Alcuni design pattern

Cosa sono i design pattern

- I problemi incontrati nello sviluppo di grossi progetti software sono spesso **ricorrenti** e prevedibili
- I design pattern sono “schemi di soluzioni” riutilizzabili
- Permettono quindi di non inventare da capo soluzioni ai problemi già risolti, ma di utilizzare dei “mattoni” di provata efficacia
  - Un bravo progettista sa riconoscerli, nella documentazione o direttamente nel codice, e utilizzarli per comprendere i programmi scritti da altri
  - Forniscono quindi un vocabolario comune che facilita la comunicazione tra progettisti
- ..ma possono anche rendere la struttura del progetto/codice più complessa del necessario

Definizione

Descrizione di oggetti e classi comunicanti adattabili per risolvere un problema ricorrente di progettazione in un contesto specifico

- Sufficientemente **astratti**
  - In modo da poter essere condivisi da progettisti con punti di vista diversi
- Non complessi e **non domain-specific**
  - Non rivolti alla specifica applicazione ma riusabili in applicazioni diverse

Classificazione dei pattern
(Gamma, Helm, Johnson, and Vlissides)

- Pattern **creazionali**
  - Riguardano il processo di creazione di oggetti
- Pattern **strutturali**
  - Hanno a che fare con la composizione di classi ed oggetti
- Pattern **comportamentali**
  - Si occupano di come interagiscono gli oggetti e distribuiscono fra loro le responsabilità

Lista completa

- Creational Patterns
  - Abstract Factory: Creates an instance of several families of classes
  - Builder: Separates object construction from its representation
  - Factory Method: Creates an instance of several derived classes
  - Prototype: A fully initialized instance to be copied or cloned
  - Singleton: A class of which only a single instance can exist

- Structural Patterns
  - Adapter: Match interfaces of different classes
  - Bridge: Separates an object's interface from its implementation
  - Composite: A tree structure of simple and composite objects
  - Decorator: Add responsibilities to objects dynamically
  - Facade: A single class that represents an entire subsystem
  - Flyweight: A fine-grained instance used for efficient sharing
  - Proxy: An object representing another object

Lista completa

- Behavioral Patterns
  - Chain of Resp.: A way of passing a request between a chain of objects
  - Command: Encapsulate a command request as an object
  - Interpreter: A way to include language elements in a program
  - Iterator: Sequentially access the elements of a collection
  - Mediator: Defines simplified communication between classes
  - Memento: Capture and restore an object's internal state
  - Observer: A way of notifying change to a number of classes
  - State: Alter an object's behavior when its state changes
  - Strategy: Encapsulates an algorithm inside a class
  - Template Method: Defer the exact steps of an algorithm to a subclass
  - Visitor: Defines a new operation to a class without change

Creazione di oggetti

- Limitare le dipendenze dalle classi è desiderabile perché permette di sostituire un'implementazione con un'altra
  - …ad esempio, utilizzando Interfacce ed assegnamenti polimorfici in Java
- Eccezione: le chiamate ai costruttori
  - Il codice utente che chiama il costruttore di una determinata classe rimane vincolato a quella classe
- Per questo esistono pattern per un'operazione semplice come la **creazione di un oggetto**
  - Disaccoppiano (separandolo) il codice che fa uso di un tipo da quello che sceglie quale implementazione del tipo (classe) utilizzare

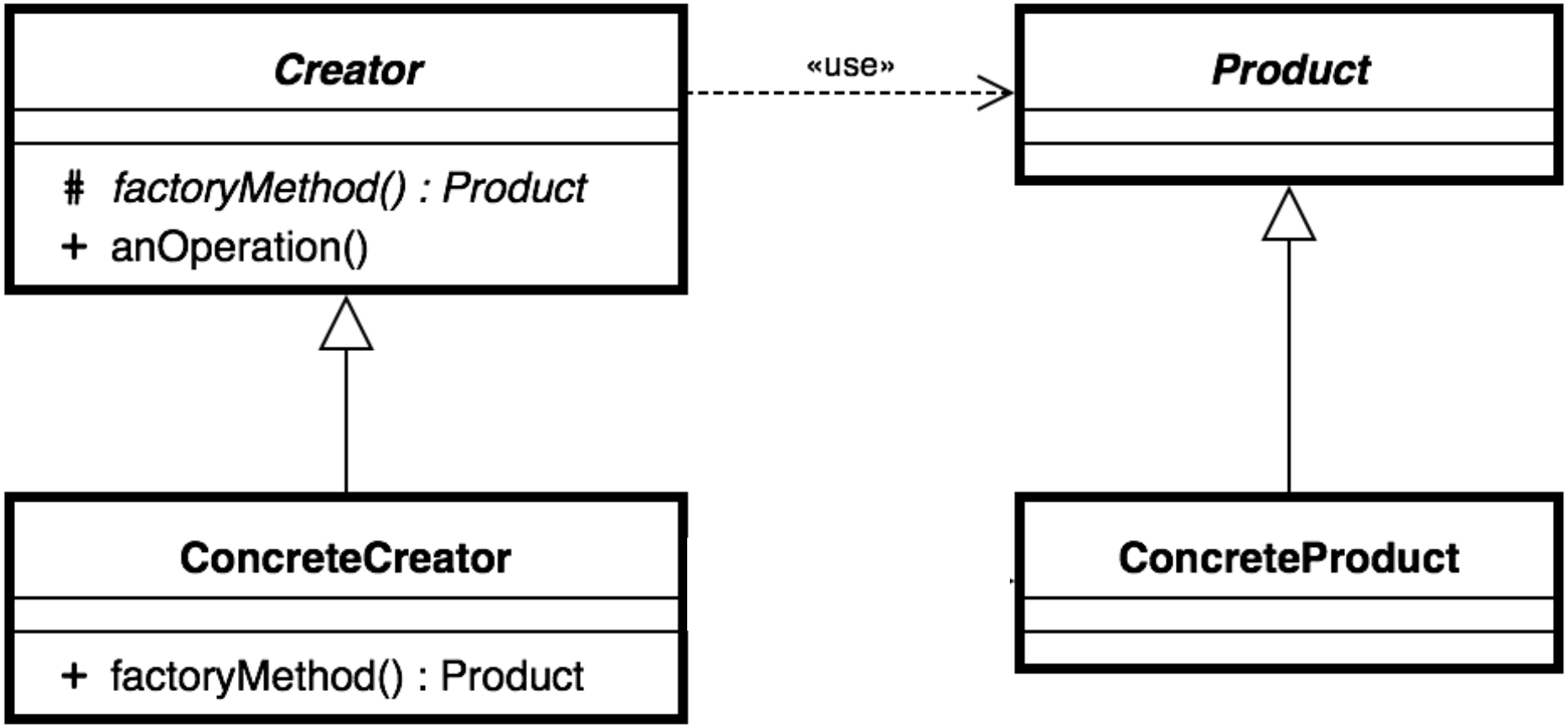
Factory Method

- In Java le chiamate ai costruttori non sono personalizzabili
- La soluzione è nascondere la creazione in un metodo detto **factory**: un metodo che restituisce un oggetto di una classe senza essere costruttore di quella classe
- Un **factory method** può invece scegliere internamente la strategia di creazione

Factory Method

- Soluzione
  - ll costruttore è protected o private
  - La creazione è possibile invocando un metodo pubblico statico, detto factory method
    - Restituisce un oggetto di una classe senza essere costruttore di quella classe
- La soluzione con un factory method è generale
  - Factory è uno dei pattern più usati

UML


Creator
factoryMethod() : Product
+ anOperation()
«use»
Product
ConcreteCreator
+ factoryMethod() : Product
ConcreteProduct

Esempio

```
public interface Trace {

    // turn on and off debugging
    public void setDebug(boolean debug);

    // write out a debug message
    public void debug(String message);

    // write out an error message
    public void error(String message);
}
```

Esempio

```
public class SystemTrace implements Trace {

    private boolean debug;

    SystemTrace() {
        debug = false;
    }

    public void setDebug(boolean debug) {
        this.debug = debug;
    }

    public void debug(String message) {
        if(debug) {  // only print if debug is true
            System.out.println("DEBUG: " + message);
        }
    }

    public void error( String message ) {
            // always print out errors
        System.out.println("ERROR: " + message);
    }
}
```

```
public class FileTrace implements Trace {
   private java.io.PrintWriter pw;
   private boolean debug;

   FileTrace() throws java.io.IoException {

   // a real FileTrace would need to obtain the filename
   // somewhere for the example it is hardcoded

      pw = new java.io.PrintWriter(new
                java.io.FileWriter("trace.log"));
   }

   public void setDebug(boolean debug) {
       this.debug = debug;
   }

   public void debug( String message ) {
      if( debug ) {  // only print if debug is true
         pw.println("DEBUG: " + message);
         pw.flush();
      }
   }

   public void error(String message) {
   // always print out errors

      pw.println("ERROR: " + message);
      pw.flush();
   }
}
```

Esempio

```
public static void main(String[] args) {
    Trace t = new SystemTrace();
    ...
}
```

Senza usare il Factory pattern…

```
public class FactoryTrace {
    public static Trace getTrace() {
        try {
            return new FileTrace();
        }
        catch ( java.io.IOException ex ) {
            Trace t = new SystemTrace();
            t.error("could not instantiate FileTrace: " +
                                                ex.getMessage());
            return t;
        }
    }
}
```

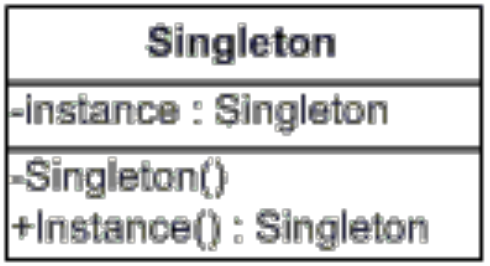
La Factory…

…e il suo utilizzo

```
public static void main(String[] args) {
    Trace t = FactoryTrace.getTrace();
    ...
}
```

Singleton

- A volte una classe viene usata per definizione per istanziare **un solo oggetto**
- Usare una normale classe con soli metodi statici non assicura che esista un solo esemplare della classe, se viene reso visibile il costruttore
- In una classe Singleton il costruttore è protetto o privato
- Un metodo statico fornisce l'accesso alla sola copia dell'oggetto

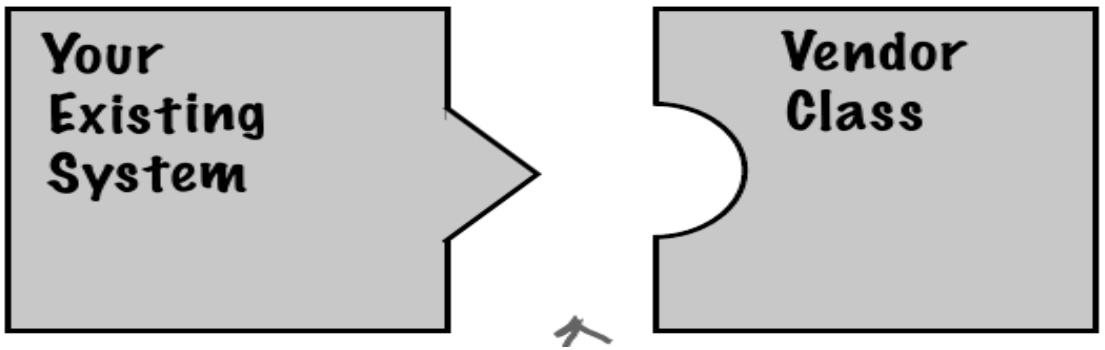


Esempio

```
public class Singleton {
   private static Singleton instance; // istanza singola

   private Singleton() { } // costruttore

   public static Singleton instance() {
      // crea l'oggetto solo se non esiste
      if (instance == null) instance = new Singleton();
      return instance;
   }

   // altri metodi
}
```

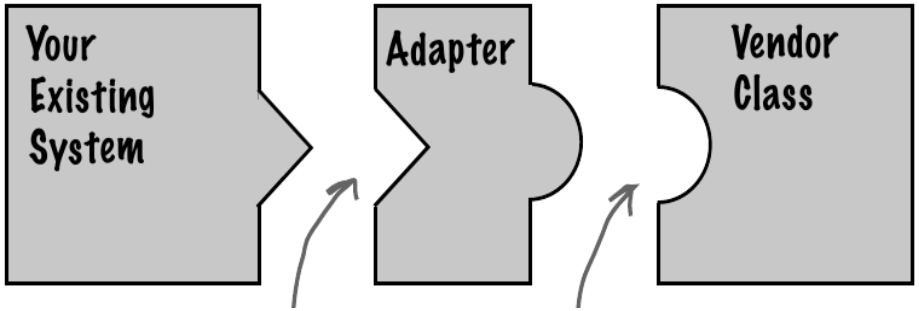
“Adattare” interfacce diverse

- Un software esistente usa una data interfaccia
- Un altro software esistente ne usa un'altra
- Come combinarle senza cambiare il software?
  - Molto spesso librerie diverse espongono interfacce diverse… per fare la stessa cosa
  - Ad esempio, le librerie grafiche:
    - …software scritto per Windows che viene portato sotto MacOS o X (ambienti grafici incompatibili tra loro)

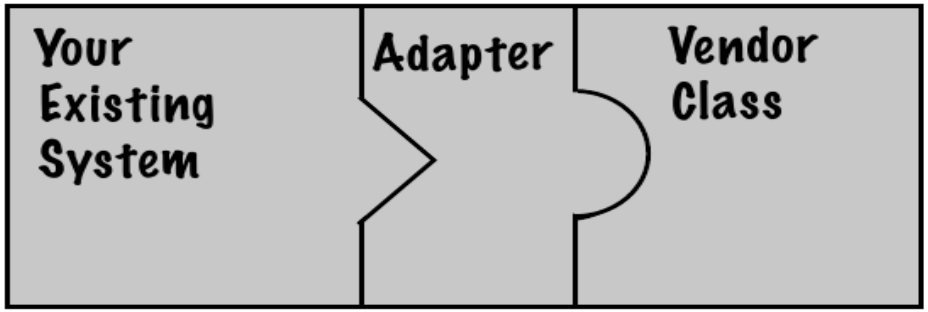


Adattare

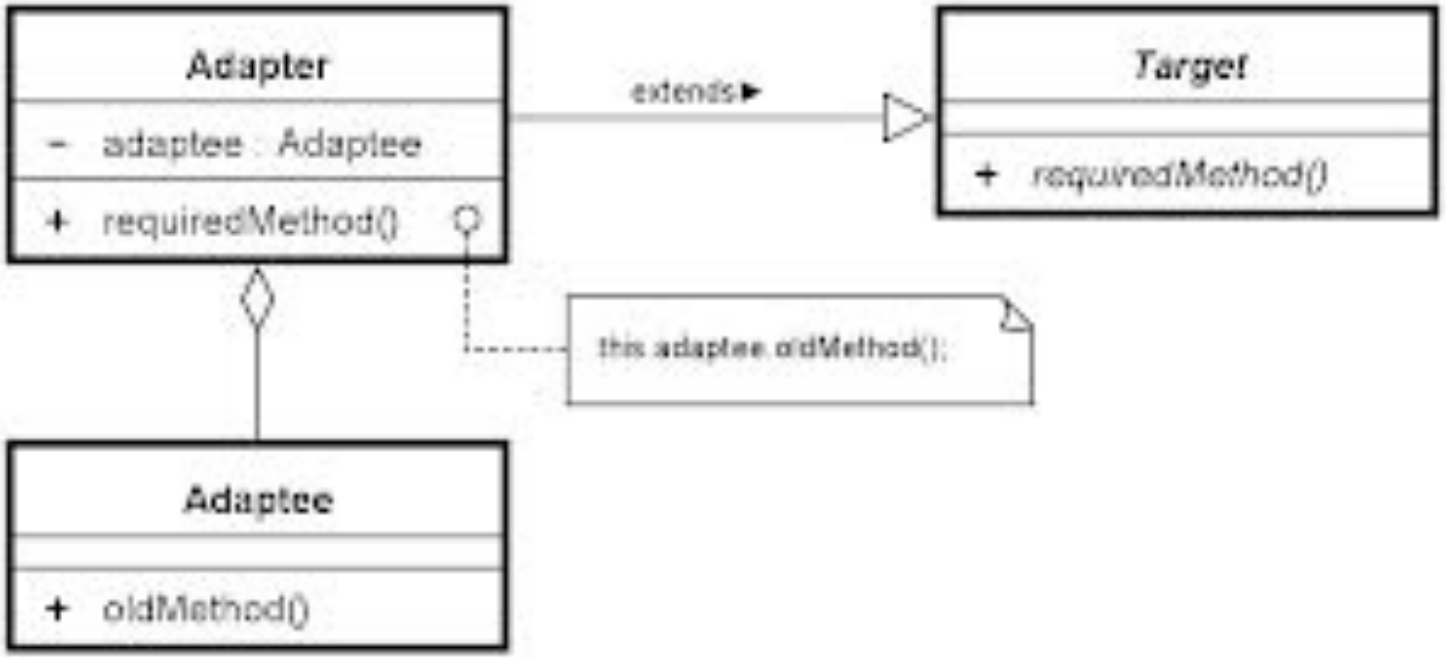
- Si interpone un **Adapter** con interfaccia esistente che richiama i metodi dell'oggetto da adattare (**Adaptee**)



- Chi deve sfruttare il metodo della classe da adattare “vede” solo un’interfaccia o classe astratta “Target”
  - Una classe concreta che implementa l'interfaccia si occupa di “adattare” i metodi verso gli Adaptee
- Risultato: non si modifica ne’ l’utilizzatore ne’ l’Adaptee



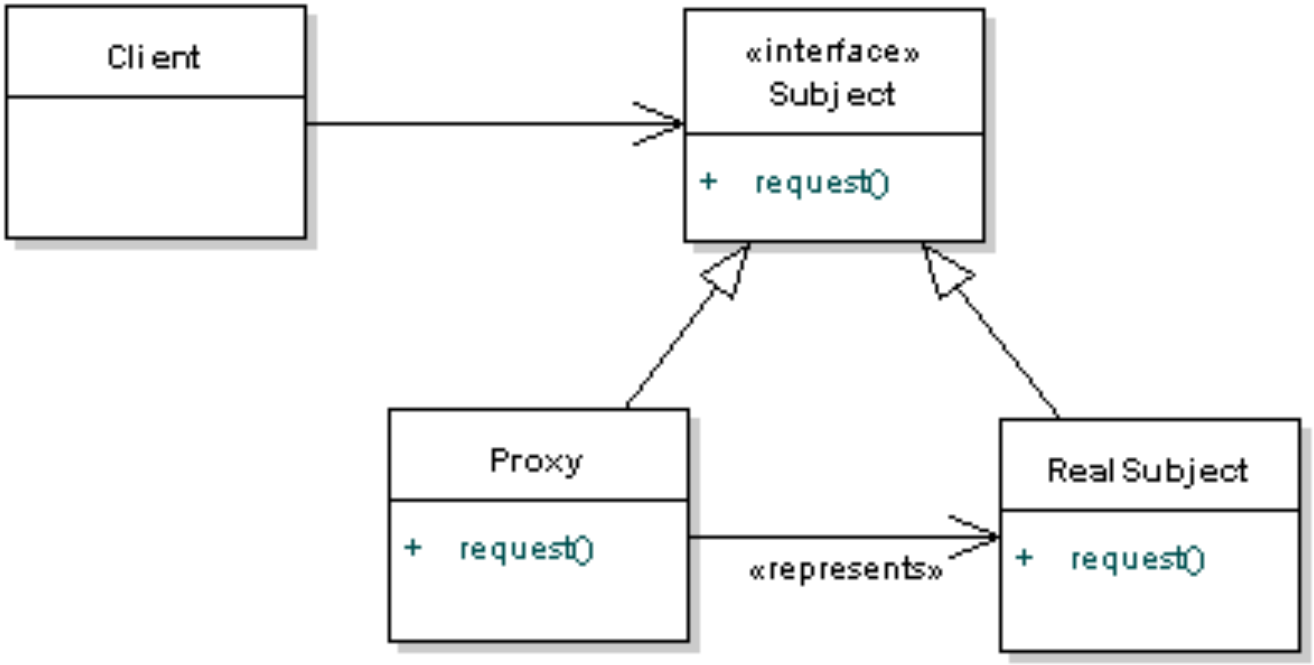
UML


Adapter
- adaptee : Adaptee
+ requiredMethod()
extends
Target
+ requiredMethod()
this.adaptee.oldMethod();
Adaptee
+ oldMethod()

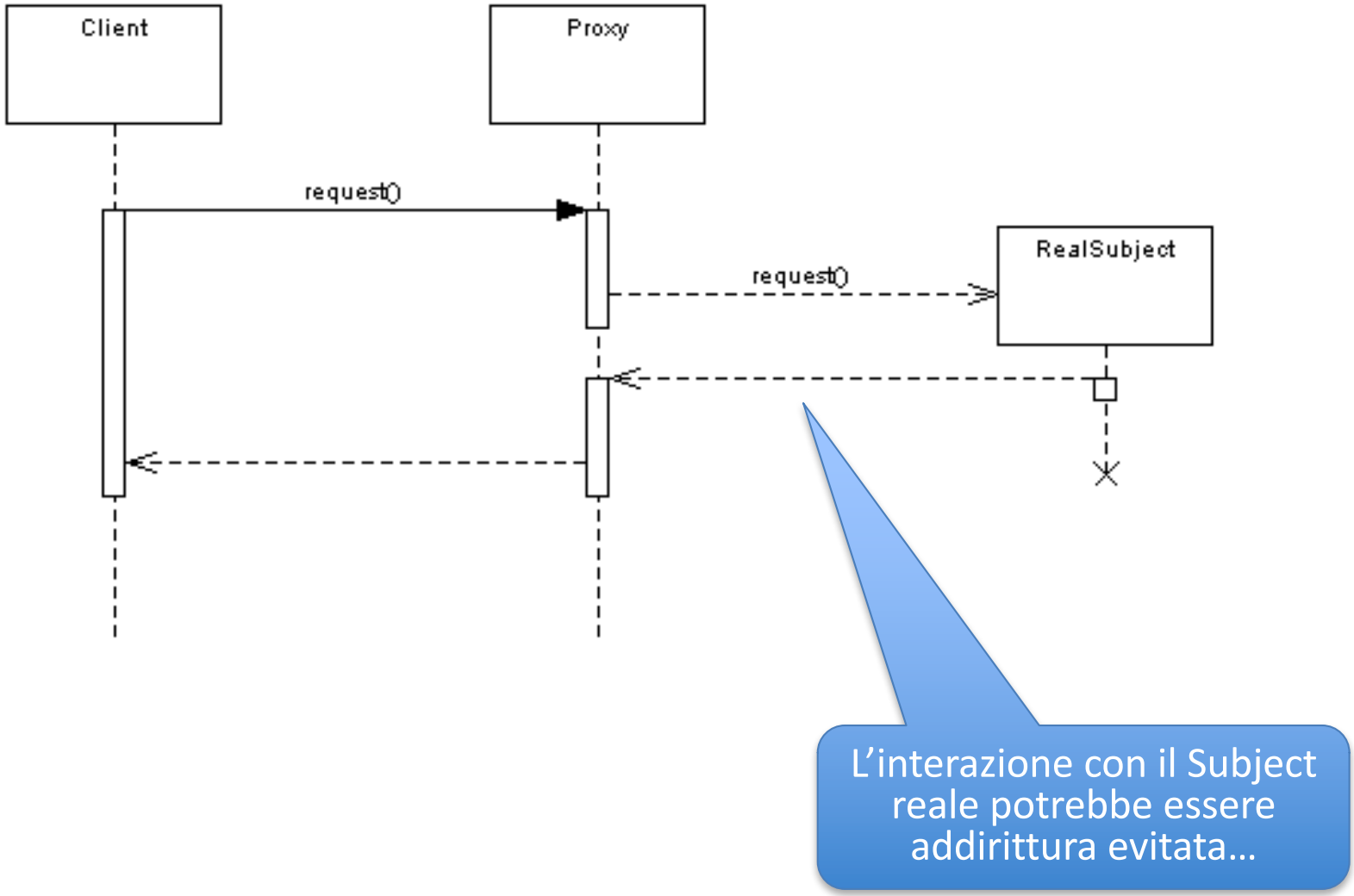
Proxy

- Posporre o addirittura evitare l‘istanziazione di oggetti “pesanti” se non necessaria
- Si interpone un oggetto (**proxy**) con la **stessa interfaccia** dell’oggetto “pesante” di cui fa le veci
  - L’oggetto può fare preprocessing
    - … ad esempio, serializzazione e trasmissione dati perché l'oggetto “pesante” è su un server remoto
  - A volte il proxy può rispondere alle richieste direttamente se è in grado di farlo
    - … ad esempio, se memorizza una cache delle richieste precedenti
- I client dell'oggetto alle volte chiamano i metodi di un **Subject**, a sua volta super-classe del Proxy

UML


Client
«interface»
Subject
+ request()
Proxy
+ request()
«represents»
Real Subject
+ request()

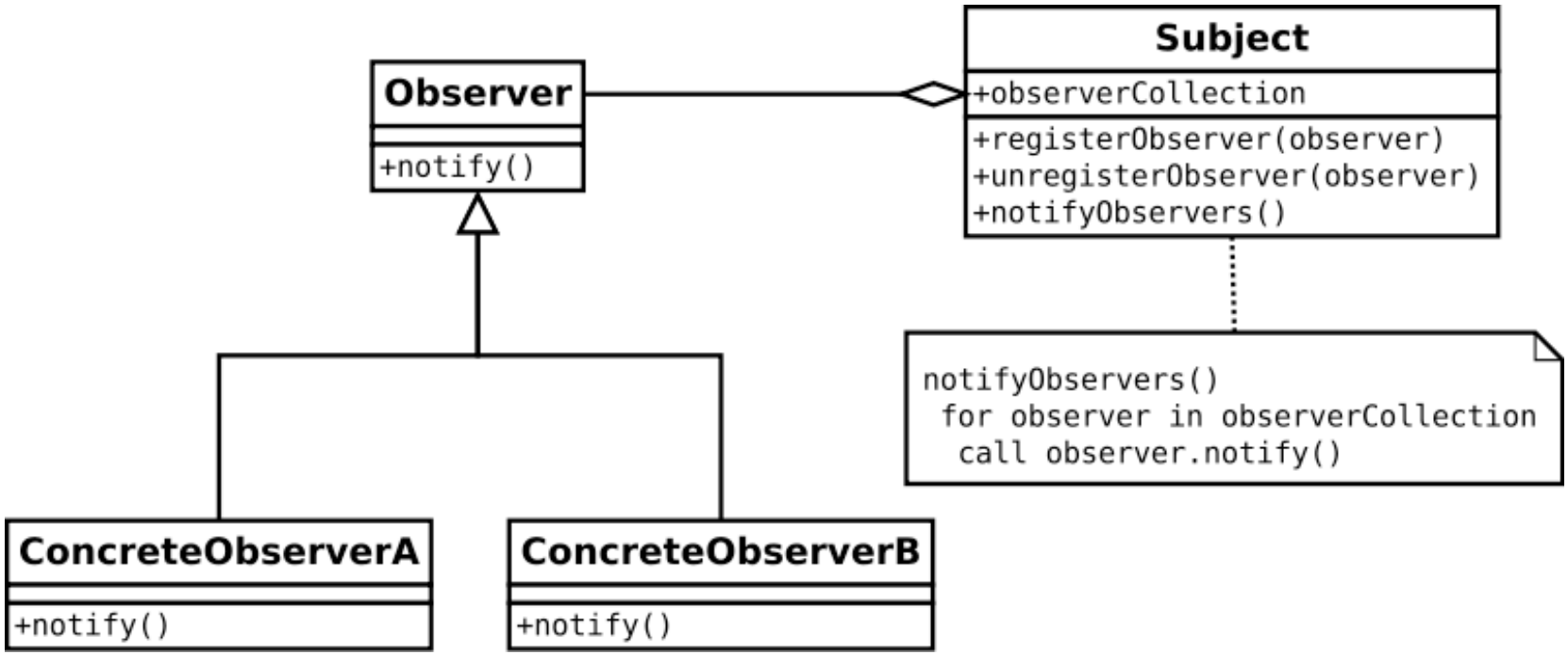
Sequence diagram



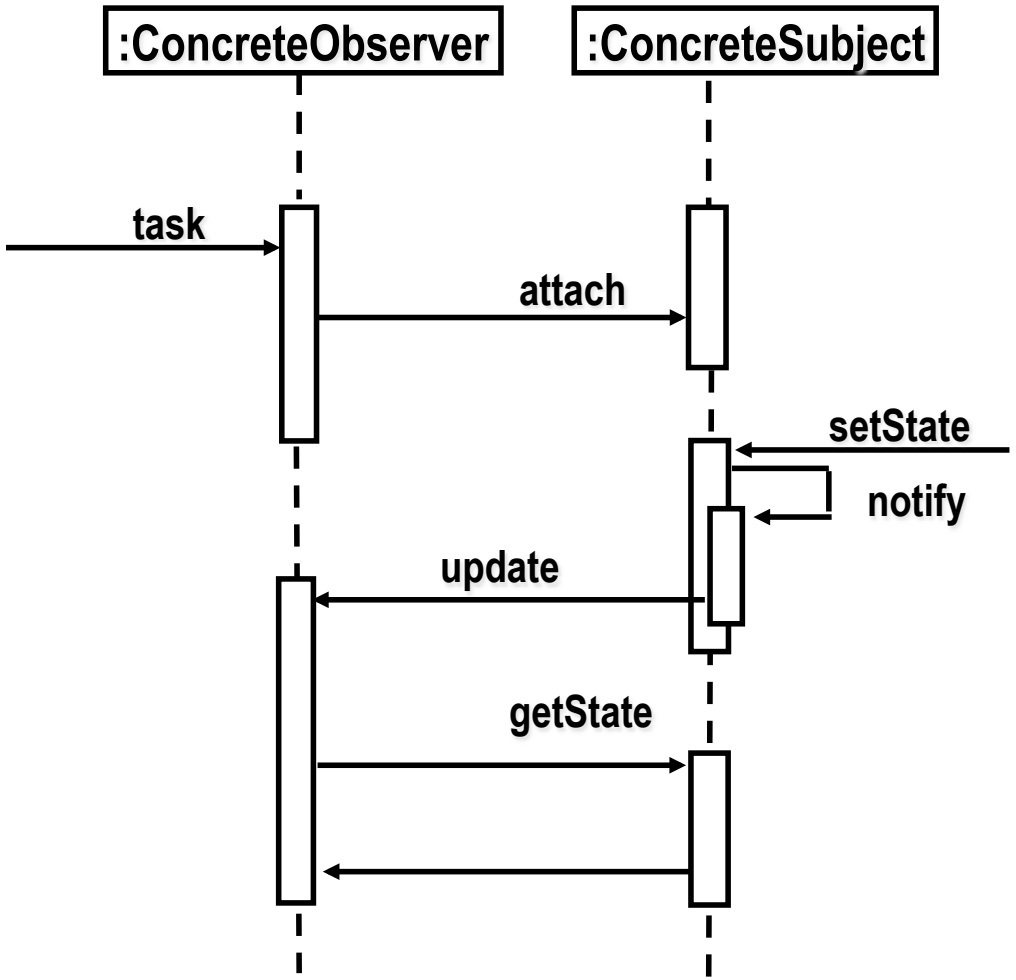
Observer

- Usato da Swing per la creazione di interfacce grafiche con Java
- Ruoli
  - **Subject** (chi è osservato)
  - **Observer** (detti anche **Listener**)
- Più oggetti possono essere interessati ad essere informati quando un Subject cambia stato
- Il soggetto non ha legami con il numero e tipo degli osservatori
  - Questi possono anche cambiare a runtime

Diagramma delle classi



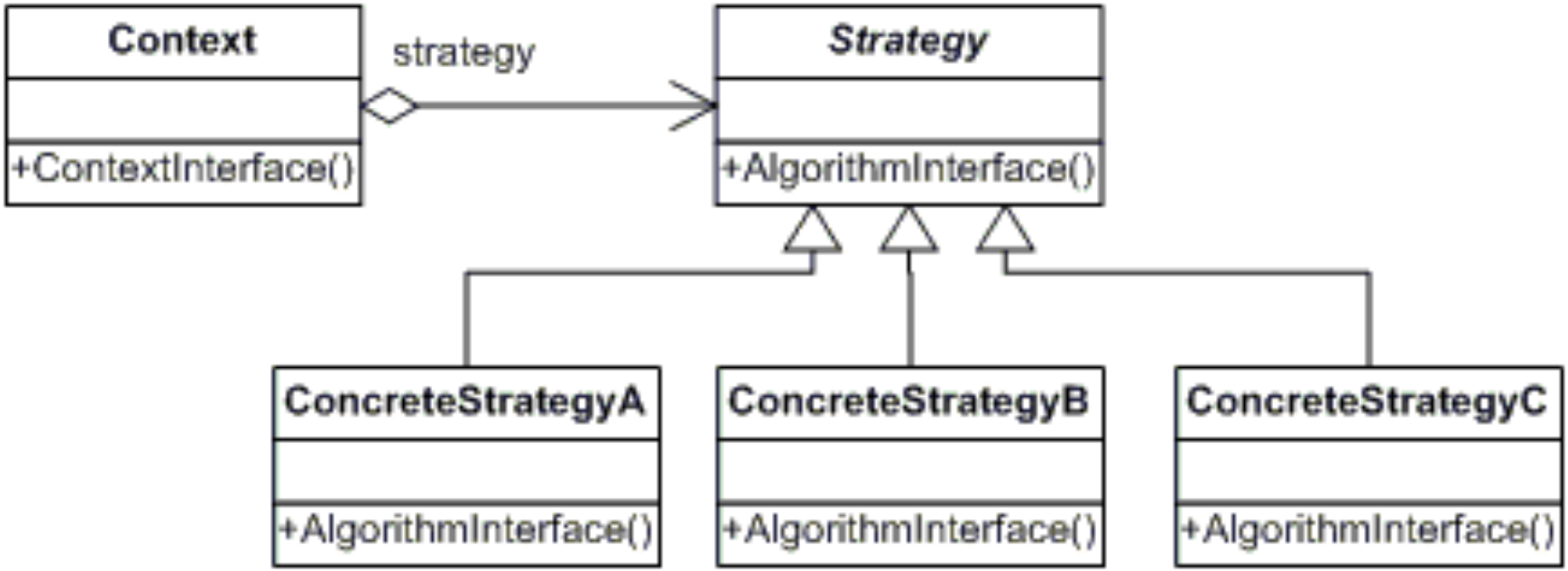
Comportamento dinamico



Strategy

- Utile dove è necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione
  - Mediante il pattern **Strategy** è possibile selezionare a runtime uno tra diversi algoritmi
- Questo pattern prevede che gli algoritmi siano intercambiabili tra loro (in base ad una qualche condizione) in modo trasparente da chi ne fa uso
  - La famiglia di algoritmi che implementa una funzionalità (ad esempio di visita o di ordinamento) esporta sempre la medesima interfaccia
  - Il cliente dell'algoritmo non deve fare nessuna assunzione su quale sia la strategia istanziata in un particolare istante

UML


Context
strategy
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Strategy per ordinamento

- Vogliamo ordinare un contenitore di oggetti (ad esempio un array)
  - Qualcosa come:

```
public static void sort(Object [] s){…}
```

- Come definire una relazione di ordinamento su un dato definito dall'utente?
  - Object non ha un metodo per il confronto e quindi occorre definirlo da qualche altra parte
- Aggiungo come argomento al metodo un "oggettino" incaricato del confronto con un tipo Interfaccia
  - Uso l'interfaccia **Comparator** (in java.util), che definisce sintatticamente il confronto di due oggetti
  - Fornisco una implementazione di Comparator per il tipo che voglio ordinare (ad esempio IntegerComparator)
  - Passo anche un Comparator quando chiamo la procedura per confrontare gli elementi

Interface Comparator

```
public interface Comparator<T> {
    // se o1 e o2 non sono di tipi confrontabili
    // lancia ClassCastException
    // altrimenti: o1<o2 -> ret -1
    //             o1==o2 -> ret 0
    //             o1>o2 -> ret 1

    public int compare(T o1, T o2) throws
          ClassCastException, NullPointerException;
}
```

Come usarla

- Un oggetto di tipo Comparator
  - Uno solo per tutti gli elementi!

```
public static void sort (String[] s, Comparator<String> c) {
    ...
    if  (c.compare(s[i], s[i+1])==0)
    ...
}

public static void main(String[] args) {
    String[] s = new String[30];
    sort(s, new AlphaComparator());
}
```

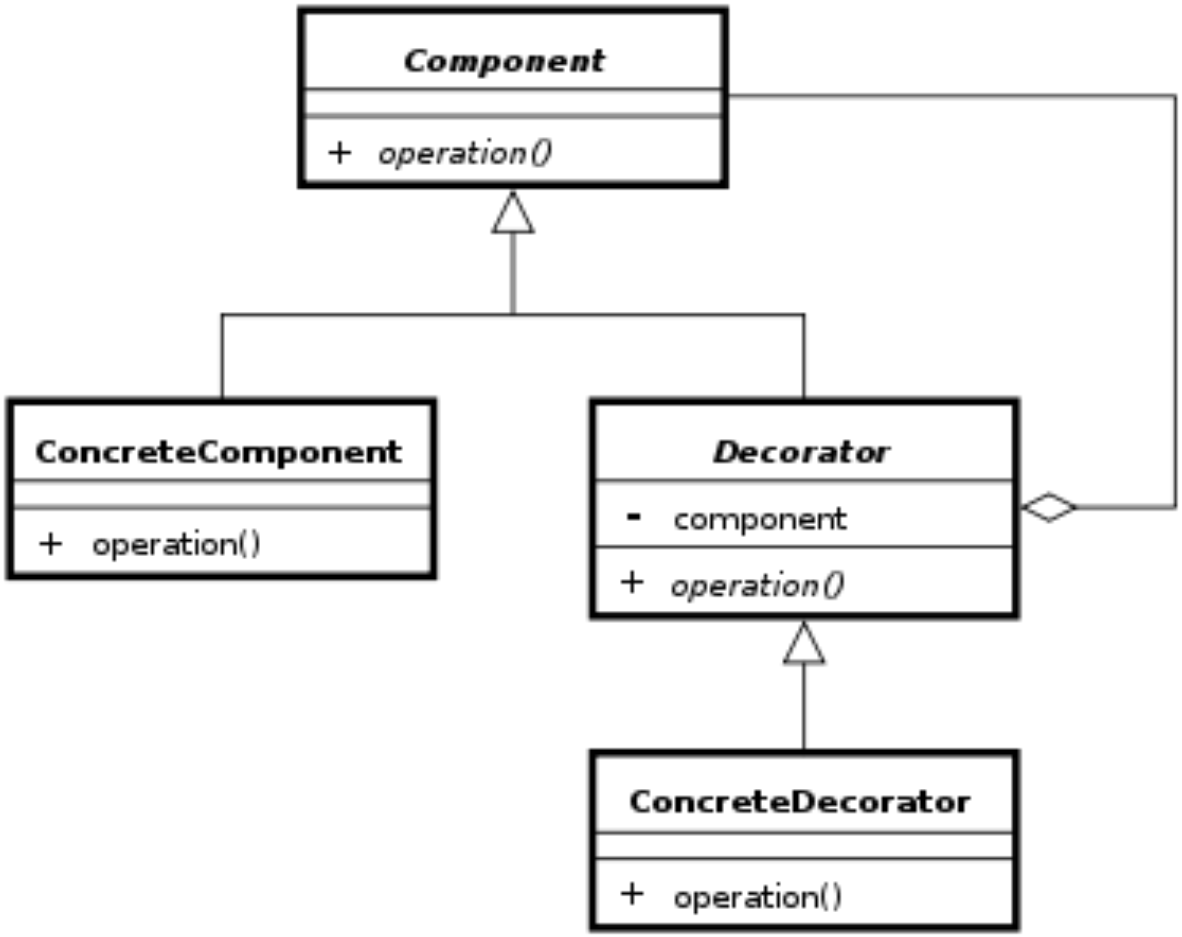
Come implementarla

```
import java.util.Comparator;

public class AlphaComparator implements Comparator<String>
{
   public int compare(String s1, String s2) {
      return s1.toLowerCase().compareTo(s2.toLowerCase());
   }
}
```

Decorator

- Questo pattern consente di aggiungere nuove funzionalità ad oggetti già esistenti a runtime
- Questo viene realizzato costruendo una nuova classe decoratore che "avvolge" l'oggetto originale
  - Ciò viene realizzato passando l'oggetto originale come parametro al costruttore del decoratore
- Questo pattern offre un'alternativa alle sottoclassi
  - Il decorator permette di aggiungere nuove funzionalità in un secondo momento e solo per determinati oggetti

UML


Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

Uso del Decorator

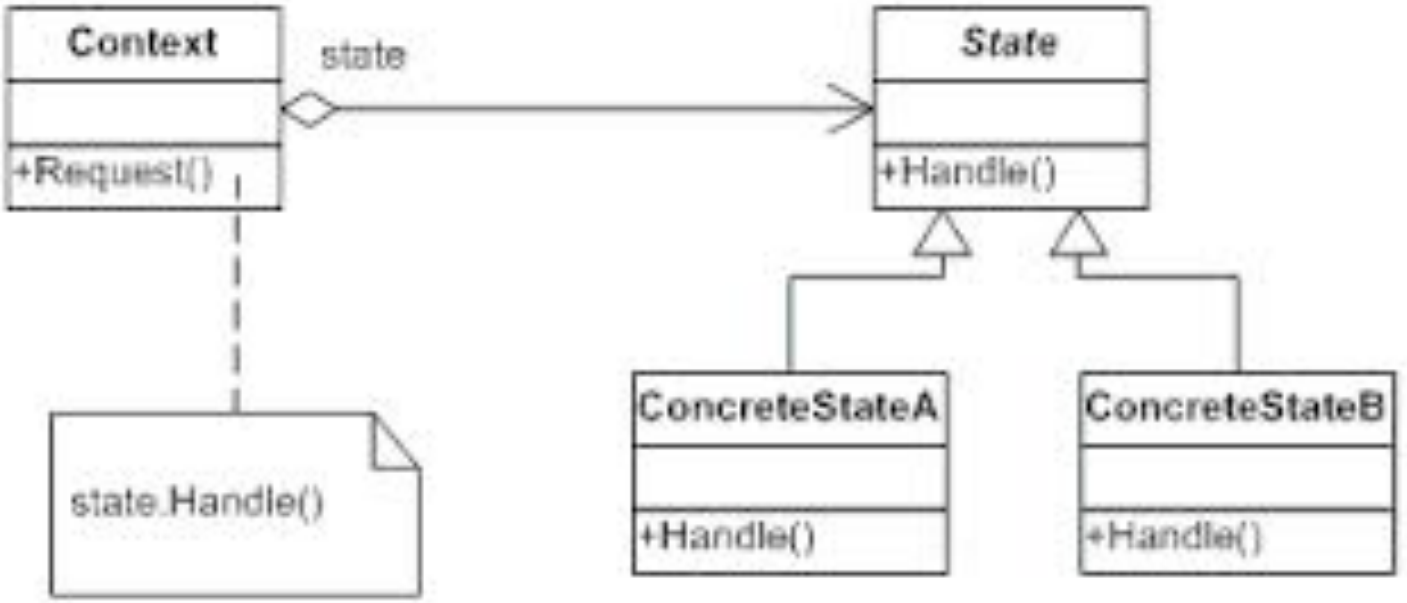
- I decoratori forniscono un'alternativa flessibile alle sottoclassi per estendere le funzionalità di una classe
- A differenza della definizione di sottoclassi, consente di estendere anche a runtime funzionalità di un oggetto esistente

Il pattern State

- Nel caso di classi mutabili, si può pensare a implementazioni multiple, i cui oggetti cambiano dinamicamente configurazione a seconda dello stato
  - Le implementazioni multiple corrispondono a diversi stati in cui possono trovarsi gli esemplari del tipo astratto
  - Nel corso della vita dell'oggetto, possono essere utilizzate diverse implementazioni senza che l'utente se ne accorga

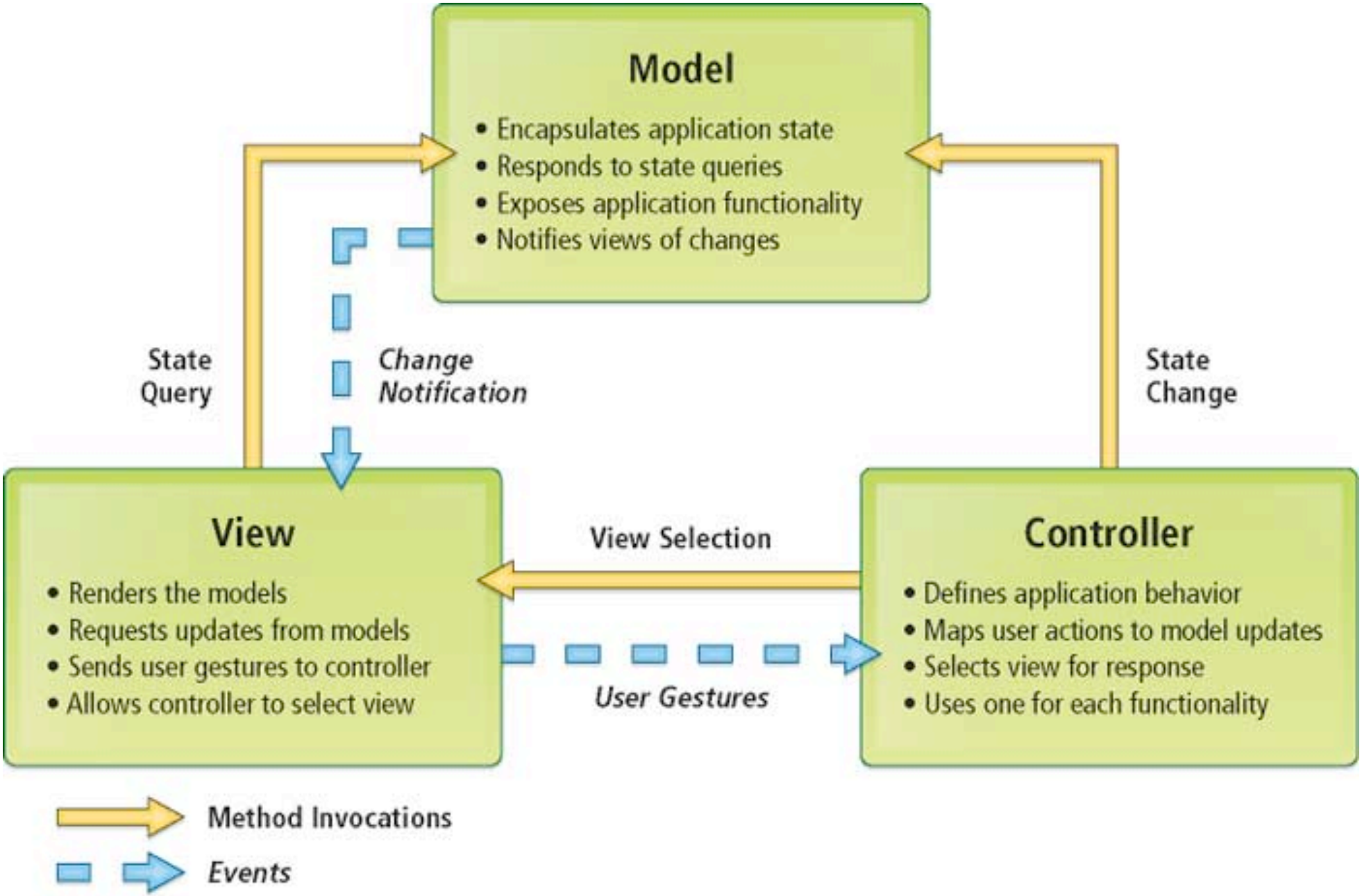
- Esempio: Una classe IntSet, insieme di interi, con due implementazioni
  - ArrayList, valida per insiemi di interi qualunque
  - BitSet, quando i valori sono piccoli

UML


Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Model-View-Controller

- Il pattern è basato su tre ruoli principali:
  - Il **Model** fornisce i metodi per accedere ai dati utili all'applicazione
  - Il **View** visualizza i dati contenuti nel **Model** e si occupa dell'interazione con utenti e agenti
  - Il **Controller** riceve i comandi dell'utente (in genere attraverso il View) e li attua modificando lo stato degli altri due componenti


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• Uses one for each functionality
Method Invocations
Events

Conclusioni

- I pattern forniscono un vocabolario comune tra i progettisti che facilita la comprensione di un progetto esistente o lo sviluppo di uno nuovo
- I pattern migliorano le prestazioni del codice e/o lo rendono più flessibile
- Il codice che utilizza i pattern potrebbe risultare più complesso del necessario
  - Occorre quindi valutare e confrontare costi e benefici

KEEP IT
SIMPLE