Programmazione Java – Design Patterns

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

- Cosa è
- Template
- Catalogo
- Favorire composizione rispetto ereditarietà
- Patterns
  - Creazionali: Singleton, Abstract Factory
  - Strutturali: Decorator
  - Comportamentali: Observer
- Bibliografia

Cosa è: Cristopher Alexander

- Ogni pattern descrive
  - Un problema che si ripete più è più volte nel nostro ambiente
  - Il *core* di una soluzione al problema, dove tale soluzione può essere utilizzata un milione di volte senza mai applicarla nella stessa maniera
- Proviene dal mondo dell'architettura però patterns possono essere applicati a differenti aree tra cui lo sviluppo di software
- Ogni pattern è una regola di tre parti il quale esprime una relazione tra un certo contesto, un problema e una soluzione

Cosa è: GoF (1)

» Quattro elementi essenziali

- Nome
- Problema
- Soluzione
- Conseguenze

Cosa è: GoF (2)

» Nome

- Costituisce un *nome simbolico* per descrivere il pattern
- Aiuta la comprensione poiché permette di ragionare ad un più alto livello di astrazione
- Migliora la comunicazione tra sviluppatori poiché si ha un unico vocabolario

» Problema

- Spiega il problema e il contesto
- Descrive quando applicare il pattern
- Potrebbero descrivere classi o strutture di oggetti che sono sintomatiche di un design inflessibile
- Può includere lista di condizioni che devono essere rispettate per poter applicare il pattern

Cosa è: GoF (3)

» Soluzione

- Descrive gli elementi facenti parte del pattern, le relazioni, responsabilità e collaborazioni
- Non descrive una particolare soluzione poiché il pattern è un template che può essere applicato in differenti situazioni
- Fornisce una descrizione astratta degli elementi che costituiscono la soluzione e non un design o implementazione concreta

» Conseguenze

- Risultati e trade-off (pro e contro) nell'applicare il pattern
- Riguardano problemi con un linguaggio di programmazione o con l'implementazione
- Include eventuali impatti su affidabilità, portabilità, estendibilità

Benefici

- Cattura delle *expertise* e le rende accessibili anche ai non esperti
- Facilita la comunicazione tra sviluppatore fornendo un linguaggio comune
- Rende più facile il riuso di design di successo ed elimina alternative che diminuiscono il riuso
- Facilita modifiche al design
- Migliora la documentazione e la comprensione del design

Storia

» 1987 - Cunningham e Beck utilizzarono le idee di Alexander per sviluppare un piccolo linguaggio di pattern per Smalltalk

» 1990 - Gang of Four (Gamma, Helm, Johnson e Vlissides) iniziano a realizzare un catalogo di design pattern

» 1991 - Bruce Anderson a OOPSLA mostra i primi patterns

» 1993 - Kent Beck e Grady Booch sponsorizzano il primo meeting che è conosciuto come Hillside Group

» 1994 – Conferenze First Pattern Languages of Programs (PLoP)

» 1995 - Gang of Four (GoF) pubblicano il libro *Design Patterns*

Template (1)

- Nome pattern e Classificazione
  - Buon e nome conciso è vitale
- Intento
  - Breve frase per mostrare ciò che fa il pattern
  - Quale è il suo fondamento logico e intento?
- Detto anche….
  - Altri nomi per il pattern
- Motivazione
  - Scenario che illustra un problema di progettazione e il modo in cui la struttura di classi e oggetti definita nel pattern lo risolve
- Applicabilità
  - Situazioni dove il pattern può essere utilizzato

Template (2)

- Struttura
  - Rappresentazione grafica del pattern
- Partecipanti
  - Classi e oggetti che fanno parte del design e le loro responsabilità
- Collaborazioni
  - Come collaborano i partecipanti per potersi assumere le loro responsabilità
- Conseguenze
  - Come fa il pattern a raggiungere i propri obiettivi
  - Pro e contro nell'utilizzare il pattern

Template (3)

» Implementazione

- Suggerimenti e tecniche per implementare il pattern
- Problemi specifici connessi a un particolare linguaggio di programmazione

» Codice di esempio

- Frammenti di codice in C++ o Smalltalk (slide in Java)

» Usi conosciuti

- Utilizzo di pattern in sistemi reali

» Pattern correlati

- Altri pattern che sono in relazione

Catalogo (1)

		Scopo		
		Creational	**Structural**	**Behavioral**
Raggio d'azione	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Catalogo (2)

» Scopo

- Cosa fa il pattern
- **Creational**
  - Riguarda processo di creazione di oggetti
- **Structural**
  - Focalizza attenzione su composizione di classi e oggetti
- **Behavioral**
  - Caratterizzano il modo nel quale classi o oggetti interagiscono e distribuiscono responsabilità

Catalogo (3)

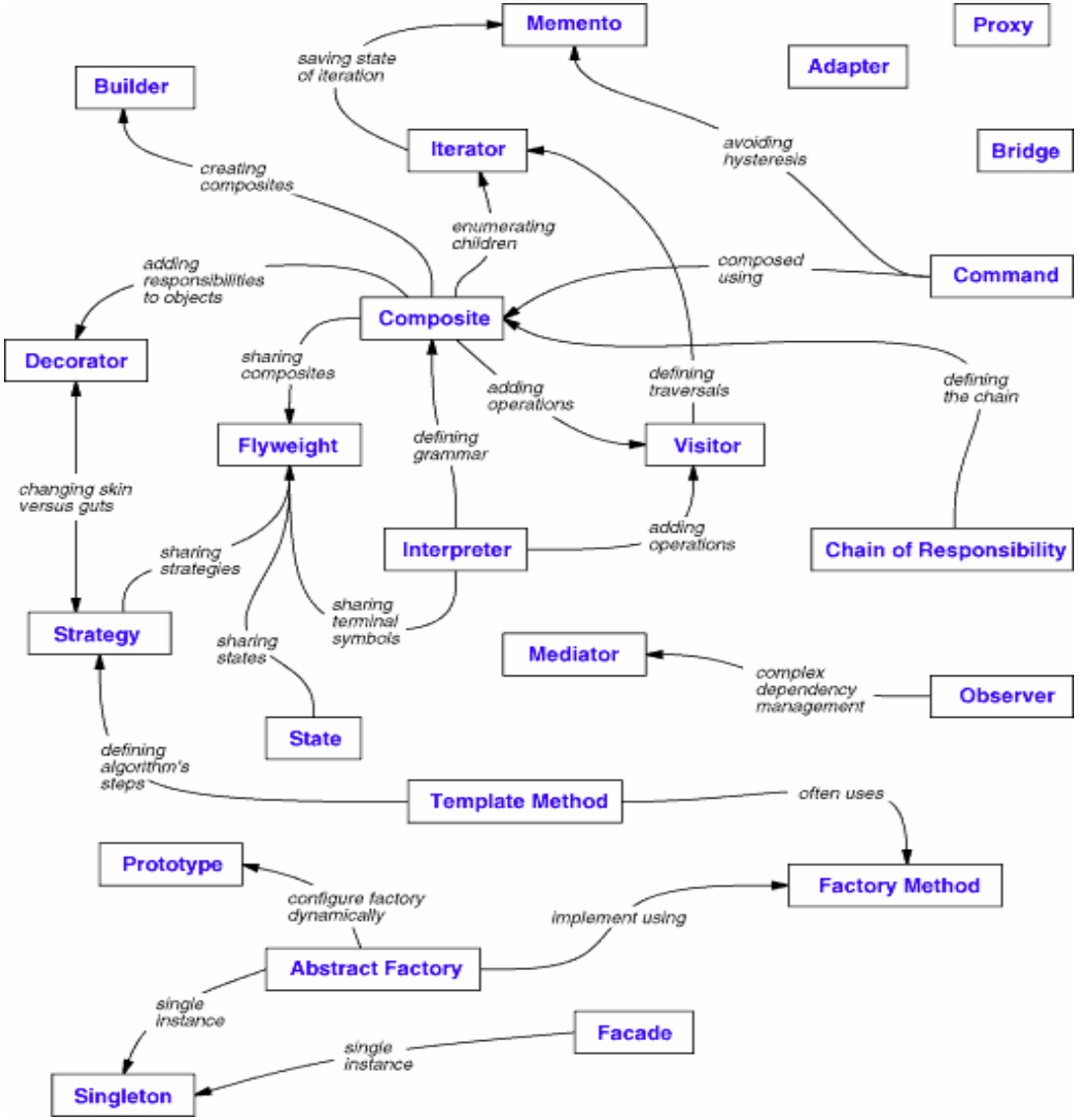
» Raggio di azione

- Pattern applicato a classi o ad oggetti
- **Class**
  - Considera relazioni tra classi e loro sottoclassi stabilite attraverso ereditarietà (statica)
- **Object**
  - Considera relazioni tra oggetti che sono modificate a run-time e sono più dinamiche

Catalogo (4)

- **Creational class** patterns delegano parte del processo di creazione di un oggetto a sottoclassi, mentre quelli **object** lo delegano ad altri oggetti
- **Structural class** patterns utilizzano l'ereditarietà per comporre classi, mentre quelli **object** descrivono modi per raggruppare oggetti
- **Behavioral class** patterns utilizzano ereditarietà per descrivere algoritmi e flusso di controllo, mentre quelli **object** descrivono come gruppi di oggetti cooperano per eseguire un compito che un singolo oggetto non potrebbe portare a termine da solo

Catalogo (5)


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Favorire composizione rispetto ereditarietà (1)

» Tecniche più comuni per il riuso nel paradigma OO sono

- Ereditarietà di classi
  - Definiamo un oggetto in termini di un altro
  - Riuso *white-box* ovvero visibilità della classi antenate è visibile alle sottoclassi
- Composizione di oggetti
  - Funzionalità sono ottenute assemblando o componendo gli oggetti per avere funzionalità più complesse
  - Riuso *black-box* poiché dettagli interni non sono conosciuti

Favorire composizione rispetto ereditarietà (2)

» Vantaggi ereditarietà

- Definita staticamente ed è immediata da utilizzare poiché supportata dal linguaggio
- Rende più facile la modifica dell'implementazione del padre

» Svantaggi ereditarietà

- Essendo statica non è possibile cambiarla e tempo di esecuzione (minore flessibilità)
- *Rompe il principio dell'incapsulamento*

Favorire composizione rispetto ereditarietà (3)


```
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public InstrumentedHashSet(int initCap,float loadFactor) {
        super(initCap, loadFactor);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
```

……

Favorire composizione rispetto ereditarietà (4)

.....

```
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    public static void main(String[] args) {
        InstrumentedHashSet s = new InstrumentedHashSet();
        s.addAll(Arrays.asList("Snap","Crackle","Pop"));
        System.out.println(s.getAddCount());
    }
}
```

Favorire composizione rispetto ereditarietà (5)

» Possibili soluzioni

- Eliminare override di `addAll`
  - Soluzione specifica di una certa release di java
- Modifica di implementazione ad `addAll`
  - Iterare sulla collection e invocare il metodo `add`
  - Reimplementiamo il metodo `addAll` nelle sottoclassi

Favorire composizione rispetto ereditarietà (6)


```
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

 public InstrumentedSet() {
        s = new HashSet();
    }
    public InstrumentedSet(Set s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
```

.....

Favorire composizione rispetto ereditarietà (7)


```
    public int getAddCount() {
        return addCount;
    }

    // Forwarding methods
    public void clear()             { s.clear(); }
    public boolean contains(Object o) {
        return s.contains(o);
    }
    public boolean isEmpty()        { return s.isEmpty(); }
    public int size()               { return s.size(); }
    public Iterator iterator()      { return s.iterator();}
    public boolean remove(Object o){ return s.remove(o);}
    public boolean containsAll(Collection c) {
      return s.containsAll(c);
    }
```

Favorire composizione rispetto ereditarietà (8)


```
.....
    public boolean removeAll(Collection c) {
     return s.removeAll(c);
    }
    public boolean retainAll(Collection c) {
     return s.retainAll(c);
    }
    public Object[] toArray() { return s.toArray();  }
    public Object[] toArray(Object[] a) {
     return s.toArray(a);
}
    public boolean equals(Object o){ return s.equals(o);  }
    public int hashCode() { return s.hashCode(); }
    public String toString() { return s.toString(); }
    public static void main(String[] args) {
         InstrumentedSet s = new InstrumentedSet();
         s.addAll(Arrays.asList("Snap","Crackle","Pop"));
         System.out.println(s.getAddCount());
    }
}
```

Pattern Creazionali

- Forniscono un'astrazione del processo di istanziazione degli oggetti e rendono il sistema indipendente da tale modalità
- Basati su *classi* utilizzano ereditarietà per scegliere la particolare classe da istanziare
- Basati su *oggetti* delegano l'istanziazione ad un altro oggetto
- Rendono il sistema maggiormente flessibile poiché conosce soltanto le interfacce degli oggetti definite mediante classi astratte

Singleton (1)

» Intento (Scopo)

- Assicurare che una classe abbia una sola istanza e fornire un punto d'accesso globale a tale istanza

» Motivazione

- Esempi
  - Diverse stampanti ma una sola coda di stampa
  - Unico window manager
- Come si assicura l'univocità dell'istanza?
  - Stessa classe ha la responsabilità di creare le istanze

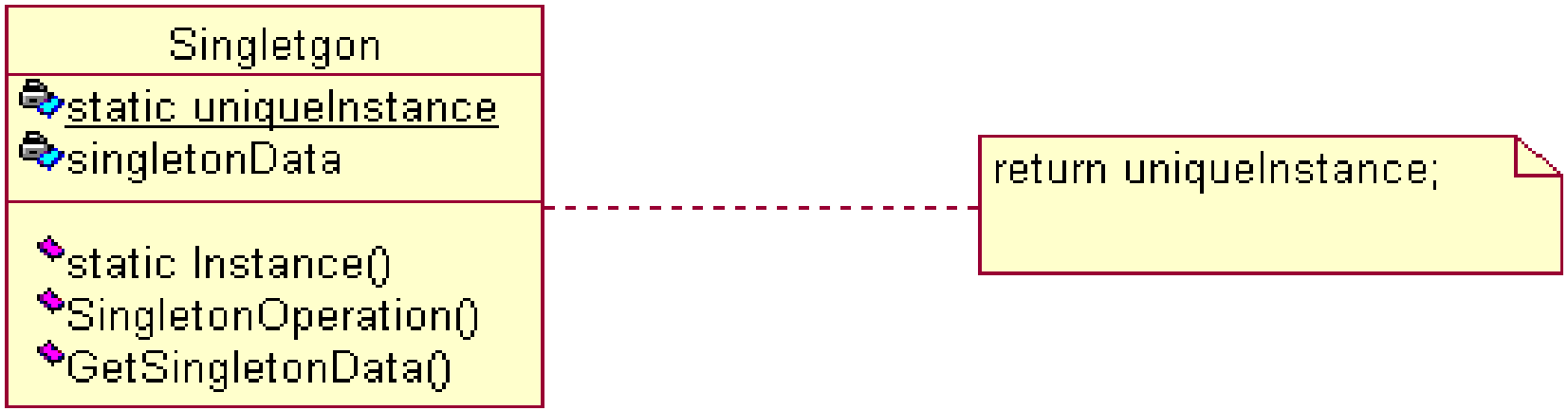
Singleton (2)

» Applicabilità

- Quando deve esistere esattamente un'istanza di una classe e tale istanza deve essere accessibile ai client attraverso un punto di accesso noto a tutti gli utilizzatori
- Quando l'unica istanza deve poter essere estesa attraverso la definizione di sottoclassi e i client devono essere in grado di utilizzare le istanze estese senza dover modificare il proprio codice

Singleton (3)

» Struttura



Singleton (4)

» Partecipanti

- **Singleton**
  - Definisce un'operazione *Instance* che consente ai client di accedere all'unica istanza esistente della classe
  - *Instance* deve essere un'operazione di classe
  - Può essere responsabile della creazione della sua unica istanza

» Collaborazioni

- Client possono accedere a un'istanza di un Singleton soltanto attraverso l'operazione *Instance*

Singleton (5)

» Conseguenze

- Accesso controllato a un’unica istanza
- Riduzione dello spazio dei nomi ovvero non è necessario definire variabili globali
- Permette il raffinamento di operazioni e rappresentazione ovvero è possibile definire delle sottoclassi che costituiscono l’unica istanza
- Permette di gestire un numero variabili di istanze
- Maggiore flessibilità rispetto a operazioni di classe

Singleton (6)

» Implementazione

- Assicurare l'esistenza di un'unica istanza
  - Costruttore privato
  - Variabile di classe privata
  - Metodo statico che restituisce la variabile di classe
- Definizione di sottoclassi di Singleton
  - Costruttore protetto
  - Metodo set oppure utilizzo di meccanismi globali per *ottenere* l'istanza della sottoclasse

Singleton: Soluzione 1 (7)


```
public final class Singleton {

    private static Singleton instance = new Singleton( 10 );

    private int singletonData;
    private Singleton( int data ) {
        singletonData = data;
    }

    public void singletonOperation() {
        singletonData += 10;
    }

    public int getSingletonData() {
        return singletonData;
    }

    public static final Singleton getInstance() {
        return instance;
    }
}
```

Singleton: Soluzione 1 (8)


```
public class TestSingleton {

   public static void main( String[] args ) {

     Singleton s = Singleton.getInstance();

     s.singletonOperation();
     System.out.println( "Prima reference: " + s );
     System.out.println( "Valore del singleton data: " +
                                          s.getSingletonData() );
     Singleton s1 = Singleton.getInstance();
     s1.singletonOperation();
     System.out.println( "\nSeconda reference: " + s1 );
     System.out.println( "Valore del singleton data di s1: " +
                                          s1.getSingletonData() );
     System.out.println( "Valore del singleton data di s: " +
                                          s.getSingletonData() );
  }
}
```

Singleton: Soluzione 1 (9)

OUTPUT

```
Primo reference: Singleton@182f0db
Valore del singleton data: 20

Seconda reference: Singleton@182f0db
Valore del singleton data di s1: 30
Valore del singleton data di s: 30
```

Singleton: Soluzione 2 (10)


```
public final class SingletonLazyInstantiation {
    private static SingletonLazyInstantiation instance;

    private int singletonData;
    private SingletonLazyInstantiation( int data ) {
        singletonData = data;
    }
    public void singletonOperation() {
        singletonData += 10;
    }
    public int getSingletonData() {
        return singletonData;
    }

    public static final synchronized SingletonLazyInstantiation getInstance() {
        if ( instance == null ) {
            instance = new SingletonLazyInstantiation( 10 );
        }
        return instance;
    }
}
```

Singleton: Soluzione 3 (11)


```
public abstract class BaseSingleton {

    private static BaseSingleton instance;

    private int singletonData;

    protected BaseSingleton() {
        singletonData = 10;
    }

    public void singletonOperation() {
        singletonData += 10;
    }

    public int getSingletonData() {
        return singletonData;
    }
```

.....

Singleton: Soluzione 3 (12)

.....

```
    public static final synchronized BaseSingleton getInstance() {
        if ( instance == null ) {
         throw
           new InstantiationError( "Singleton instance is null! " );
        }
        return instance;
    }

    public static final synchronized void setInstance(
                                        BaseSingleton singleton ) {
        if ( instance != null ) {
         throw
           new InstantiationError( "Singleton is already created" );
        }
        instance = singleton;
    }

}
```

Singleton: Soluzione 3 (13)


```
public class ExtendedSingleton extends BaseSingleton {

    public ExtendedSingleton() {
        super();
    }

    public void singletonOperation() {
        //Do nothing
    }
}
```

Singleton: Soluzione 3 (14)


```
public class TestSingleton3 {

    public static void main( String[] args ) {
        BaseSingleton.setInstance( new ExtendedSingleton() );
        BaseSingleton singleton = BaseSingleton.getInstance();
        System.out.println( "data: " + singleton.getSingletonData() );
    }
}
```

Singleton: esempio


```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();
    public static Runtime getRuntime() {
     return currentRuntime;
    }
    private Runtime() {}
    public void exit(int status) {
      ..............
    }
    public void addShutdownHook(Thread hook) {
      ..............
    }
    public boolean removeShutdownHook(Thread hook) {
      ....................
    }
    public void halt(int status) {
      ....................
    }
    public Process exec(String command) throws IOException {
      return exec(command, null, null);
    }
    public Process exec(String command, String[] envp) throws IOException {
        return exec(command, envp, null);
    }
    public Process exec(String command, String[] envp, File dir) throws IOException {
      .................
    }

    public native int availableProcessors();
    public native long freeMemory();
    public native long totalMemory();
    public native long maxMemory();
    public native void gc();
.....................................
}
```

Abstract Factory (1)

» Scopo

- Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete

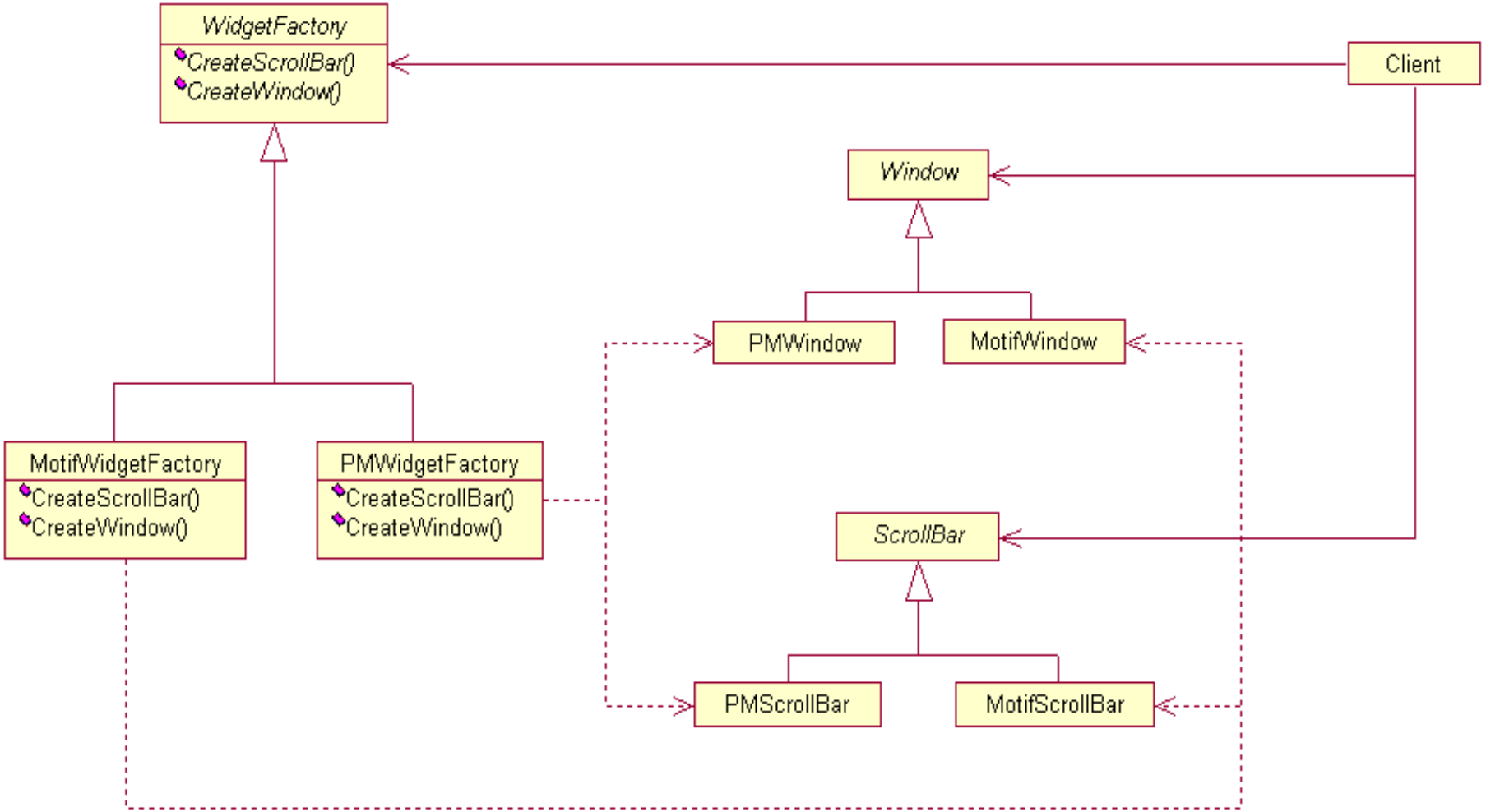
» Detto anche…

- Kit

» Motivazione

- GUI toolkit che supporta diversi standard di look-and-feel (motiv, presentation manager)
- Diverse modalità di presentazione e comportamento per gli elementi (*widget*)
- Per garantire portabilità gli elementi grafici di un look-and-feel non devono essere cablati nel codice

Abstract Factory (2)



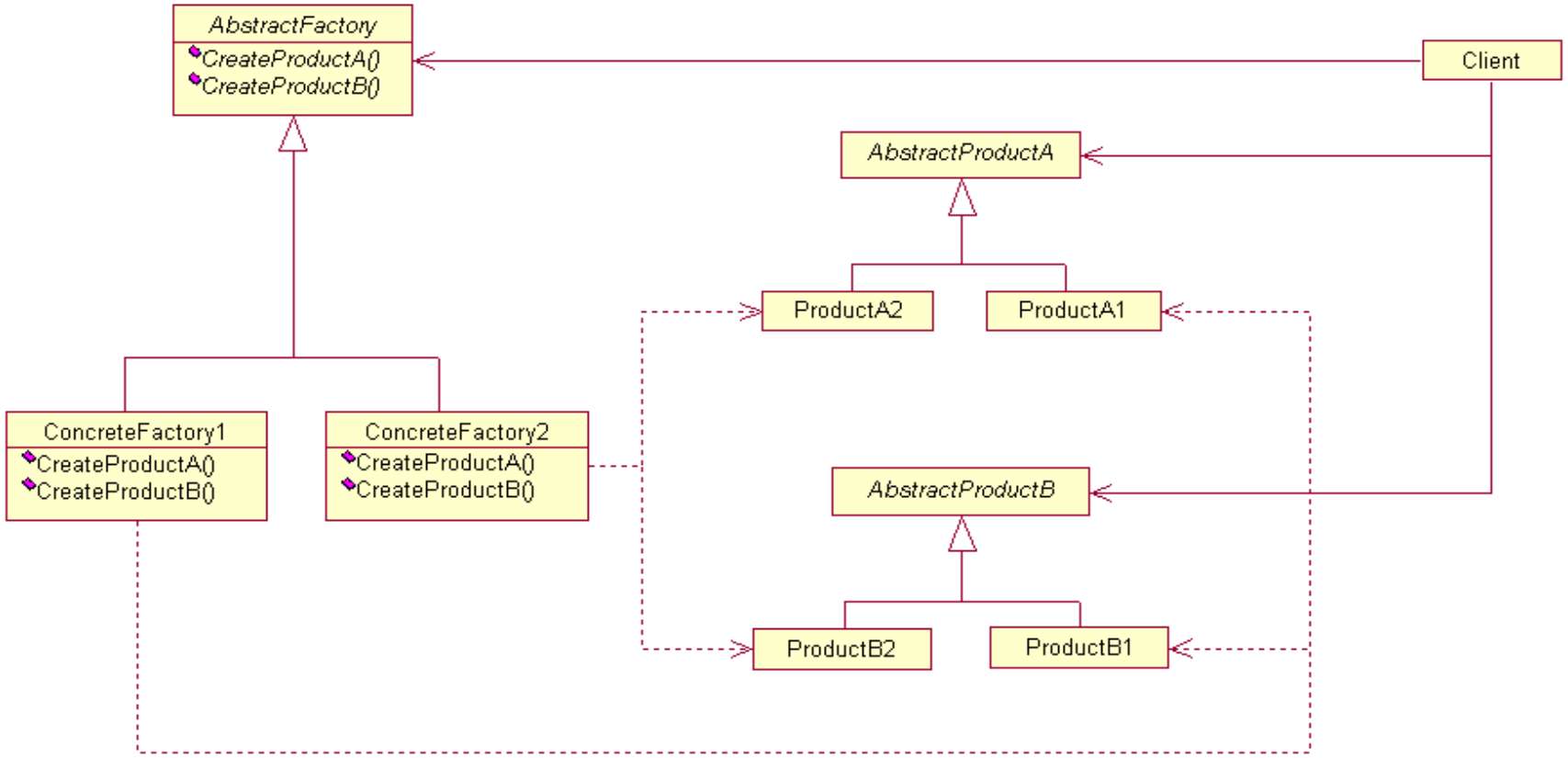
Abstract Factory (3)

» Applicabilità

- Sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti
- Sistema deve poter essere configurato scegliendo una tra più famiglie di prodotti
- Esistono diverse famiglie di prodotti che devono essere insieme
- Si vuole libreria di classi che espone soltanto l'interfaccia e non l'implementazione

Abstract Factory (4)

Struttura



Abstract Factory (5)

» Partecipanti

- **AbstractFactory** (WidgetFactory)
  - Dichiara un'interfaccia per le operazioni di creazione di oggetti prodotto astratti
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - Implementa le creazioni degli oggetti prodotto concreti
- **AbstractProduct** (Window, ScrollBar)
  - Dichiara un'interfaccia per una tipologia di oggetti prodotto
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - Definisce un oggetto prodotto che dovrà essere creato dalla corrispondente factory concreta
  - Implementa l'interfaccia AbstractProduct
- **Client**
  - Utilizza soltanto le interfacce dichiarate dalle classi AbstractFactory e AbstractProduct

Abstract Factory (6)

» Collaborazioni

- Generalmente si crea una singola istanza di una classe ConcreteFactory durante l'esecuzione
- Tale factory concreta gestisce la creazione di una famiglia di oggetti con un'implementazione specifica
- AbstractFactory delega la creazione di oggetti prodotto alle sue sottoclassi ConcreteFactory

» Conseguenze

- Isola classi concrete
- Consente di cambiare in modo semplice famiglia di prodotti utilizzata
- Promuove coerenza utilizzo di prodotti
- Aggiunta del supporto di nuovi tipologie di prodotti difficile

Abstract Factory (7)

» Implementazione

- Factory come *Singleton*
- Creazioni dei prodotti
  - Ogni prodotto viene creato mediante un factory method
    - Diverse sottoclassi per ogni famiglia di prodotti
  - Pattern *Prototype* elimina necessità di diverse sottoclassi concrete di factory
  - Un unico metodo che restituisce un solo tipo di prodotto (sconsigliato e utilizzato per particolari linguaggi)
- Definire factory estendibili ovvero un unico metodo con un parametro in ingresso che mi stabilisce il *tipo* del prodotto (meno sicuro)

Abstract Factory: Soluzione (8)


```
public abstract class WidgetFactory {

    private static WidgetFactory instance;

    protected WidgetFactory() {
    }

    public static final synchronized WidgetFactory getInstance() {
        if ( instance == null ) {
            throw new InstantiationError( "Error!" );
        }
        return instance;
    }
```

……

Abstract Factory (9)


```
.....
    public static final synchronized void setInstance(
            WidgetFactory widgetFactory ) {
        if ( instance != null ) {
            throw new InstantiationError( "Error!" );
        }

        instance = widgetFactory;

    }

    public abstract ScrollBar createScrollBar();

    public abstract Window createWindow();


}
```

Abstract Factory (10)


```
public abstract class Window {
    //Methods
}

public abstract class ScrollBar {
    //Methods
}
```

Abstract Factory (11)


```
public class MotifWidgetFactory extends WidgetFactory {

    public ScrollBar createScrollBar() {
        return new MotifScrollBar();
    }

    public Window createWindow() {
        return new MotifWindow();
    }
}

public class MotifWindow extends Window {
    //Methods
}

public class MotifScrollBar extends ScrollBar {
    //Methods
}
```

Abstract Factory (12)


```
public class PMWidgetFactory extends WidgetFactory {

    public ScrollBar createScrollBar() {
        return new PMScrollBar();
    }

    public Window createWindow() {
        return new PMWindow();
    }
}

public class PMWindow extends Window {
    //Methods
}

public class PMScrollBar extends ScrollBar {
    //Methods
}
```

Abstract Factory (13)


```
public class TestClient {
    public static void main( String[] args ) {

        WidgetFactory.setInstance( new PMWidgetFactory() );
        WidgetFactory factory = WidgetFactory.getInstance();
        ScrollBar scrollBar = factory.createScrollBar();

    }
}
```

Pattern Strutturali

- Si occupano della modalità di composizione di classi e oggetti per formare strutture complesse
- Basati su *classi*
  - Utilizzano l'ereditarietà per comporre interfacce o implementazioni
  - *Esempio:* ereditarietà multipla combina due o più classi per ottenere una classe con tutte le proprietà delle superclassi
- Basati su *oggetti*
  - Descrivono modalità di composizione di oggetti per realizzare nuove funzionalità
  - Hanno maggiore flessibilità poiché è possibile cambiare la composizione durante l'esecuzione

Decorator (1)

» Scopo

- Aggiungere dinamicamente responsabilità ad un oggetto. I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità

» Detto anche…

- Wrapper

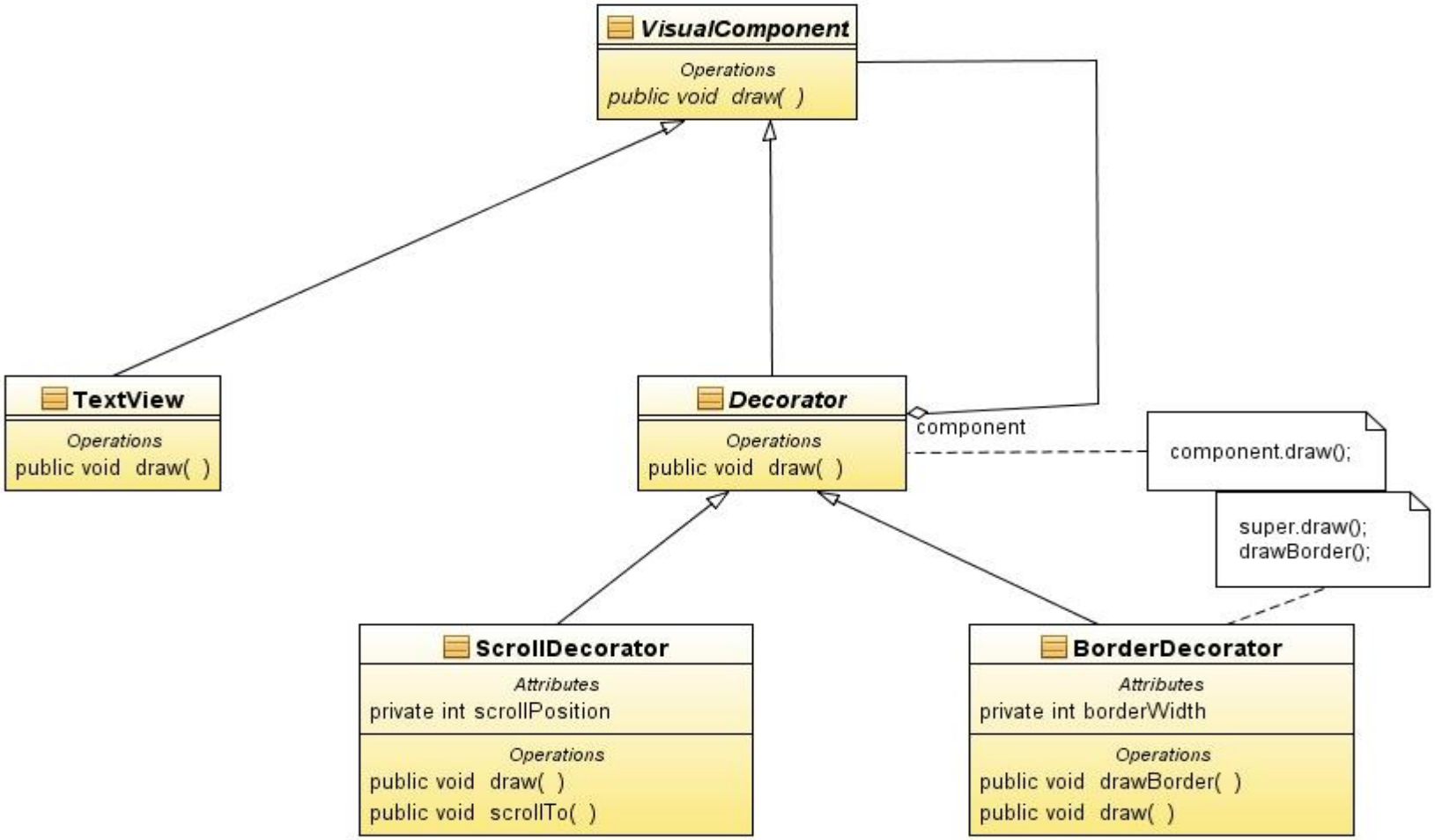
» Motivazione

- GUI toolkit dovrebbe consentire di aggiungere proprietà come bordi, scorrimento, ai singoli elementi grafici
- Modo per aggiungere responsabilità è tramite ereditarietà
  - Estendere una classe e aggiungere il *bordo*

Decorator (2)

- Approccio più flessibile è racchiudere il componente da decorare in un altro che ha la responsabilità di aggiungere il bordo
  - Oggetto contenitore detto *decorator*
  - Decorator ha un'interfaccia conforme all'oggetto decorato in modo tale da essere trasparente ai vari client
  - Decorator trasferisce le richieste al componente decorato effettuando azioni aggiuntive (esempio decorando il bordo) prima o dopo il trasferimento della richiesta
  - Essendo trasparente ai client è possibile annidare i decorator consentendo l'aggiunta di un numero illimitato di responsabilità agli oggetti decorati

Decorator (3)


VisualComponent
Operations
public void draw( )
TextView
Operations
public void draw( )
Decorator
Operations
public void draw( )
component
component.draw();
super.draw();
drawBorder();
ScrollDecorator
Attributes
private int scrollPosition
Operations
public void draw( )
public void scrollTo( )
BorderDecorator
Attributes
private int borderWidth
Operations
public void drawBorder( )
public void draw( )

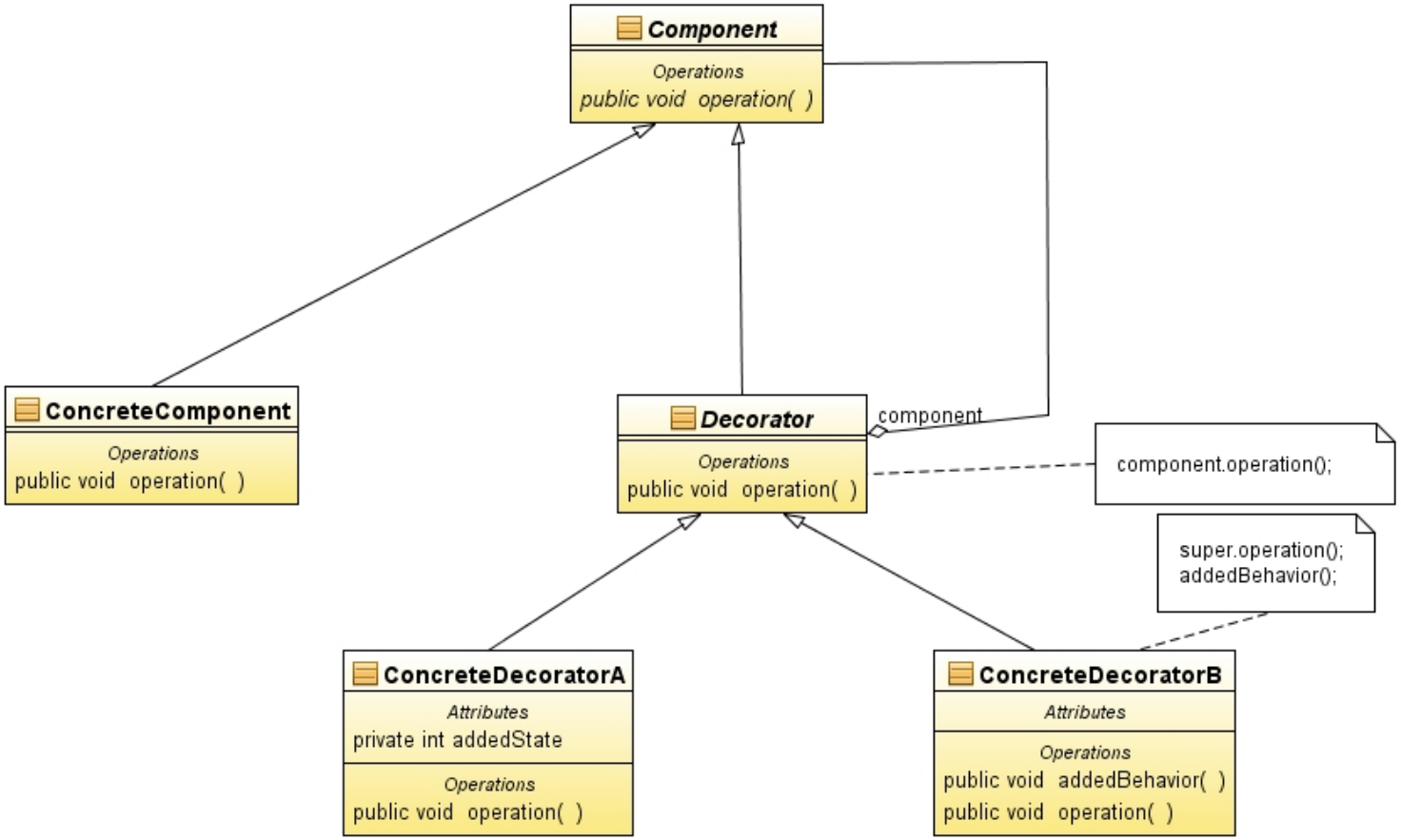
Decorator (4)

» Applicabilità

- Si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente ed in modo trasparente, senza coinvolgere altri oggetti
- Si vuole poter togliere responsabilità agli oggetti
- L'estensione attraverso le definizione di sottoclassi non è praticabile. Esplosione di sottoclassi per supportare ogni possibile combinazione

Decorator (5)

Struttura



Decorator (6)

» Partecipanti

- **Component** (VisualComponent)
  - Definisce l'interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente
- **ConcreteComponent** (TextView)
  - Definisce un oggetto al quale possono essere aggiunte responsabilità ulteriori
- **Decorator**
  - Mantiene un riferimento ad un oggetto `Component` e definisce un'interfaccia conforme all'interfaccia di `Component`
- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  - Aggiunge responsabilità al componente

Decorator (7)

» Collaborazioni

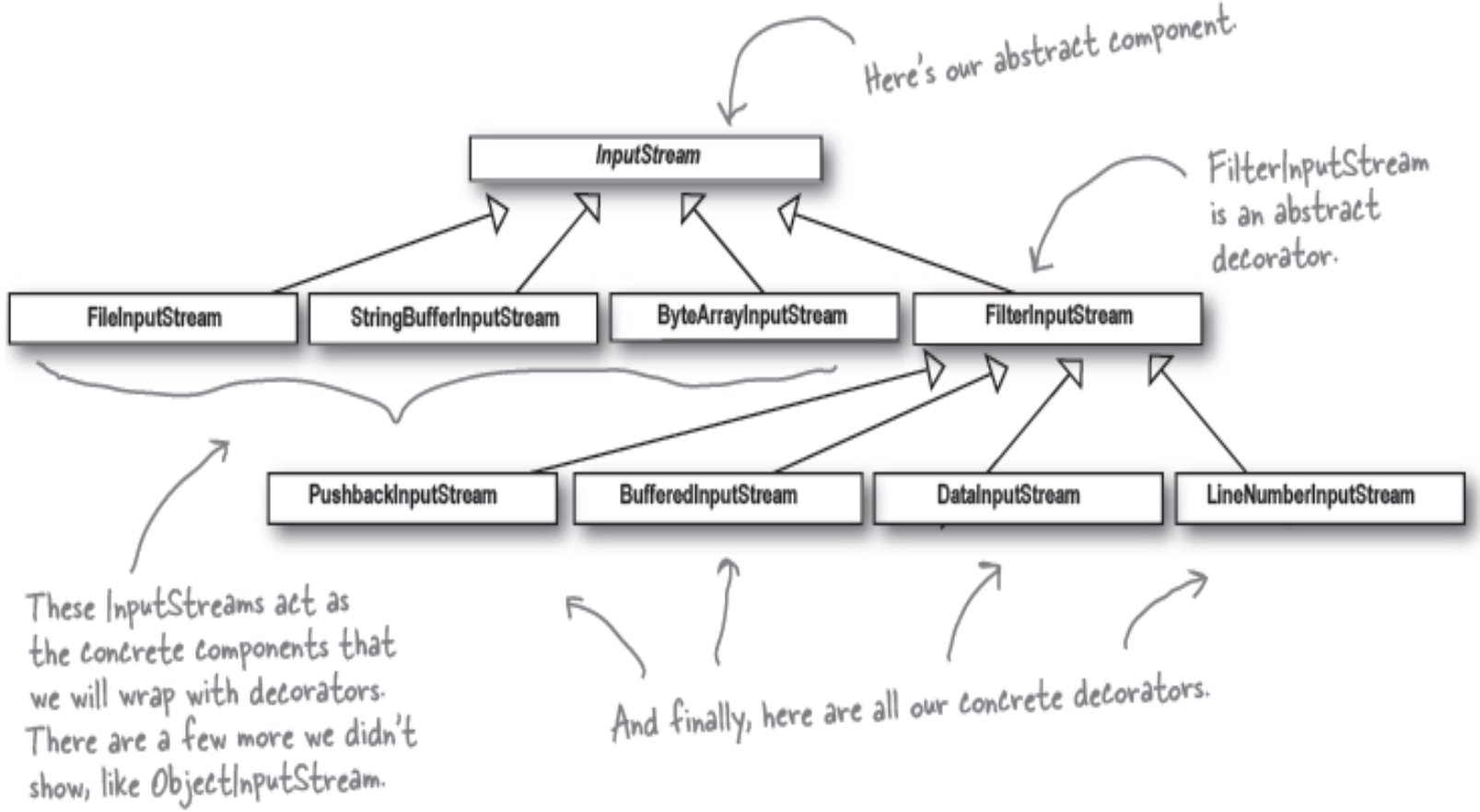
- Un Decorator trasferisce le richieste al suo oggetto Component. Può svolgere opzionalmente operazioni ulteriori prima e dopo il trasferimento della richiesta

» Conseguenze

- Maggiore flessibilità rispetto all'utilizzo dell'ereditarietà (multipla) statica
  - Responsabilità possono essere aggiunte e rimosse in esecuzione semplicemente collegando e scollegando i decoratori agli oggetti decorati
  - In caso di ereditarietà è necessario creare una nuova classe per ogni responsabilità (es. BorderedScrollableTextView, BorderedTextView)
- Evita di definire classi troppo complesse nella gerarchia

Decorator (8)

» Codice d'esempio



Decorator (9)


```
public abstract class InputStream {

    public abstract int read() throws IOException;
    public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
    }
    public int read(byte b[], int off, int len) throws IOException {
           ..............................................
    }

    public long skip(long n) throws IOException {
           ..............................................
    }

    public int available() throws IOException {
           return 0;
    }

    public void close() throws IOException {}

    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws IOException {
           throw new IOException("mark/reset not supported");
    }

    public boolean markSupported() {
           return false;
    }

}
```

Decorator (10)


```
public class FileInputStream extends InputStream {
    public FileInputStream(String name) throws FileNotFoundException {
        this(name != null ? new File(name) : null);
    }
    public FileInputStream(File file) throws FileNotFoundException {
    ..........................
    }
    public FileInputStream(FileDescriptor fdObj) {
    ..............
    }
    private native void open(String name) throws FileNotFoundException;
    public native int read() throws IOException;
    private native int readBytes(byte b[], int off, int len) throws IOException;
    public int read(byte b[]) throws IOException {
      return readBytes(b, 0, b.length);
    }
    public int read(byte b[], int off, int len) throws IOException {
      return readBytes(b, off, len);
    }
    public native long skip(long n) throws IOException;
    public native int available() throws IOException;
    public void close() throws IOException {
      ..............................
    }

.........................

}
```

Decorator (11)


```
public class FilterInputStream extends InputStream {

    protected InputStream in;

    protected FilterInputStream(InputStream in) {
      this.in = in;
    }
    public int read() throws IOException {
      return in.read();
    }
    public int read(byte b[]) throws IOException {
      return read(b, 0, b.length);
    }
    public int read(byte b[], int off, int len) throws IOException {
      return in.read(b, off, len);
    }
    public long skip(long n) throws IOException {
      return in.skip(n);
    }
    public int available() throws IOException {
      return in.available();
    }
    public void close() throws IOException {
      in.close();
    }
    public synchronized void mark(int readlimit) {
      in.mark(readlimit);
    }
    public synchronized void reset() throws IOException {
      in.reset();
    }
    public boolean markSupported() {
      return in.markSupported();
    }
}
```

Decorator (12)


```
public class BufferedInputStream extends FilterInputStream {


    public BufferedInputStream(InputStream in) {
        ..........................
    }

    public BufferedInputStream(InputStream in, int size) {
        ..........................
    }

    public synchronized int read() throws IOException {
        ..........................
    }

    private int read1(byte[] b, int off, int len) throws IOException {
        ................
    }

   ..........................

}
```

Pattern Comportamentali

- Si occupano di algoritmi e dell'assegnamento di responsabilità tra oggetti collaboranti
- Non descrivono solo pattern di classi e oggetti ma pattern di comunicazione (flussi di controllo difficili da seguire durante l'esecuzione) fra questi ultimi
- Basati su *classi*
  - Utilizzano l'ereditarietà per distribuire responsabilità e comportamento tra oggetti diversi
- Basati su *oggetti*
  - Usano la composizione degli oggetti al posto dell'ereditarietà

Observer (1)

» Scopo

- Definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente

» Detto anche…

- Dependents, Publish-Subscribe

» Motivazione

- GUI toolkit dovrebbe consentire la separazione tra l'interfaccia grafica presentata all'utente e la sottostante struttura dati dell'applicazione
- Classi che implementano un modello e una rappresentazione in modo separato possono essere usate indipendentemente, ma possono anche lavorare insieme

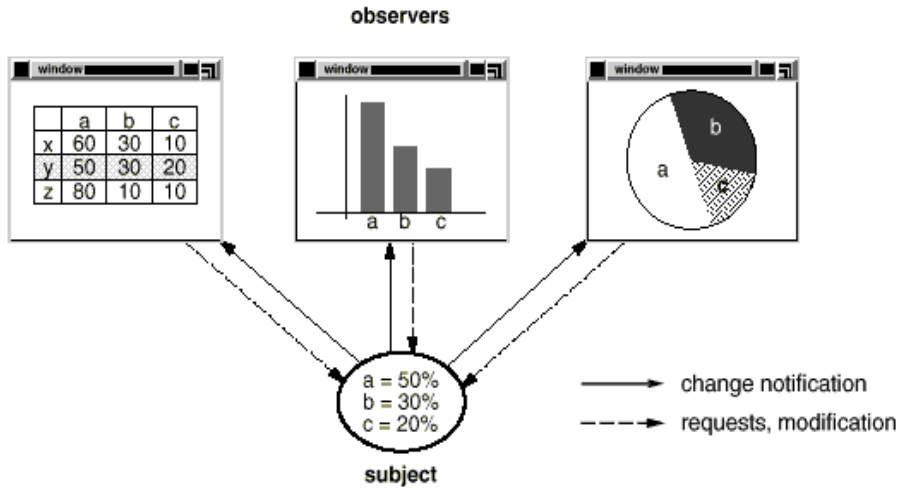
Observer (2)

»Foglio elettronico non sa nulla del grafico a barre e viceversa

»Se l'utente cambia le informazioni nel foglio elettronico le modifiche vengono apportate sul grafico a barre, e viceversa

»Foglio elettronico e grafico a barre dipendono dalla struttura dati sottostante e vengono notificati di qualsiasi modifica dello stato

»Possono esserci più di due oggetti (es. grafico a torta)



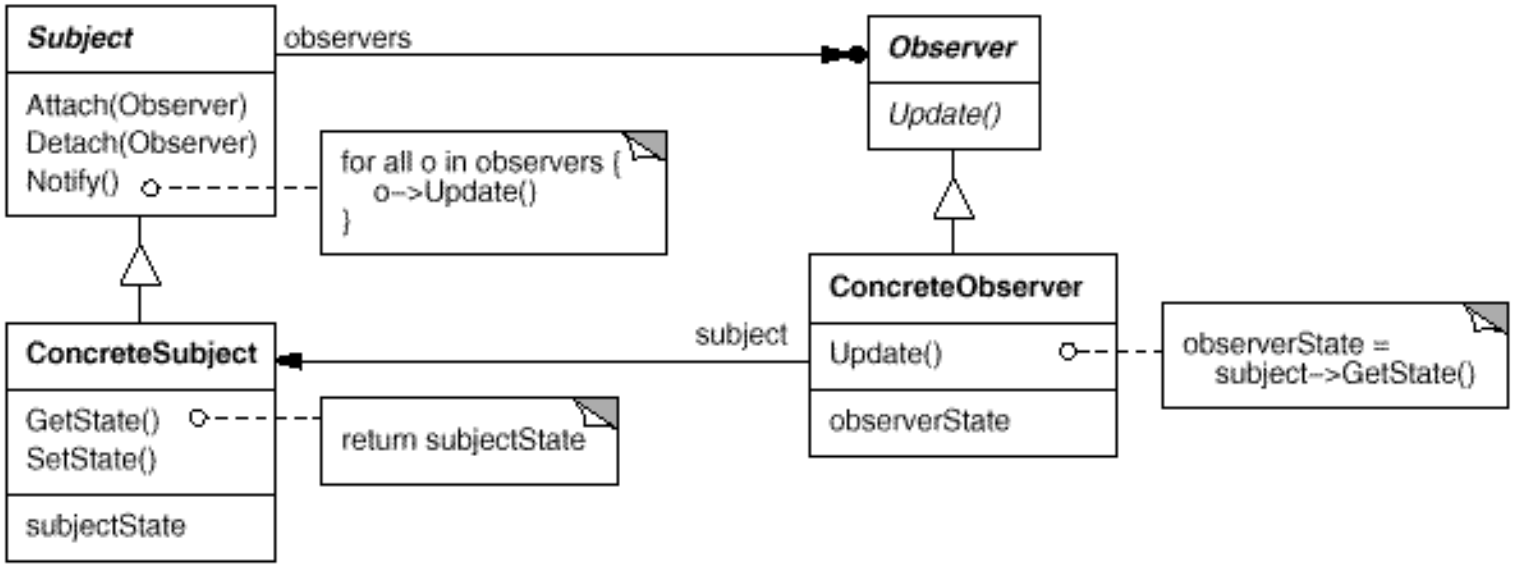
Observer (3)

» Applicabilità

- Quando un'astrazione presenta due aspetti, di cui dipendente dall'altro. Incapsulando questi aspetti in due oggetti separati è possibile riusarli indipendentemente
- Quando una modifica ad un oggetto richiede modifiche ad altri oggetti che dipendono da questo, ma in generale non si conosce il numero di oggetti dipendenti
- Quando un oggetto ha bisogno di notificare ad altri oggetti senza conoscerne l'dentità precisa. In altre parole si vuole mantenere un alto livello di disaccoppiamento

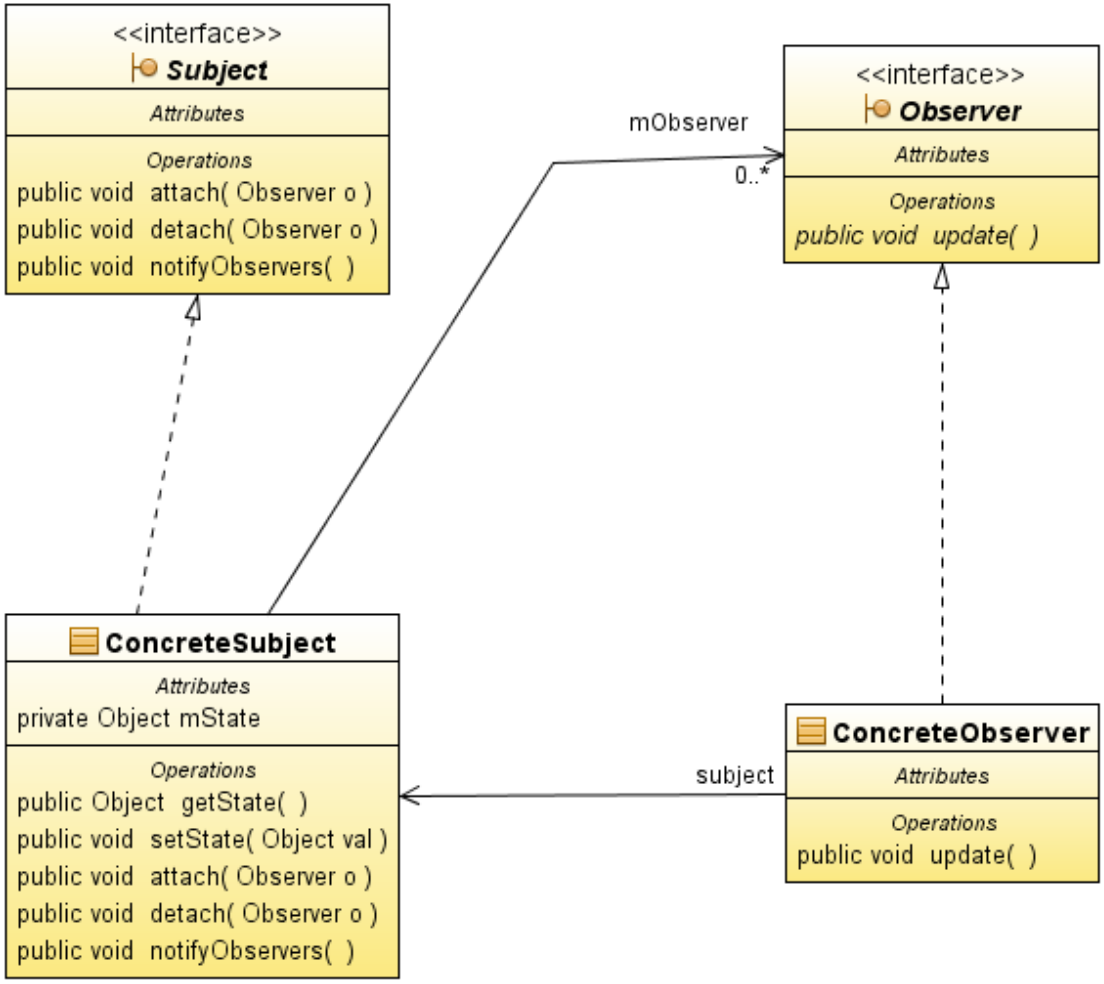
Observer (4)

Struttura



Observer (5)

Struttura usando interfacce



Observer (6)

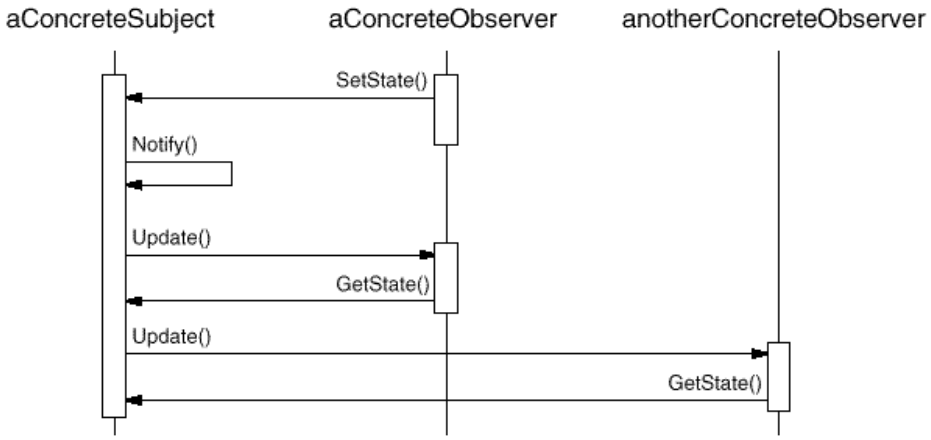
» Partecipanti

- **Subject**
  - Conosce i propri `Observer`. Un numero qualsiasi di oggetti `Observer` può osservare un oggetto
- **Observer**
  - Fornisce un'interfaccia di notifica per gli oggetti a cui devono essere notificati i cambiamenti del `Subject`
- **ConcreteSubject**
  - Contiene lo stato a cui gli oggetti `ConcreteObserver` sono interessati
- **ConcreteObserver**
  - Memorizza un riferimento a un oggetto `ConcreteSubject`
  - Contiene informazioni che devono essere costantemente sincronizzate con lo stato del `Subject`
  - Implementa l'interfaccia di notifica di `Observer` per mantenere il proprio stato consistente con quello del `Subject`

Observer (7)

» Collaborazioni

- `ConcreteSubject` notifica ai propri `Observer` quando il suo stato cambia in modo tale da poter rendere inconsistente lo stato degli `Observer` con il proprio
- Dopo che a un `ConcreteObserver` viene notificato un cambio di stato nel `Subject` concreto questo può richiedere ulteriori informazioni riguardo al `Subject`. `ConcreteObserver` userà le informazioni ottenute dal `Subject` per sincronizzare il proprio stato



Observer (8)

» Conseguenze

E’ possibile variare soggetti e osservatori in modo indipendente. Inoltre, è possibile riusare oggetti senza riusare i loro osservatori e viceversa.

- Accoppiamento astratto far `Subject` e `Observer`
  - I `Subject` conoscono una lista di `Observer` conformi ad una specifica interfaccia. Il soggetto non conosce nessuna classe concreta di alcun osservatore. Accoppiamento tra `Subject` o `Observer` è astratto e minimale
- Supporto per comunicazioni *broadcast*
  - La notifica è inoltrata non a uno specifico destinatario ma a tutti gli `Observer` interessati che si non registrati. Il soggetto non si occupa di quanti osservatori sono presenti; la sua unica responsabilità è inoltrare a ognuno di loro la notifica
  - E’ possibile togliere o aggiungere osservatori in qualsiasi momento
  - Osservatore deciderà come e se reagire

Observer (9)

» .....

- Aggiornamenti inattesi
  - `Observer` non hanno conoscenza della presenza l'uno dell'altro, possono essere del tutto all'oscuro del costo effettivo di richiedere una modifica al `Subject`. Una modifica può scatenare una catena di aggiornamenti e sincronizzazioni su tutti gli osservatori e su tutti gli oggetti da questi dipendenti
  - Protocollo di notifica non fornisce dettagli su cosa sia cambiato effettivamente nel soggetto. Senza protocolli aggiuntivi che aiutino gli osservatori a capire cosa sia cambiato nel soggetto, questi potrebbero faticare molto per capire il cambiamento di stato

Observer (10)


```
public interface Observer {
  void update(Observable o, Object arg);
}

public class Observable {
    private boolean changed = false;
    private Vector obs;
    public Observable() {
  obs = new Vector();
    }
    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
  if (!obs.contains(o)) {
      obs.addElement(o);
  }
    }
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }
    public void notifyObservers() {
      notifyObservers(null);
    }
 ……
```

Observer (11)


```
    ……..
    public void notifyObservers(Object arg) {
        Object[] arrLocal;
        synchronized (this) {
          if (!changed)
                return;
          arrLocal = obs.toArray();
          clearChanged();
        }
        for (int i = arrLocal.length-1; i>=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }
     public synchronized void deleteObservers() {
          obs.removeAllElements();
    }
    protected synchronized void setChanged() {
          changed = true;
    }
    protected synchronized void clearChanged() {
          changed = false;
    }
    public synchronized boolean hasChanged() {
          return changed;
    }
    public synchronized int countObservers() {
          return obs.size();
    }
}
```

Bibliografia

» Titolo: Design Patterns (Elementi per il riuso di software a oggetti)
Autori: Gamma, Helm, Johnson, Vlissides (GoF)
Lingua: Italiana
ISBN: 887192150
Casa Editrice: Addison-Wesley (1995)
Titolo Inglese: *Design Patterns: Elements of Reusable Object-Oriented Software*
ISBN: 0201633612