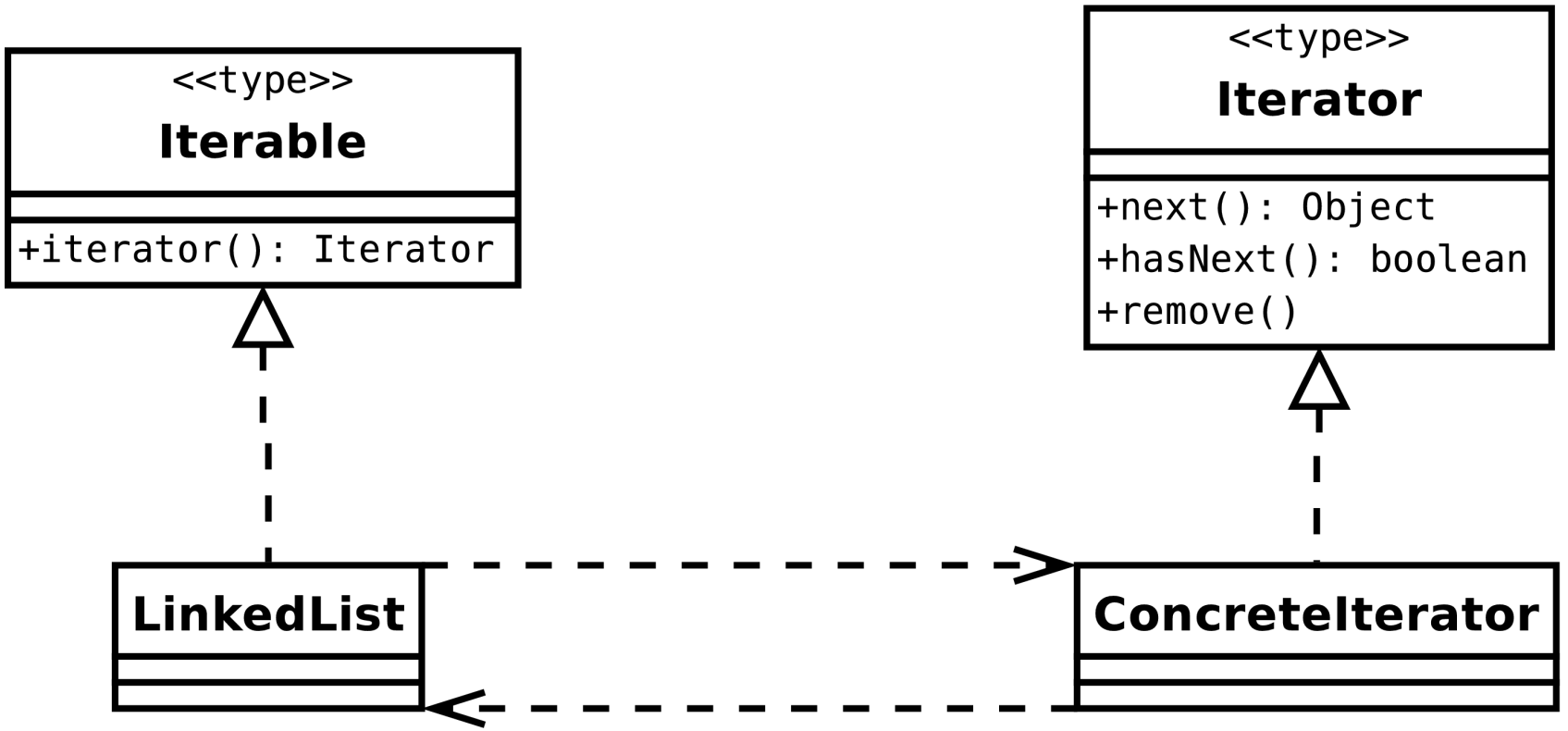
Pattern ITERATOR

- Contesto:
  1) Un oggetto (*aggregato*) contiene altri oggetti (*elementi*)
  2) I clienti devono accedere a tutti gli elementi
  3) L'aggregato non deve esporre la sua struttura interna
  4) Più clienti devono poter accedere contemporaneamente
- Soluzione:
  1) Definire una classe *iteratore* che recupera un elemento per volta
  2) L'aggregato ha un metodo che restituisce un iteratore

Diagramma del pattern ITERATOR



Iteratori in Java



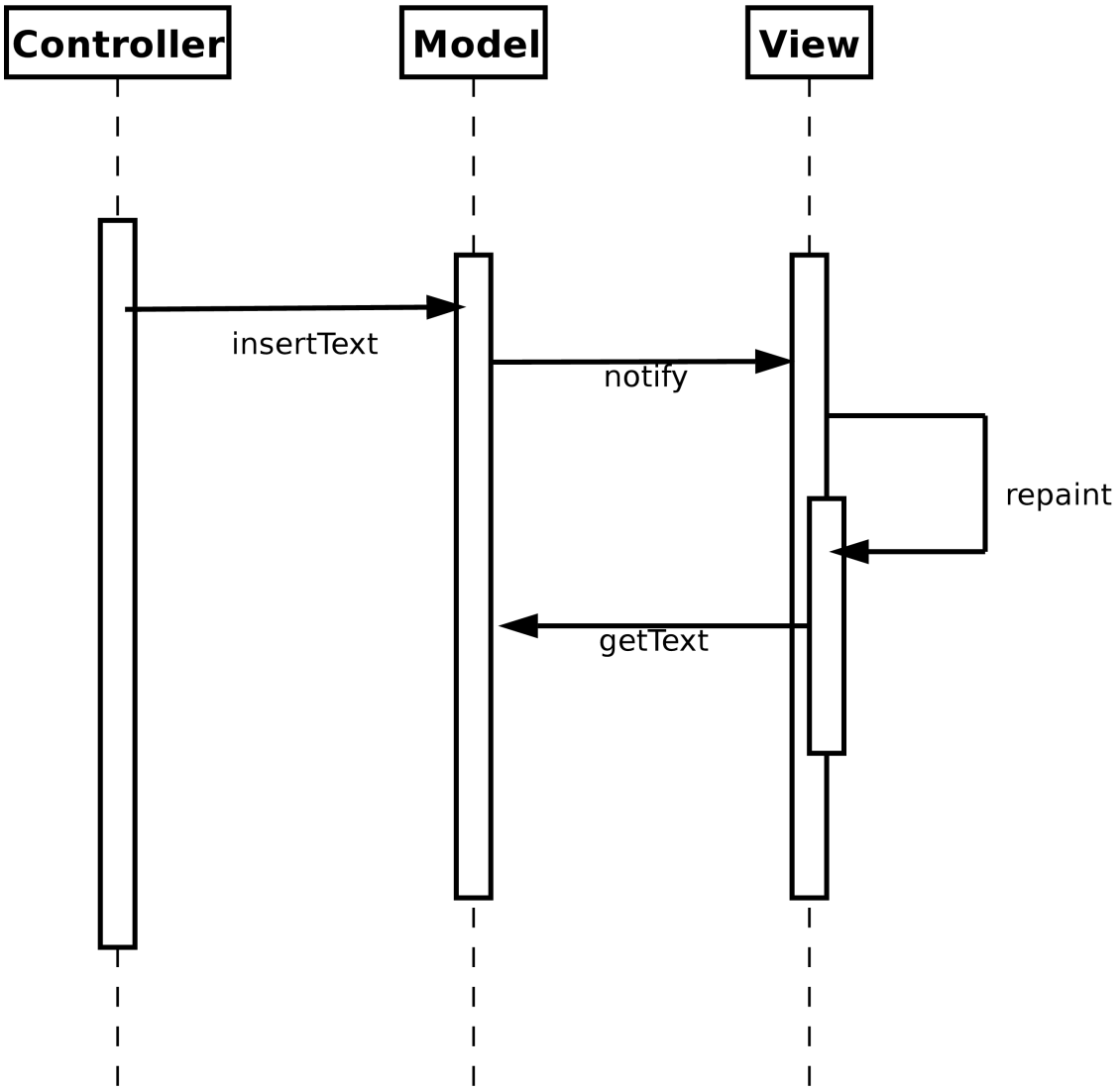
Paradigma Model-View-Controller (MVC)

- Dividere le responsabilità tra:
  - i dati di interesse (model)
  - la loro presentazione (view)
  - l'interazione con l'utente (controller)

Esempio: editor di pagine HTML

- **Model**: i caratteri che compongono la pagina
  - classe HtmlPage
- **View**: una vista carattere per carattere, o una vista *WYSIWYG* (what you see is what you get)
  - classi HtmlRawView, HtmlRenderedView
- **Controller**: l'observer che riceve gli eventi dall'utente e produce modifiche sul modello
  - classe InsertListener

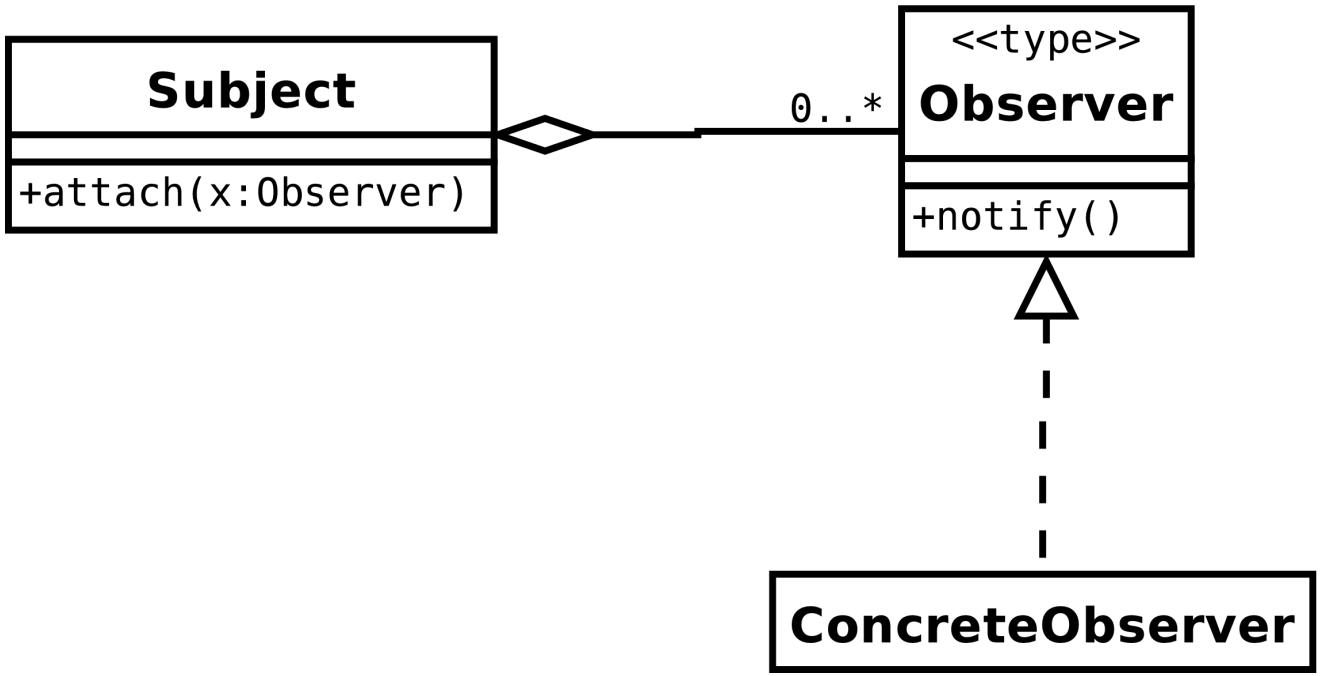
Esempio: pagine HTML



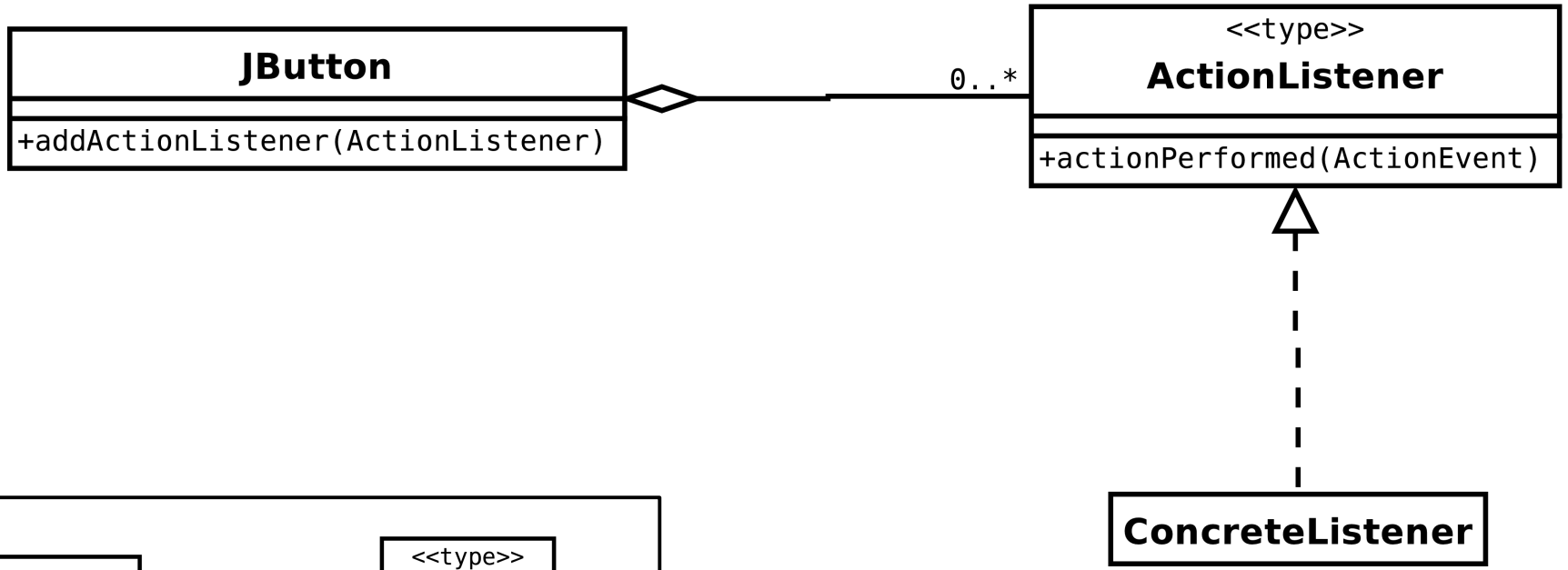
Pattern OBSERVER

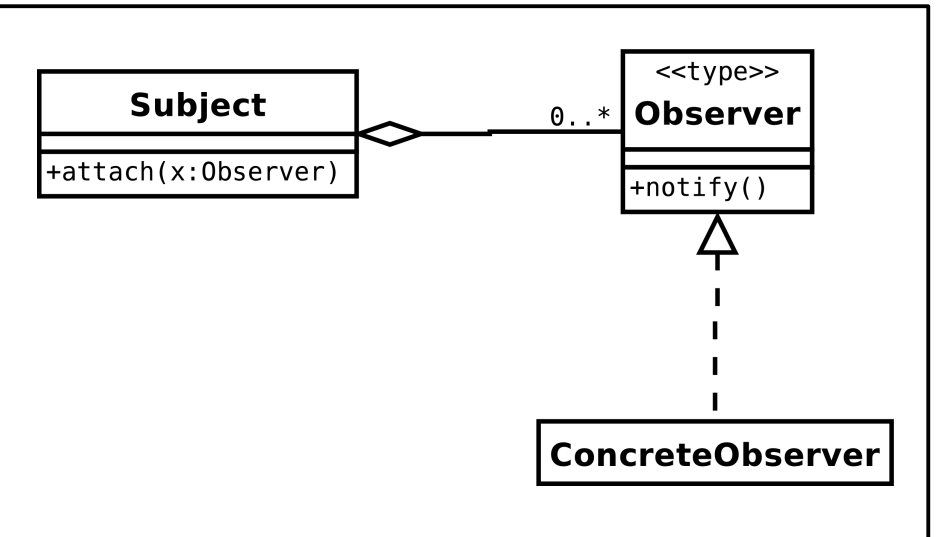
- Contesto:
  - Un oggetto (*soggetto*) genera eventi
  - Uno o più oggetti (*osservatori*) vogliono essere informati del verificarsi di tali eventi
- Soluzione:
  - Definire un'interfaccia con un metodo *notify*, che sarà implementata dagli osservatori
  - Il soggetto ha un metodo (*attach*) per registrare un osservatore
  - Il soggetto gestisce l'elenco dei suoi osservatori registrati
  - Quando si verifica un evento, il soggetto informa tutti gli osservatori registrati, chiamando il loro metodo notify

Diagramma del pattern OBSERVER



Applicazione del pattern OBSERVER



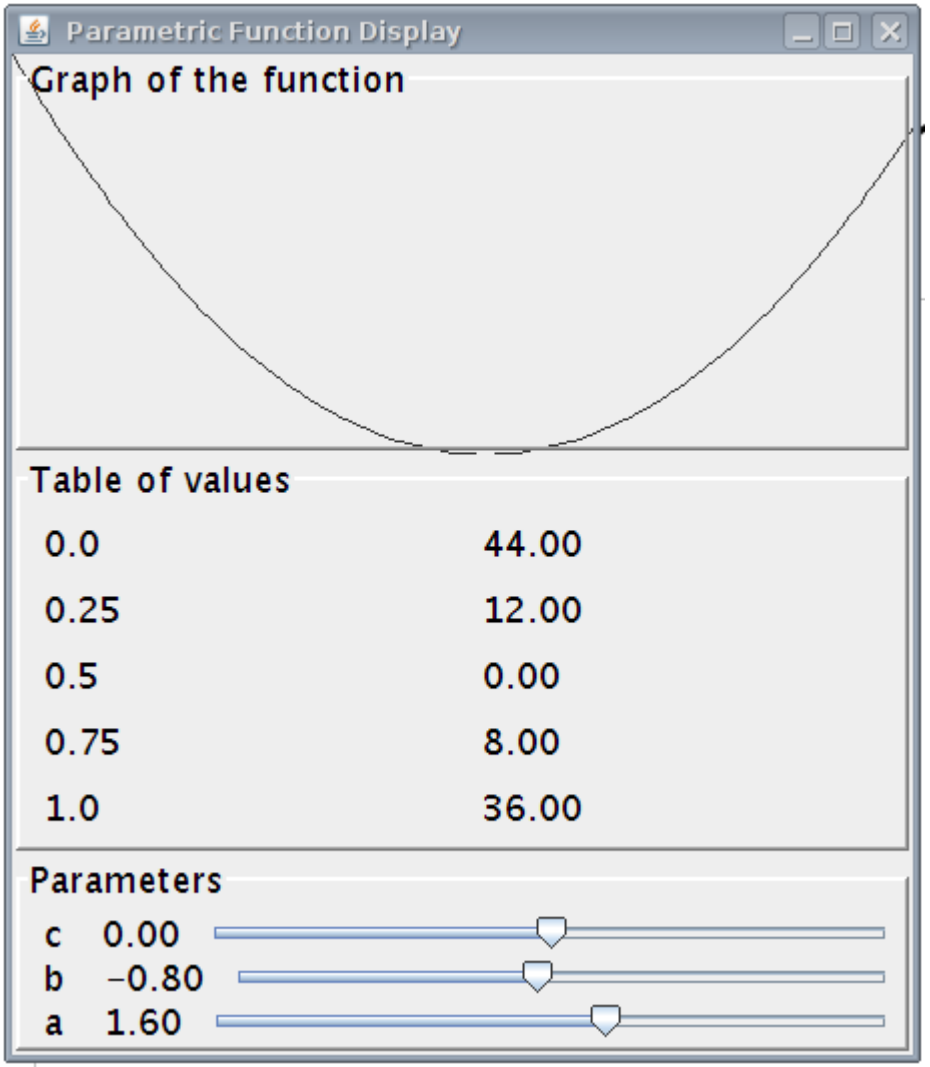


Esempio: funzioni matematiche

- Studio di funzioni parametriche di una variabile reale
- Ad esempio, parabole
  - $a\,x^2 + b\,x + c$; variabile: x; parametri: a, b, c

- Model: la funzione
- View: il suo grafico, o una tabella di valori
- Controller: le classi che rispondono allo spostamento degli slider e modificano i parametri della funzione

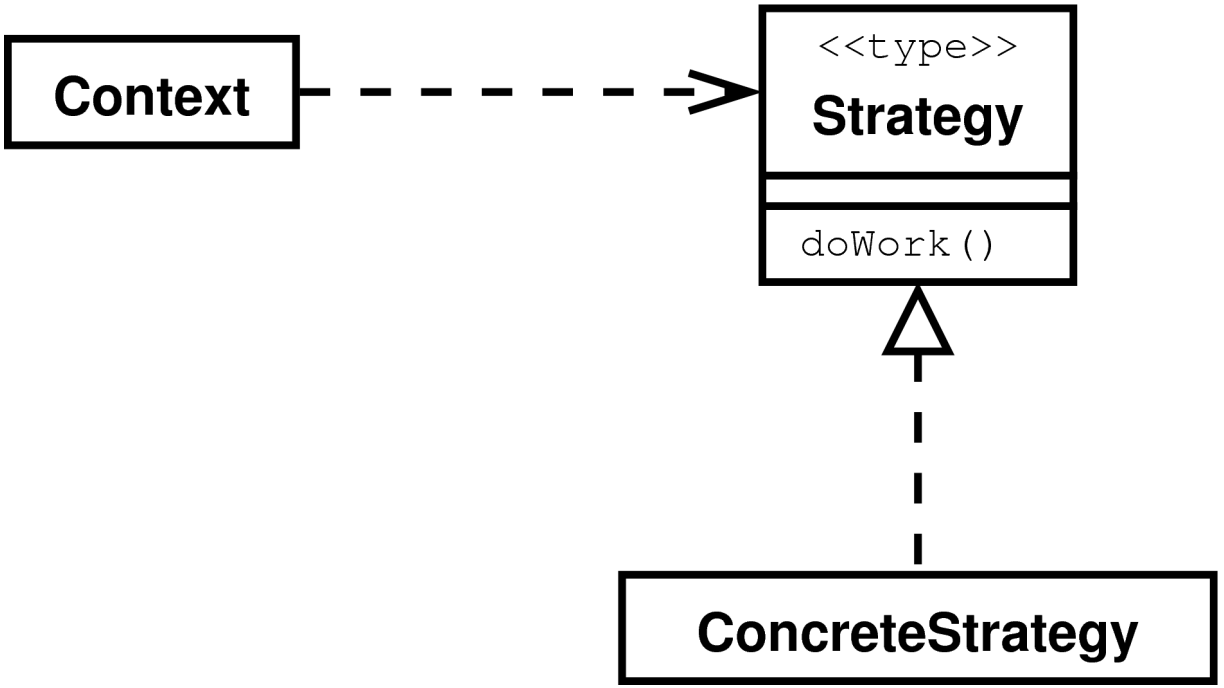
Esempio: funzioni matematiche



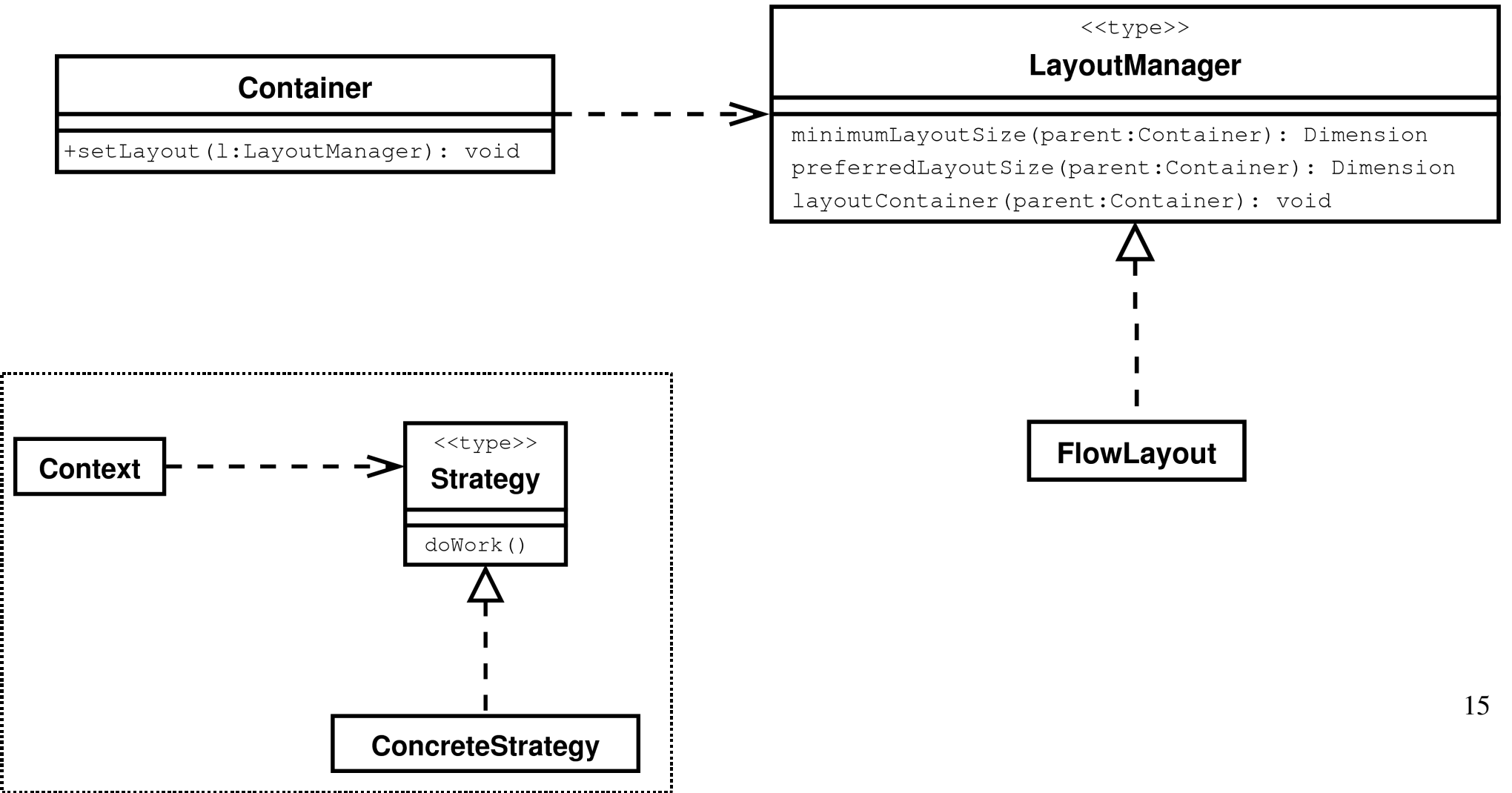
Pattern STRATEGY

- Contesto:
  1) Una classe (*Context*) può sfruttare diverse varianti di un algoritmo
  2) I clienti della classe vogliono fornire versioni particolari dell'algoritmo
- Soluzione:
  1) Definire un'interfaccia (*Strategy*) che rappresenti un'astrazione dell'algoritmo
  2) Per fornire una variante dell'algoritmo, un cliente costruisce un oggetto di una classe (*ConcreteStrategy*) che implementa l'interfaccia Strategy e lo passa alla classe Context
  3) Ogni volta che deve eseguire l'algoritmo, la classe Context invoca il corrispondente metodo dell'oggetto che concretizza la strategia

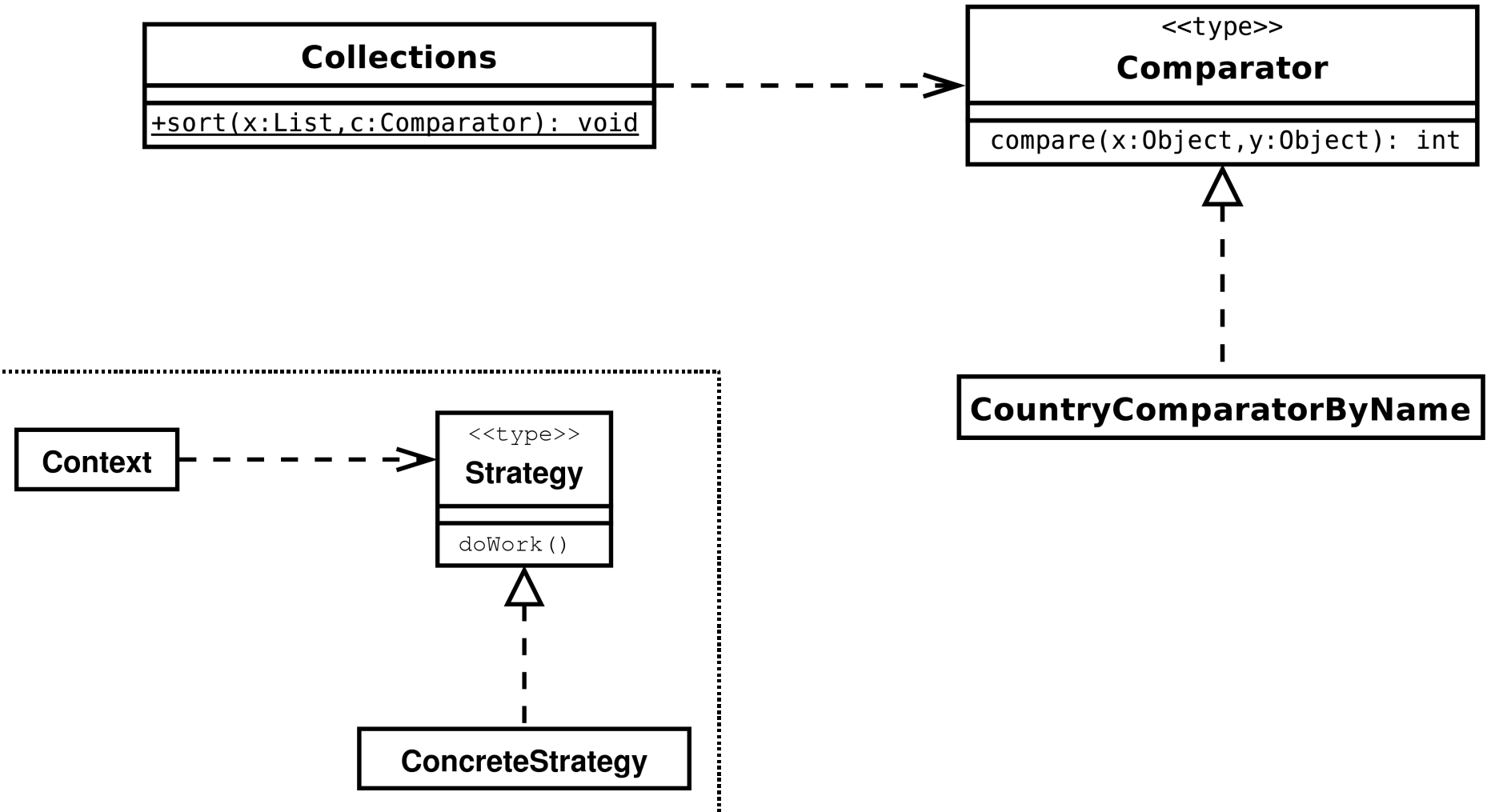
Diagramma del pattern STRATEGY



Applicazione del pattern STRATEGY (1)

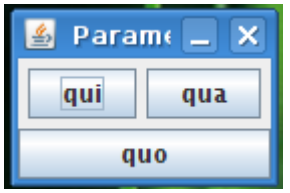


Applicazione del pattern STRATEGY (2)



Contenitori e componenti

output:



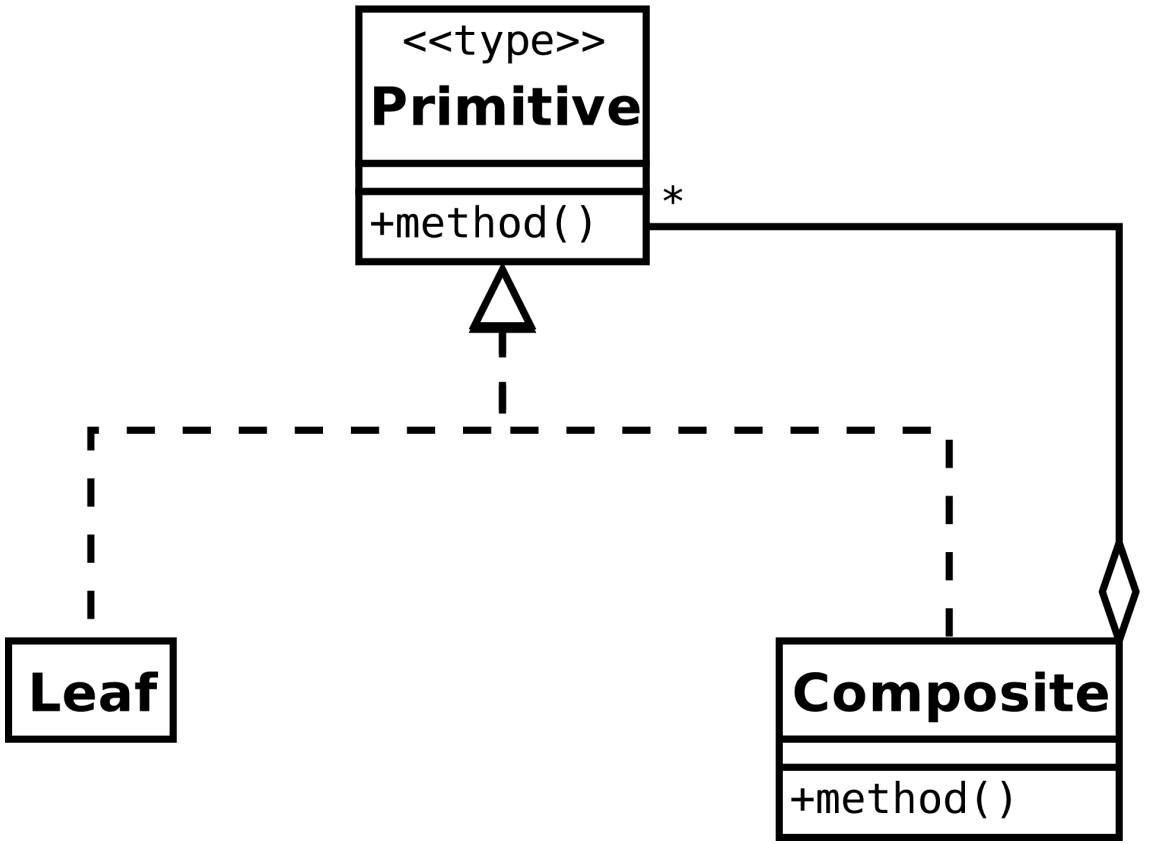

```
// un contenitore
Container x = new Container();
x.setLayout(new FlowLayout());
// aggiungo un componente al contenitore
x.add(new Jbutton("qui"));
// aggiungo un altro componente al contenitore
x.add(new Jbutton("qua"));

// la finestra
JFrame f = new JFrame();
Container c = f.getContentPane();
c.setLayout(new FlowLayout());
c.add(x); // x si comporta come un componente!
c.add(new JButton("quo"));
...
```

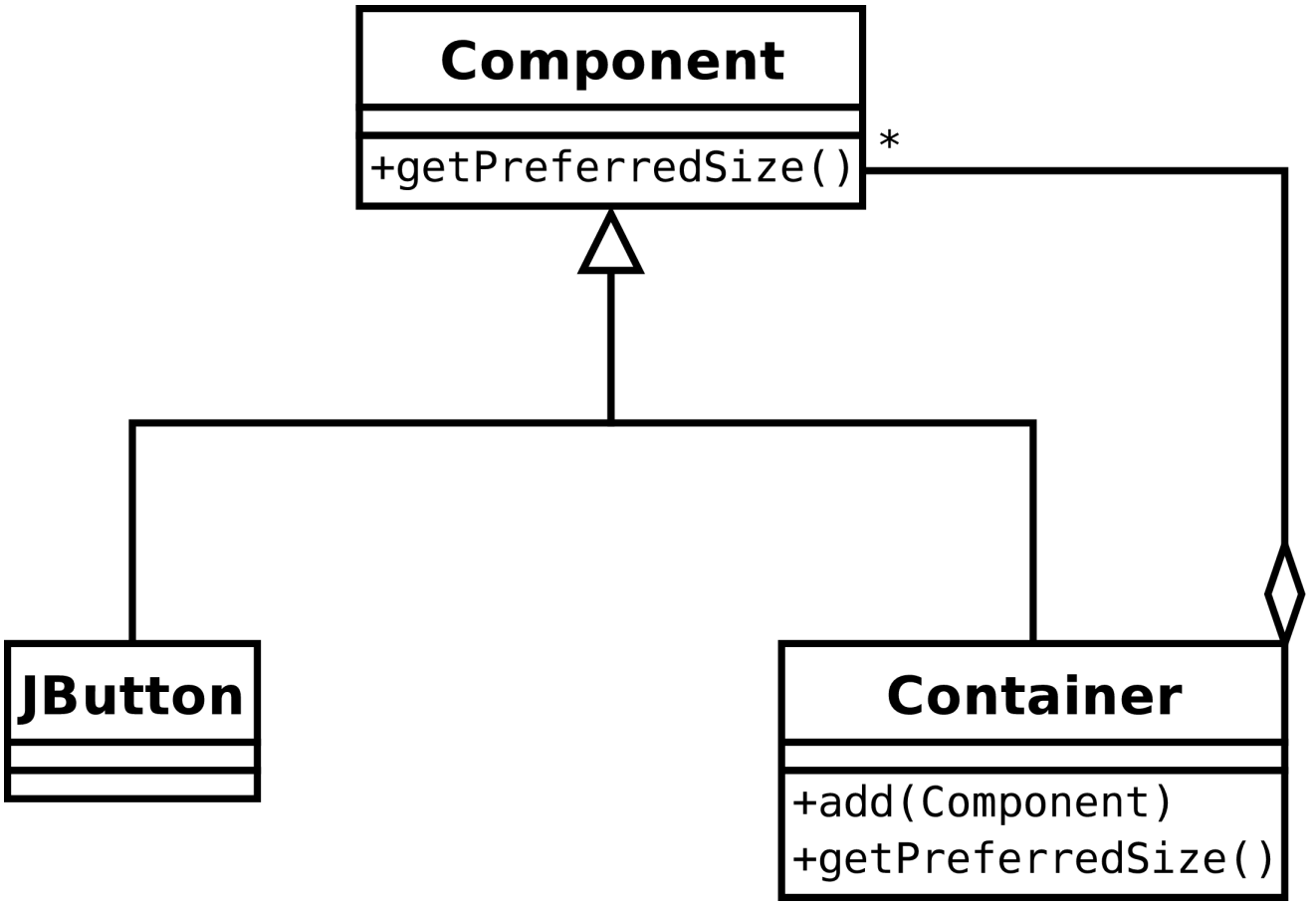
Pattern COMPOSITE

- Contesto:
  1) Oggetti primitivi possono essere combinati in un oggetto composito
  2) I clienti possono trattare un oggetto composito come primitivo
- Soluzione:
  1) Definire un'interfaccia (*Primitive*) che rappresenti un'astrazione dell'oggetto primitivo
  2) Un oggetto composito contiene una collezione di oggetti primitivi
  3) Sia gli oggetti primitivi che quelli compositi implementano l'interfaccia Primitive
  4) Nel realizzare un metodo dell'interfaccia Primitive, un oggetto composito applica il metodo corrispondente a tutti i propri oggetti primitivi, e poi combina i risultati ottenuti

Diagramma del pattern COMPOSITE



Applicazione del pattern COMPOSITE



Barre di scorrimento

```
/* un'area di testo con 10 righe e 25 colonne */
Component area = new JTextArea(10, 25);

/* aggiungiamo le barre di scorrimento */
Component scrollArea = new JScrollPane(area);
```

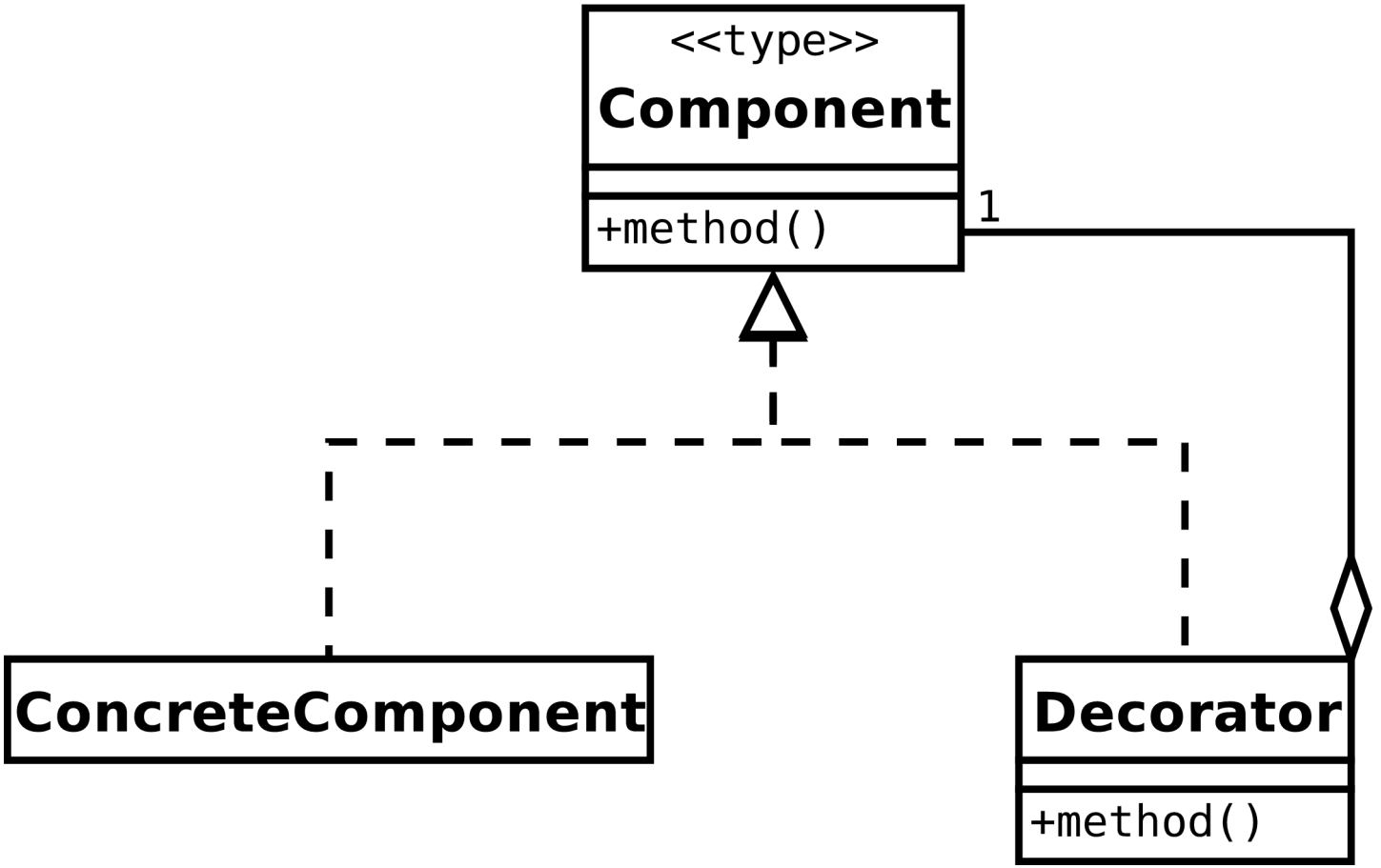
Pattern DECORATOR

- Contesto:
  1) Si vuole *decorare* (ovvero migliorare, ovvero aggiungere funzionalità a) una classe **componente**
  2) Un componente decorato può essere utilizzato nello stesso modo di uno normale
  3) La classe componente non vuole assumersi la responsabilità della decorazione
  4) L'insieme delle decorazioni possibili non è limitato

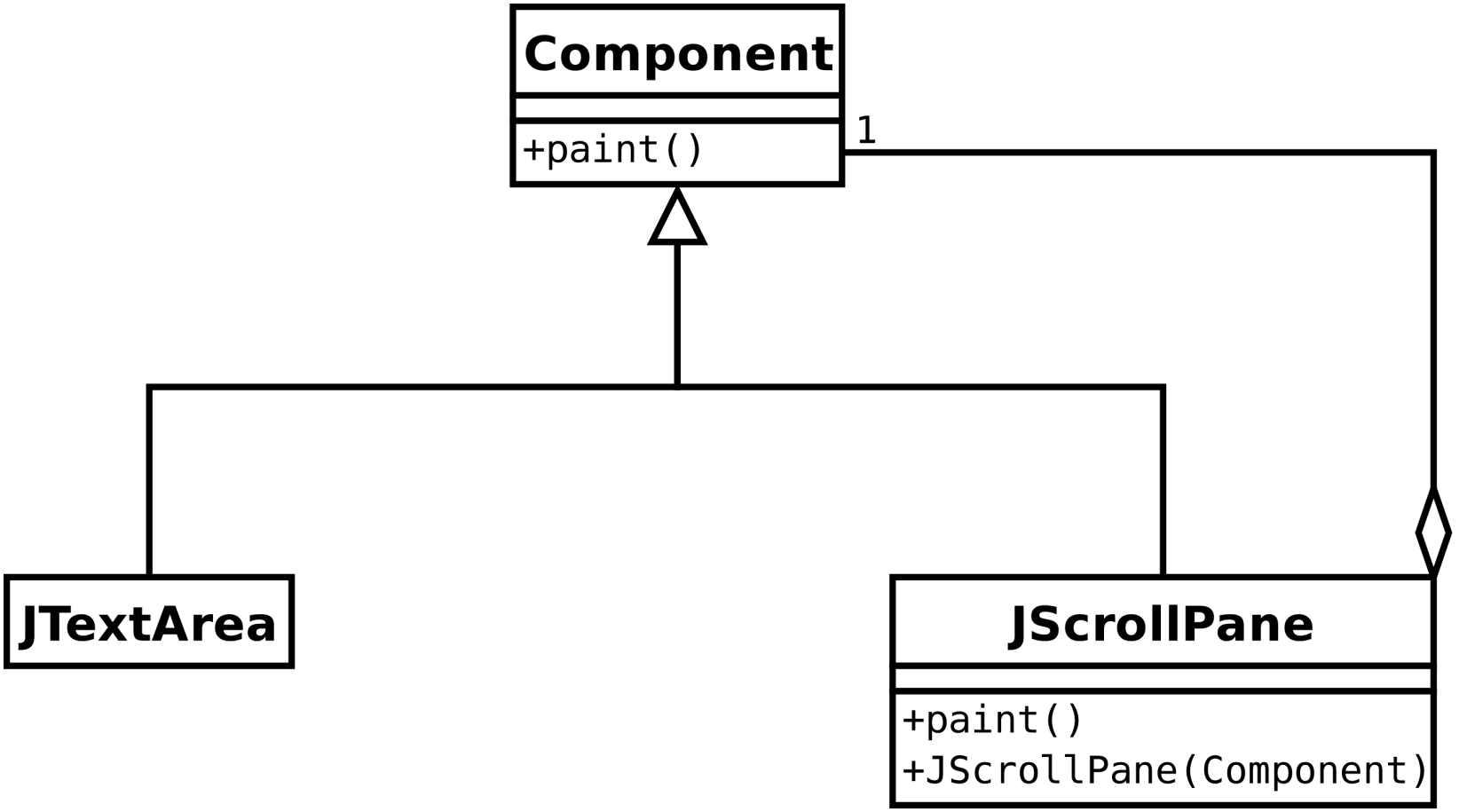
Pattern DECORATOR

- Soluzione:
  1) Definire un'interfaccia (*Component*) che rappresenti un'astrazione di un componente
  2) Le classi concrete che definiscono componenti implementano l'interfaccia *Component*
  3) Definire una classe (*Decorator*) che rappresenta la decorazione
  4) Un oggetto decoratore contiene e gestisce l'oggetto che decora
  5) Un oggetto decoratore implementa l'interfaccia *Component*
  6) Nel realizzare un metodo di *Component*, un oggetto decoratore applica il metodo corrispondente all'oggetto decorato e ne combina il risultato con l'effetto della decorazione

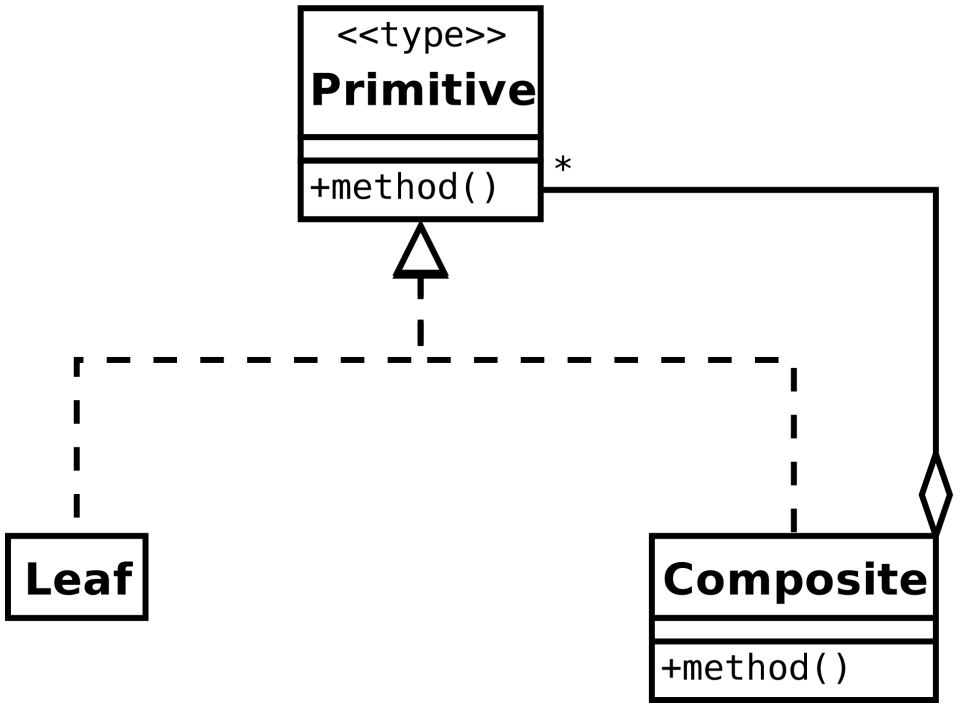
Diagramma del pattern DECORATOR

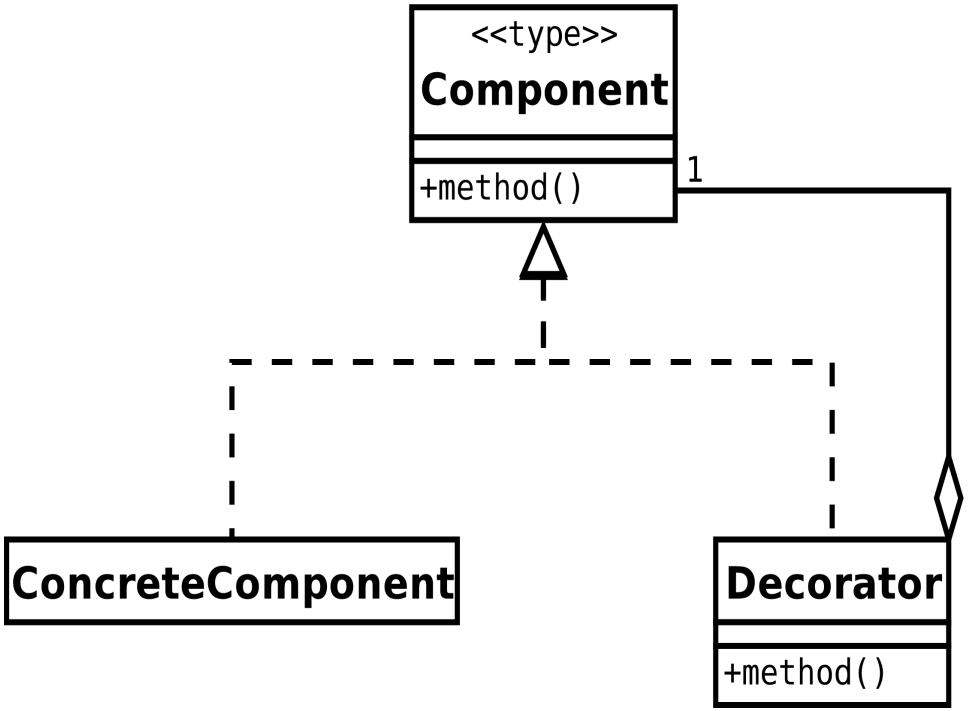


Applicazione del pattern DECORATOR



COMPOSITE vs DECORATOR





Pattern TEMPLATE METHOD

- 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

Pattern TEMPLATE METHOD

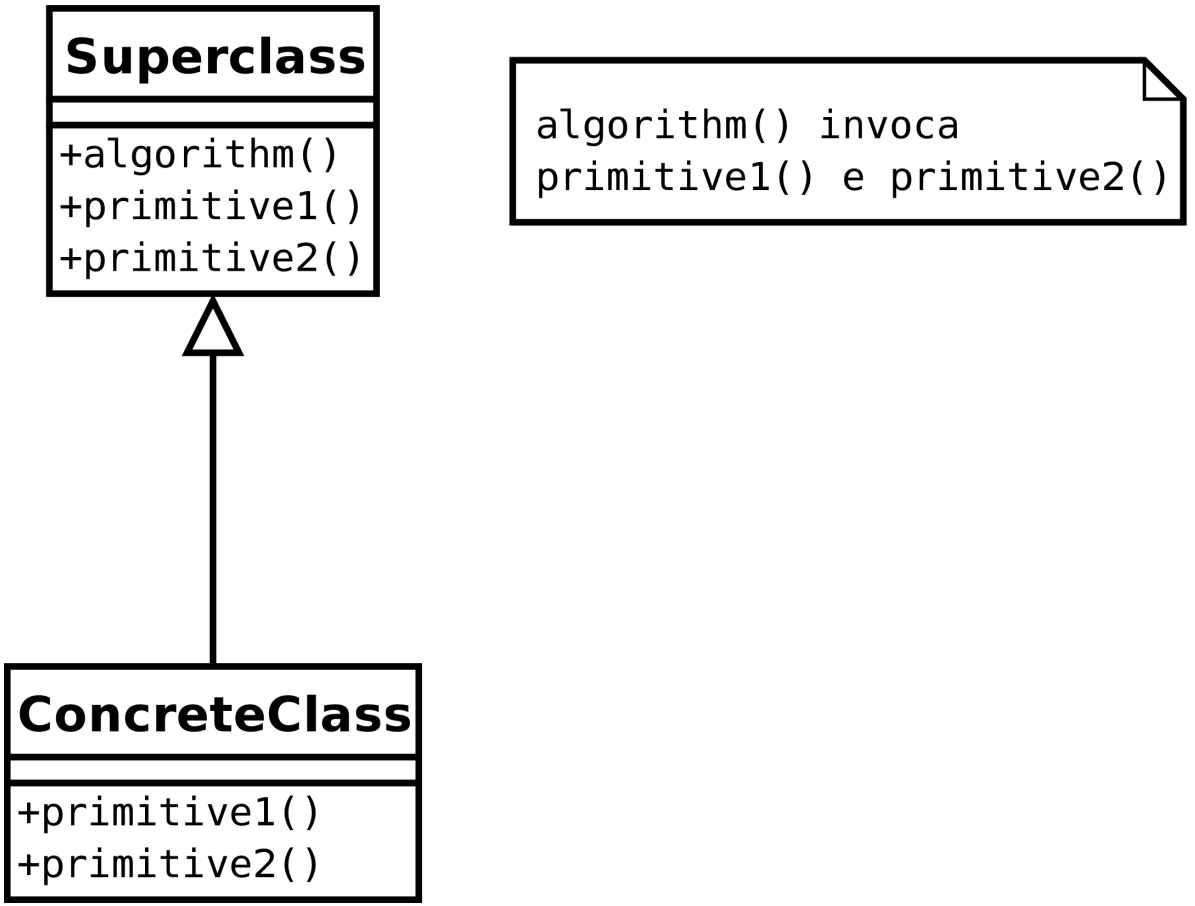
- Soluzione:

  1) Definire una superclasse che abbia un metodo che realizza l'algoritmo e un metodo per ogni operazione primitiva

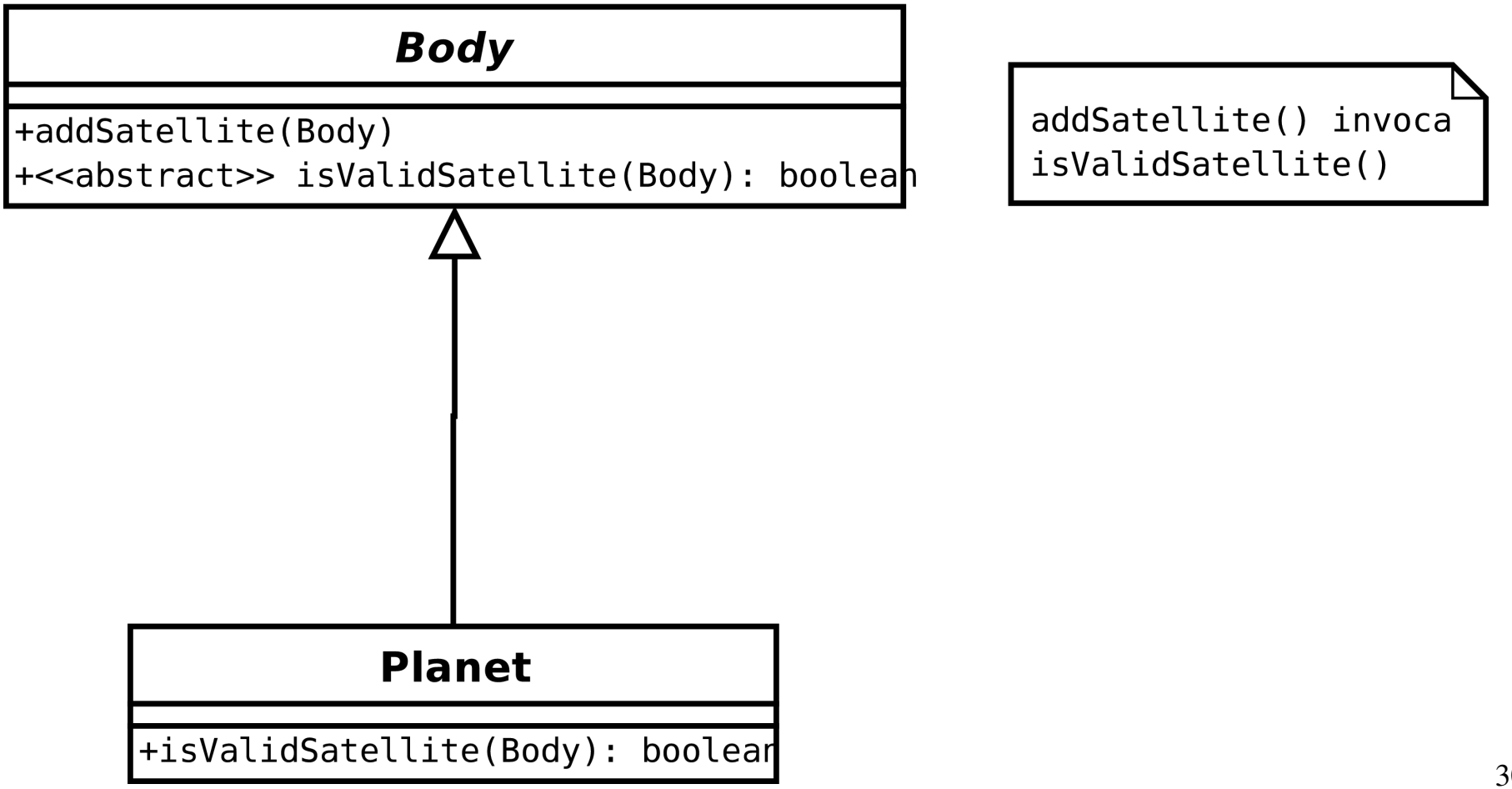
  2) Le operazioni primitive non sono implementate nella superclasse (metodi astratti), oppure sono implementate in modo generico

  3) Ridefinire in ogni sottoclasse i metodi che rappresentano operazioni primitive, ma non il metodo che rappresenta l'algoritmo

Diagramma del pattern TEMPLATE METHOD



Applicazione del pattern TEMPLATE METHOD



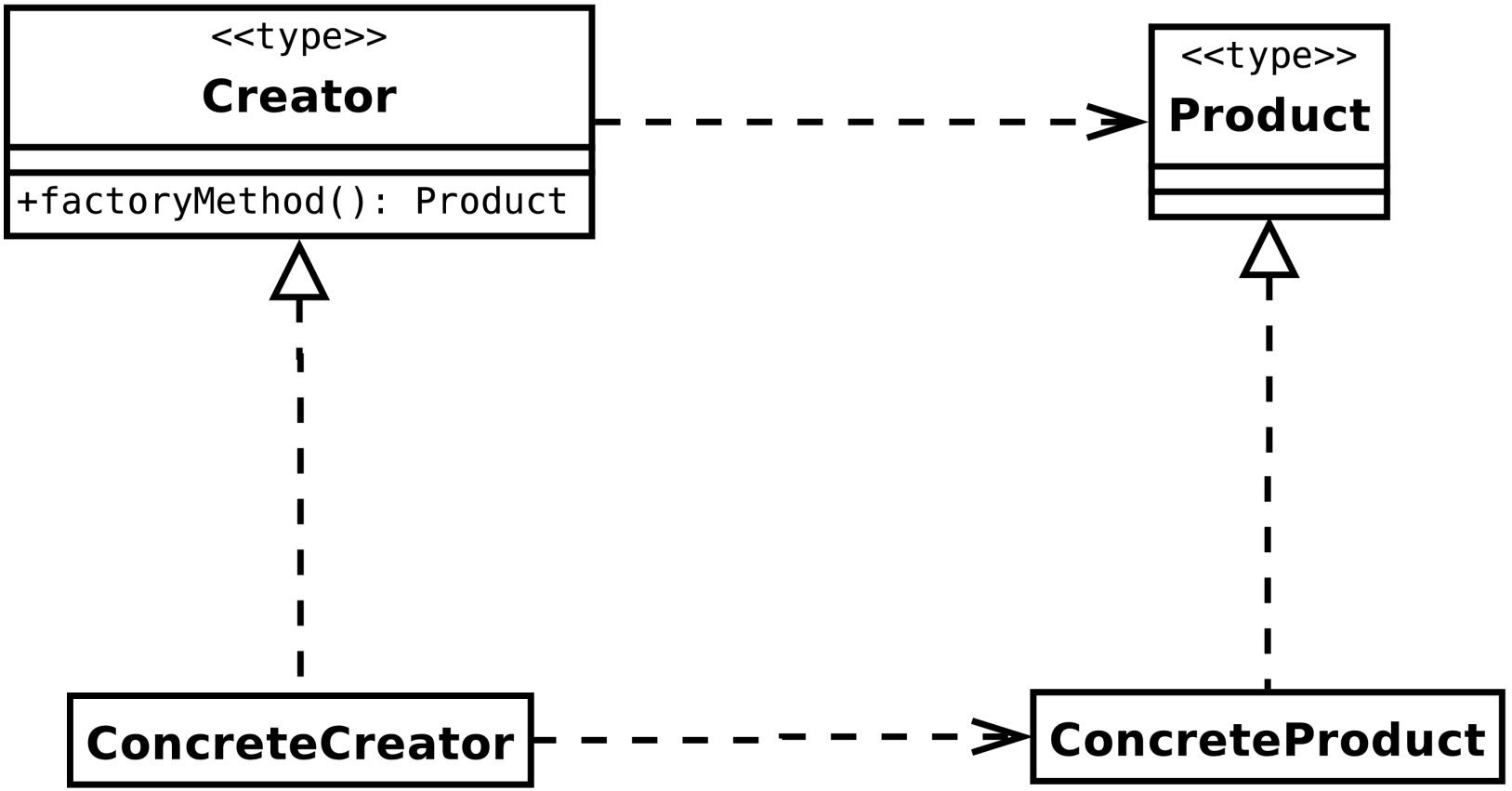
Pattern FACTORY METHOD

- Contesto:
  1) Un tipo (*creatore*) crea oggetti di 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

Pattern FACTORY METHOD

- 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 *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



Applicazione del pattern FACTORY METHOD