Programmazione Orientata agli Oggetti in Linguaggio Java

Design Pattern:
Pattern nelle API di Java

versione 1.2

Questo lavoro è concesso in uso secondo i termini di una licenza Creative Commons
(vedi ultima pagina)

SOME RIGHTS RESERVED
creative commons

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

Introduzione

- Le API di java
  - ⇨sono frequentemente basate sull'uso dei pattern
- Infatti
  - ⇨quando Java ha raggiunto la maturità (Java 1.2, 1998), i design pattern della GoF erano già noti
  - ⇨nel seguito vediamo alcuni esempi

Stringhe e Flyweight

- Un primo esempio
  - ⇨le stringhe di Java
- Semantica poco intuitiva
  - ⇨prima ragione: sono immutabili
  - ⇨seconda ragione: l'uguaglianza (a volte == restituisce il risultato corretto, a volte no)
- La ragione
  - ⇨le stringhe sono in effetti dei Flyweight

Stringhe e Flyweight

- Uno degli obiettivi dei design pattern
  - ⇨limitare il numero di oggetti da creare
- Una delle tecniche per farlo
  - ⇨condividere oggetti esistenti, quando questo è possibile; un esempio: i singleton
- Nel caso delle stringhe
  - ⇨questo ha senso; se uso in un punto del programma la stringa “lezione”, con tutta probabilità la userò anche in altri punti

Stringhe e Flyweight

- Le stringhe, però
  - ⇨non possono essere singleton
  - ⇨sono normalmente necessarie molte stringhe (molti oggetti di tipo String) in un’applicazione
  - ⇨di conseguenza ci vuole una soluzione diversa
- Idea
  - ⇨creare un unico oggetto per ogni stringa distinta usata nel programma

Stringhe e Flyweight

- Semantica delle stringhe in Java
  - ⇨sono oggetti immutabili, in modo da poter essere condivisi in punti diversi dell'applicazione senza problemi
  - ⇨inoltre, la macchina virtuale attribuisce una semantica speciale all'inizializzazione
  - ⇨questa semantica è basata sull'utilizzo di un "registro" delle stringhe esistenti nel programma

Stringhe e Flyweight

- In particolare
  - ⇨la macchina virtuale tiene traccia in una apposita zona di memoria di tutti i riferimenti agli oggetti di tipo String utilizzati
  - ⇨ogni volta che incontra un'istruzione del tipo:
    String s = "valore";
    verifica se "valore" è una stringa già usata; se così è preleva il riferimento dal registro
  - ⇨altrimenti crea un nuovo oggetto e lo aggiunge al registro

Stringhe e Flyweight

- Esempio

```
public class ProvaStringhe {
   void static void main(String args[]) {
      String s1 = “prova”;          // s1 vale #235
      String s2 = “altra prova”;    // s2 vale #7865
      String s3 = “prova”;          // s3 vale #235
      if (s1 == s3) {               // vero
         System.out.println(“s1 ed s2 uguali”);
      }
   }
}
```

viene chiesto di utilizzare la stringa “prova”;
il registro è inizialmente vuoto;
viene creato l’oggetto #235

viene chiesto di utilizzare la stringa “altra prova”;
non è presente nel registro;
viene creato l’oggetto #7865

viene chiesto di ri-utilizzare la stringa “prova”;
la stringa è presente nel registro;
viene restituito il valore #235

Registro delle Stringhe

valore	riferimento
“prova”	#235
“altra prova”	#7865

Stringhe e Flyweight

- Nota
  - ⇨questo meccanismo vale solo nel caso di assegnazione di stringa costante
  - ⇨non vale, viceversa, nel caso venga esplicitamente chiamato il costruttore
- Esempio

```
String s1 = “prova”;
String s3 = new String(“prova”);
In questo caso: s1 == s3 è falso , s1.equals(s3) è vero
```

Stringhe e Flyweight

- Attenzione
  - ⇨questo meccanismo viene discusso solo per introdurre la semantica dei flyweight
  - ⇨NON è opportuno utilizzare l'operatore di confronto == sulle stringhe
  - ⇨è opportuno usare sempre il metodo equals()

Nome: Flyweight
Categoria: strutturale
Difficoltà di apprendimento media
Difficoltà di applicazione media

Stringhe e Flyweight

- Il pattern
  - ⇨Flyweight
- Descrizione
  - ⇨consente di condividere un gruppo ("pool") di istanze di una certa classe tra diversi client
  - ⇨le istanze sono contenute in un registro
  - ⇨a ciascuna è assegnato un nome che consenta di recuperarle

Stringhe e Flyweight

- Attenzione
  - ⇨nel caso delle stringhe di Java la gestione del registro è a carico della macchina virtuale
- Viceversa
  - ⇨in generale, per utilizzare il pattern è necessario programmare la gestione del registro
  - ⇨tipicamente questo viene fatto utilizzando un dizionario associativo (mappa)

Stringhe e Flyweight

- Strategia di implementazione
  - ⇨sono coinvolte due classi
  - ⇨la classe sulla base della quale vengono costruiti i Flyweight
  - ⇨una ulteriore classe che crea i flyweight e gestisce il registro
  - ⇨tipicamente questa ulteriore classe è un singleton

Stringhe e Flyweight

- Un esempio
  - ⇨le posizioni sulla scacchiera in volpi e conigli
- La classe Posizione
  - ⇨crea oggetti che rappresentano coppie riga-colonna sulla scacchiera
  - ⇨utile per dimezzare il numero di parametri da passare ai metodi
  - ⇨è un candidato ideale a diventare un flyweight

```
public class PosizioneFlyweight {
   private int riga;
   private int colonna;

   public PosizioneFlyweight(int riga, int colonna) {
      if (riga < 0 || colonna < 0) {
         throw new IllegalArgumentException("Valori scorretti ");
      }
      this.riga = riga;
      this.colonna = colonna;
   }

   public int getRiga() { return riga; }

   public int getColonna() { return colonna; }

   public String toString() { return "[" + this.riga + ", " + this.colonna + "]"; }

}
```

```
package it.unibas.volpieconigli2.modello;

public class PosizioneFlyweightFactory {

    private static PosizioneFlyweightFactory singleton =
                                    new PosizioneFlyweightFactory();
    private PosizioneFlyweightFactory() {}

    public static PosizioneFlyweightFactory getInstance() {
        return singleton;
    }

    private java.util.HashMap cache = new java.util.HashMap();

    public PosizioneFlyweight getPosizione(int i, int j) {
        String chiave = i + "-" + j;
        PosizioneFlyweight posizione = (PosizioneFlyweight)cache.get(chiave);
        if (posizione == null) {
            posizione = new PosizioneFlyweight(i, j);
            cache.put(chiave, posizione);
        }
        return posizione;
    }
}
```

si tratta di un singleton

rappresenta il registro

gestisce il registro

Stringhe e Flyweight

- Esempio di utilizzo
  - ⇨il metodo cercaAnimale() di scacchiera

```
public Posizione cercaAnimale(Animale animale) {
    for (int i = 0; i < this.getNumeroRighe(); i++) {
        for (int j = 0; j < this.getNumeroColonne(); j++) {
            if (elementi[i][j] == animale) {
                return PosizioneFlyweightFactory.getInstance().getPosizione(i, j);
            }
        }
    }
    return null;
}
```

Stringhe e Flyweight

- Nota
  - ⇨la classe PosizioneFlyweightFactory è un esempio di classe “fabbrica” (factory)
  - ⇨genera un componente singleton la cui unica responsabilità è fabbricare e servire agli altri componenti altri oggetti
  - ⇨attorno a questo meccanismo è costruito un ulteriore pattern (“factory method”) che vedremo successivamente

Stringhe e Flyweight

- In generale
  - ⇨il pattern flyweight è molto importante per ridurre il numero di oggetti
  - ⇨consente di condividere un “pool” di oggetti
  - ⇨purchè questi siano immutabili
- Altri esempi di flyweight
  - ⇨i segni in un’applicazione sul totocalcio

Stringhe e Flyweight

- Un altro esempio
  - ⇨i caratteri in un editor di testi
  - ⇨ciascun carattere deve essere un oggetto grafico (o “glifo”) che sappia disegnarsi sullo schermo (non basta un semplice char)
  - ⇨in questo caso è essenziale utilizzare flyweight
- Attenzione
  - ⇨come è possibile gestire il “formato” ?

Stringhe e Flyweight

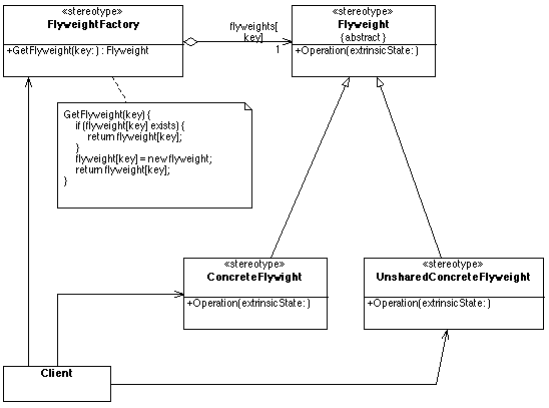
- In documenti diversi
  - ⇨uno stesso carattere può avere formati diversi (es: corsivo da una parte, grassetto dall’altra)
  - ⇨non è possibile associare direttamente il formato al flyweight (non sarebbe più immutabile)
- E’ un esempio di “stato estrinseco”
  - ⇨ovvero di stato da rappresentare all’esterno del flyw.
  - ⇨es: con oggetti di tipo “FormatoCarattere” per associare al flyweight lo stato di formato

Stringhe e Flyweight

- Stato estrinseco per il flyweight
  - ⇨oggetto associato ad un flyweight in un certo contesto (es: documento), che ne modifica il comportamento
  - ⇨i metodi del flyweight devono conoscere lo stato estrinseco
- Esempio: il metodo disegnati()
  - ⇨il metodo deve conoscere il formato per disegnare il carattere in modo opportuno

Stringhe e Flyweight

Flyweight - Diagramma UML della GoF



fonte: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

Collezioni ed Iteratori

- Un ulteriore principio dei pattern
  - ⇨programmare con le interfacce invece che con le implementazioni
  - ⇨le interfacce riducono l'accoppiamento tra le classi e semplificano i cambiamenti
- Un esempio: le liste
  - ⇨è opportuno utilizzarle per quanto possibile attraverso riferimenti di tipo java.util.List

Collezioni ed Iteratori

- Di conseguenza
  - ⇨si rende necessario un meccanismo efficiente per la scansione della collezione
  - ⇨ovvero un iteratore
- Idea
  - ⇨l'iteratore è un oggetto che consente di programmare la scansione in modo neutro rispetto all'implementazione
  - ⇨e implementa la scansione in modo ottimo

Nome: Iterator
Categoria: comportamentale
Difficoltà di apprendimento media
Difficoltà di applicazione media

Collezioni e Iteratori

- Si tratta in effetti di un pattern
  - ⇨Iterator
- Descrizione
  - ⇨utile quando è necessario scandire collezioni con uguale interfaccia ma implementazioni diverse
  - ⇨si tratta di un oggetto che sa come scandire una collezione senza mostrarne i dettagli implementativi

Collezioni e Iteratori

- L'utilizzo in java.util
  - ⇨interfaccia java.util.Iterator, che prevede i seguenti metodi
  - ⇨Object next() per spostarsi in avanti
  - ⇨boolean hasNext() per fermarsi
  - ⇨esiste poi una implementazione per ArrayList
  - ⇨ed una implementazione per LinkedList

Collezioni e Iteratori

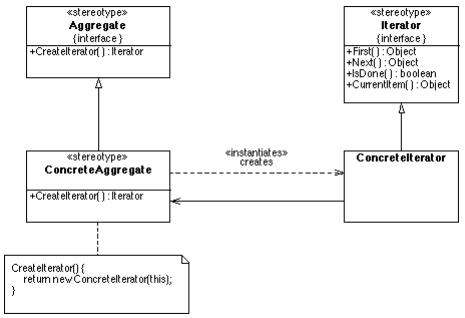
- In generale
  - ogni volta che in un'applicazione è necessario utilizzare collezioni con diverse implementazioni, sarebbe opportuno sviluppare un iteratore
  - e utilizzare sempre l'iteratore per la scansione per essere sicuri di adottare sempre la strategia migliore di scansione

Collezioni e Iteratori

- Un esempio: le matrici matematiche
  - una interfaccia Matrice
  - una implementazione basata su array bidimensionali per matrici piene
  - una implementazione basata su liste per matrici sparse
  - in questo caso sarebbe opportuno definire un iteratore per la scansione degli elementi non nulli

Collezioni e Iteratori

Iterator - Diagramma UML della GoF



fonte: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

Ordinamenti e Template Method

- Una postilla sull'ordinamento
  - ⇨l'interfaccia java.lang.Comparable corrisponde ad un pattern preciso
  - ⇨il pattern Template Method
- java.lang.Comparable
  - ⇨prevede un unico metodo:
  - ⇨int compareTo(Object o)
  - ⇨risultato positivo, negativo o nullo

Ordinamenti e Template Method

- Si tratta di un metodo “modello”
  - ⇨definisce una regola e tutte gli oggetti da ordinare devono implementarlo
- Idea
  - ⇨il metodo Collections.sort() è scritto assumendo che gli oggetti della lista implementino compareTo()
  - ⇨quindi è un metodo scritto assumendo il rispetto del modello stabilito

Ordinamenti e Template Method

- Nota
  - ⇨su questo pattern c’è una certa confusione terminologica
- Secondo alcuni
  - ⇨il template method sarebbe compareTo() – perchè fornisce un modello per le classi della gerarchia
- Secondo altri (e secongo la GoF)
  - ⇨il template method sarebbe sort() – che fornisce uno scheletro di algoritmo, ma assume l’implementazione di uno o più metodi nella gerarchia

Nome: Template Method
Categoria: comportamentale
Difficoltà di apprendimento limitata
Difficoltà di applicazione limitata

Ordinamenti e Template Me

- Pattern
  - ⇨Template Method
- Descrizione
  - ⇨consiste nello stabilire un modello di metodo
  - ⇨che tutte le classi di una certa gerarchia devono rispettare (implementare)
  - ⇨per poter essere utilizzate nel contesto richiesto

Ordinamenti e Template Method

- E' un pattern diffusissimo
  - ⇨utilizzato praticamente da tutti i framework
  - ⇨è uno degli strumenti attraverso i quali si stabiliscono le regole di funzionamento del framework
- Esempio: JUnit
  - ⇨il metodo runBare() di TestCase
  - ⇨e i metodi setUp() e tearDown()

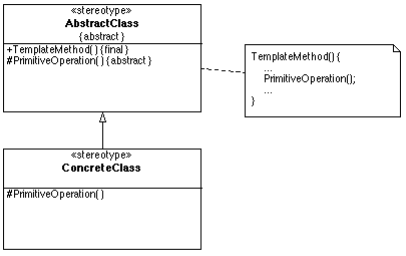
```
public void runBare()
      throws Throwable {
    setUp();
    try { runTest(); }
    finally { tearDown(); }
}
```

Ordinamenti e Template Method

- In effetti
  - ⇨è usato quasi sempre nel caso di ereditarietà di tipo
- Un ulteriore esempio
  - ⇨volpi e conigli: il metodo agisci() dell'interfaccia animale
  - ⇨il metodo simula() assume che tutti gli animali implementino agisci()

Ordinamenti e Template Method

Template Method
Diagramma UML della GoF



fonte: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

Flussi, Decoratori e Adattatori

- Un ultimo esempio rilevante
  - ⇨il package java.io
  - ⇨utilizza estensivamente due altri pattern
- Il pattern Decorator
  - ⇨per ridurre il numero di classi
- Il pattern Adapter
  - ⇨per gestire trasformare flussi binari in flussi testuali

Il Pattern Decorator

- Il problema dei flussi
  - ⇨esistono molti tipi diversi di flussi
- Esempio
  - ⇨flussi di lettura e flussi di scrittura
  - ⇨flussi orientati ai byte (binari) e ai caratteri
  - ⇨flussi da e verso i file e non
  - ⇨flussi tamponati e non tamponati

Il Pattern Decorator

- In effetti
  - ⇨esistono molti altri tipi di flussi
- Esempio: flussi filtrati
  - ⇨flussi in cui è possibile filtrare i byte o i caratteri, per esempio per sostituirli o elimin.
- Esempio: flussi di “pushback”
  - ⇨flussi in cui, dopo aver prelevato byte o caratteri, è possibile rimetterli a posto

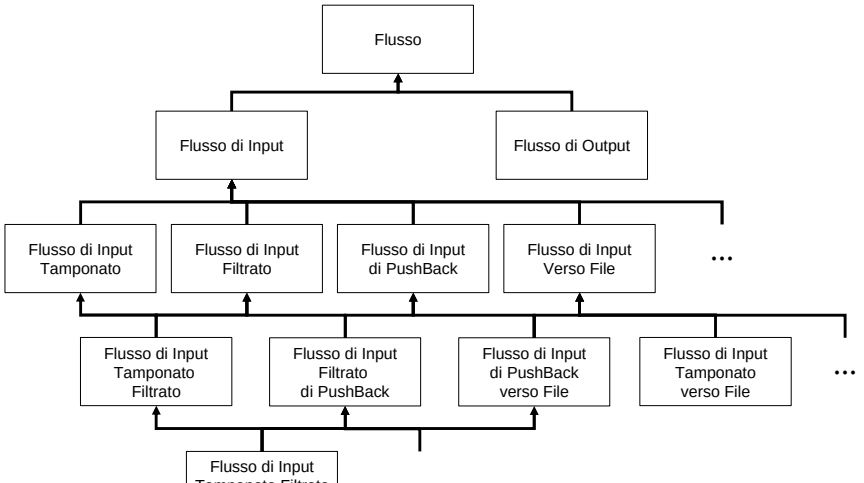
Il Pattern Decorator

- E ancora
  - ⇨ne esistono anche di altri tipi “strani”; es: flussi con conteggio dei numeri di linea
  - ⇨inoltre un flusso può avere più caratteristiche (es: flusso di ingresso binario filtrato e tamponato)
- Un possibile approccio
  - ⇨una gerarchia con una classe per ogni tipo di flusso

Il Pattern Decorator

- Primo problema
  - ⇨le classi della gerarchia tendono a diventare moltissime
  - ⇨con n tipi di flussi posso avere fino a 2^n classi
  - ⇨in java.io si contano una dozzina di tipologie di flussi ($2^{12} = 4096$)
- Secondo problema
  - ⇨sarebbe necessaria l'ereditarietà multipla

Il Pattern Decorator



Il Pattern Decorator

- Due principi dei design pattern
  - favorire l'associazione rispetto all'estensione
  - favorire la manutenibilità del codice
- Una possibile soluzione
  - fornire una collezione di classi "componibili"
  - ciascuna delle quali è in grado di aggiungere una tipologia di funzionalità ad un flusso esistente
  - in modo da poter comporre il flusso voluto

Nome: Decorator
Categoria: strutturale
Difficoltà di apprendimento media
Difficoltà di applicazione media

Il Pattern Decorator

- Il pattern
  - Decorator
- Idea
  - un decoratore è un oggetto che lavora in associazione con un altro oggetto
  - "decora" i metodi del primo oggetto aggiungendogli funzionalità
  - e può aggiungere ulteriori metodi
  - può essere composto con altri decoratori

Il Pattern Decorator

- La soluzione di java.io
  - ⇨consideriamo solo i flussi binari di ingresso
  - ⇨la classe di base: InputStream
- Decoratori disponibili
  - ⇨BufferedInputStream: rende lo stream tamponato
  - ⇨FileInputStream: rende lo stream indirizzato verso un flusso
  - ⇨FilterInputStream: rende lo stream filtrato

Il Pattern Decorator

- Come si costruisce un flusso decorato
  - ⇨si parte da un flusso di base
  - ⇨successivamente si costruiscono gli ulteriori flussi a partire da quello base

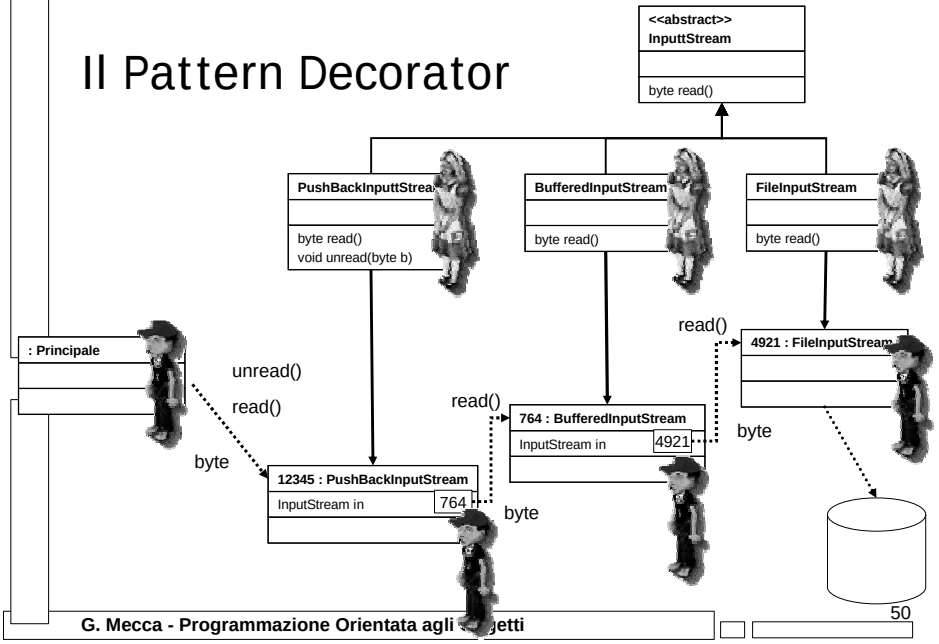
```
FileInputStream flussoBase = new FileInputStream("prova.txt")
BufferedInputStream flussoTamponato =
            new BufferedInputStream(flussoBase);
PushbackInputStream flussoTamponatoPBack =
            new PushbackInputStream(flussoTamponato);
```

ottengo un flusso binario di ingresso da file tamponato e di pushback

Il Pattern Decorator

- Cosa succede nel costruttore
  - ⇨il nuovo flusso costruito mantiene un riferimento al vecchio flusso
  - ⇨ogni volta che deve eseguire un metodo, sfrutta il metodo corrispondente del flusso originale, aggiungendo operazioni
  - ⇨può, inoltre, aggiungere metodi completamente nuovi

Il Pattern Decorator

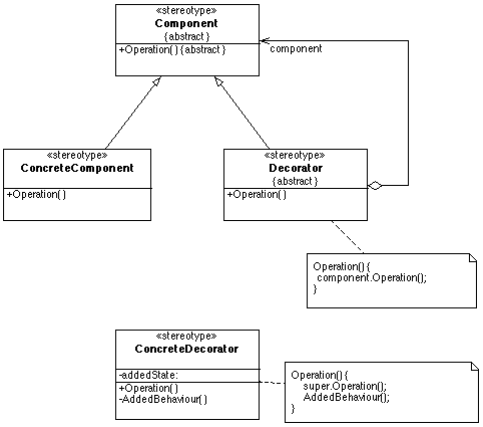


Il Pattern Decorator

- In generale
  - ⇨un oggetto decoratore viene creato a partire da un altro oggetto
  - ⇨implementa la stessa l'interfaccia dell'oggetto
  - ⇨in ciascun metodo utilizza il metodo corrispondente dell'oggetto originale, aggiungendo funzionalità
  - ⇨può, eventualmente, arricchire l'interfaccia con altri metodi

Il Pattern Decorator

Decorator - Diagramma UML della GoF



fonte: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

Il Pattern Adapter

- Una ulteriore caratteristica dei flussi
  - ⇨la versione originale di Java prevedeva solo flussi binari
  - ⇨basati sulle classi astratte InputStream e OutputStream
  - ⇨successivamente sono stati introdotti i flussi testuali compatibili con Unicode
  - ⇨basati sulle classi astratte Reader e Writer

Il Pattern Adapter

- Problema
  - ⇨era necessario poter trasformare i flussi standard (binari) in flussi del nuovo tipo
  - ⇨ma le due interfacce sono leggermente diverse
- Esempio: il metodo read() multicarattere
  - ⇨in InputStream: read(byte[] valori)
  - ⇨in Reader: read(char[] valori)
  - ⇨nel secondo caso i byte sono letti 2 a 2

Nome: Adapter
Categoria: strutturale
Difficoltà di apprendimento media
Difficoltà di applicazione media

Il Pattern Adapter

- Pattern
  - ⇨Adapter
- Descrizione
  - ⇨un oggetto costruito a partire da un altro oggetto
  - ⇨trasforma l'interfaccia dell'oggetto originale in una nuova interfaccia compatibile con il contesto in cui è usato

Il Pattern Adapter

- Esempio
  - ⇨InputStreamReader, un "adattatore" da InputStream a Reader
  - ⇨un oggetto che trasforma un flusso binario in un flusso testuale, adattandone l'interfaccia
- Esempio
  - ⇨InputStream stream = System.in;
  - ⇨Reader reader = new InputStreamReader(stream);

Il Pattern Adapter

- Attenzione alla differenza
  - ⇨un adattatore cambia l'interfaccia dell'oggetto
  - ⇨mentre un decoratore la duplica arricchendone l'implementazione
  - ⇨possono essere usati assieme
- Esempio: in Console

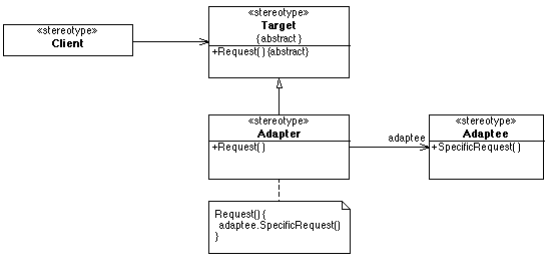
```
BufferedReader stdin =
        new BufferedReader(
                new InputStreamReader(System.in));
```

decorator

adapter

Il Pattern Adapter

Adapter
Diagramma UML della GoF



fonte: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

Riassumendo

- Creational patterns (5)
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton (discutibile, visto)
- Structural Patterns (7)
  - Adapter (visto)
  - Bridge (da rimuovere)
  - Composite
  - Decorator (visto)
  - Facade (visto)
  - Flyweight (visto)
  - Proxy
- Behavioral Patterns (11)
  - Chain of Responsibility (discutibile)
  - Command
  - Interpreter (da rimuovere)
  - Iterator (visto)
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method (visto)
  - Visitor
- Altri pattern
  - Null Object (visto)
  - DAO (visto)

Riassumendo

- Introduzione
- Stringhe e Flyweight
- Collezioni e Iteratori
- Ordinamenti e Template Method
- Flussi, Decoratori e Adattatori
  - Il Pattern Decorator
  - Il Pattern Adapter

Termini della Licenza

- This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/1.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

- Questo lavoro viene concesso in uso secondo i termini della licenza “Attribution-ShareAlike” di Creative Commons. Per ottenere una copia della licenza, è possibile visitare http://creativecommons.org/licenses/by-sa/1.0/ oppure inviare una lettera all’indirizzo Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.