

---

# Sistemi Orientati agli Oggetti

Lo sviluppo di sistemi orientati agli oggetti mira a descrivere un software nelle sue diverse fasi (analisi, progettazione, codifica) in termini di **classi**. Tali sistemi sono documentati focalizzando sulle classi ed usando diagrammi di classi. La classe è spesso pensata come l'entità riusabile più piccola (ovvero la **granularità** del riuso è a livello di classe).

Svantaggi della programmazione OO :-)

- Le conseguenze dell'uso delle classi possono non essere chiare.
- La progettazione può essere non corretta ai fini del riuso o di altre proprietà.
- Anche se ben progettate, le classi sono sì entità riusabili ma esse sono troppo piccole per ottenere grandi vantaggi dal loro riuso.

Quello che si spera (e si tenta) di riusare è un insieme di classi, ovvero la *soluzione ad un certo problema*, che molto spesso è costituita da più di una classe.

Entità con granularità più grande, **architetture di sistemi ad oggetti**, permettono di ridurre i costi di sviluppo, poiché cercano di riusare insiemi di classi e di produrre sistemi più facili da evolvere.

L'**architettura software** di un sistema è un artefatto, frutto della attività di progettazione. Una architettura software è descritta dai suoi *componenti* e dalle *relazioni* tra i componenti.

- **Componenti**, costituiscono i 'building block' di un sistema (es: moduli, classi, funzioni)
- **Relazioni**, denotano una connessione tra componenti: aggregazione, eredità, interazione

---

# Design Pattern

Nella progettazione del software, riferirsi a componenti con granularità più grande della classe, permette di ottenere un migliore riuso ed una maggiore astrazione.

I ***design pattern*** sono strutture software (ovvero micro-architetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti.

Tali micro-architetture specificano le diverse classi ed oggetti coinvolti e le loro interazioni.

Esistono tanti cataloghi di design pattern, per vari contesti (sistemi centralizzati, concorrenti, distribuiti, real-time, etc.)

Documentazione sul web:

[hillside.net](http://hillside.net)

fornisce liste articoli e libri su design pattern, conferenze, link, ...

[www.patterndepot.com](http://www.patterndepot.com)

Libro su Design Pattern con codice Java (come Gamma, vedi dopo)

[www.bruceeckel.com](http://www.bruceeckel.com)

Libro “Thinking in Patterns” in via di preparazione (come Gamma)

---

# Libri su Design Pattern

***Gamma***, Helm, Johnson, Vlissides. “Design Patterns – Elements of Reusable Object-Oriented Software”. 1994

E' il primo ed il più conosciuto libro sui design pattern. Cataloga 23 pattern per sistemi centralizzati.

***Buschmann***, Meunier, Rohnert, Sommerlad, Stal. “Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. 1996.

Cataloga pattern per sistemi centralizzati e distribuiti.

***Lea***. “Concurrent Programming in Java: Design Principles and Patterns (2nd Edition)”. 1999.

Cataloga pattern per sistemi concorrenti.

***Schmidt***, Stal, Rohnert, Buschmann. “Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. 2000.

Cataloga pattern per sistemi concorrenti e distribuiti.

Vlissides, Coplien, Kert. “Pattern Languages of Program Design”

Altri: vedi Coplien, Vlissides

---

# Design Pattern: Definizione e Usi

I design pattern descrivono un ***problema di design ricorrente*** che si incontra in ***specifici contesti*** di progettazione e presentano una ***soluzione collaudata generica*** ma specializzabile.

In altre parole, sono soluzioni riusabili per una certa classe di problemi ricorrenti.

Usi:

***Documentano*** soluzioni già applicate che si sono rivelate di successo per certi problemi e che si sono evolute nel tempo.

***Aiutano i principianti*** ad agire come se fossero esperti.

***Supportano gli esperti*** nella progettazione di software su grande scala.

***Evitano*** di re-inventare concetti e soluzioni, riducendo il costo del software.

Forniscono un ***vocabolario*** comune e permettono una ***comprensione dei principi del design.***

Analizzano le loro proprietà ***non-funzionali***: ovvero, come una funzione è portata a termine, es. affidabilità, cambiabilità, sicurezza, testabilità, riuso.

---

# Design Pattern - Descrizione

Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione:

- Le classi e le istanze che vi partecipano
- I loro ruoli e come collaborano
- La distribuzione delle responsabilità

Include 5 parti fondamentali:

- **Nome:** permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern
- **Intento:** descrive brevemente le funzionalità e lo scopo
- **Problema (Motivazione+Applicabilità):** descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo
- **Soluzione:** descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni
- **Conseguenze:** indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern

Nella sezione 'Problema' della descrizione di un design pattern si parla di **forze** (come in fisica), in pratica **obiettivi e vincoli**, spesso contrastanti, che si incontrano nel contesto di quel design pattern.

Altre parti possono essere presenti nella descrizione:

- **Esempi di utilizzo:** illustrano dove il design pattern è stato usato
- **Codice:** fornisce porzioni di codice che lo implementano

---

# Il Design Pattern Observer - 1

## Intento:

Definire una dipendenza tra un particolare oggetto (detto **subject**) ed altri oggetti detti osservatori, cosicché quando il subject cambia stato, gli osservatori sono notificati ed aggiornati.

Es. Un oggetto contiene le coordinate di un punto e la sua variazione deve essere notificata ad un oggetto che effettua una rappresentazione grafica del punto.

## Problema (Forze):

Il subject deve essere indipendente dal numero e dal tipo degli osservatori. In altre parole, il subject non fa assunzioni sugli oggetti che dipendono da esso. Oggetti lascamente accoppiati sono più facili da riusare.

Deve essere possibile aggiungere nuovi osservatori durante l'esecuzione dell'applicazione.

## Soluzione:

Il subject rende disponibili delle operazioni che consentono ad un osservatore di dichiarare il proprio interesse per un cambiamento di stato.

# Il Design Pattern Observer - 2

Componenti:

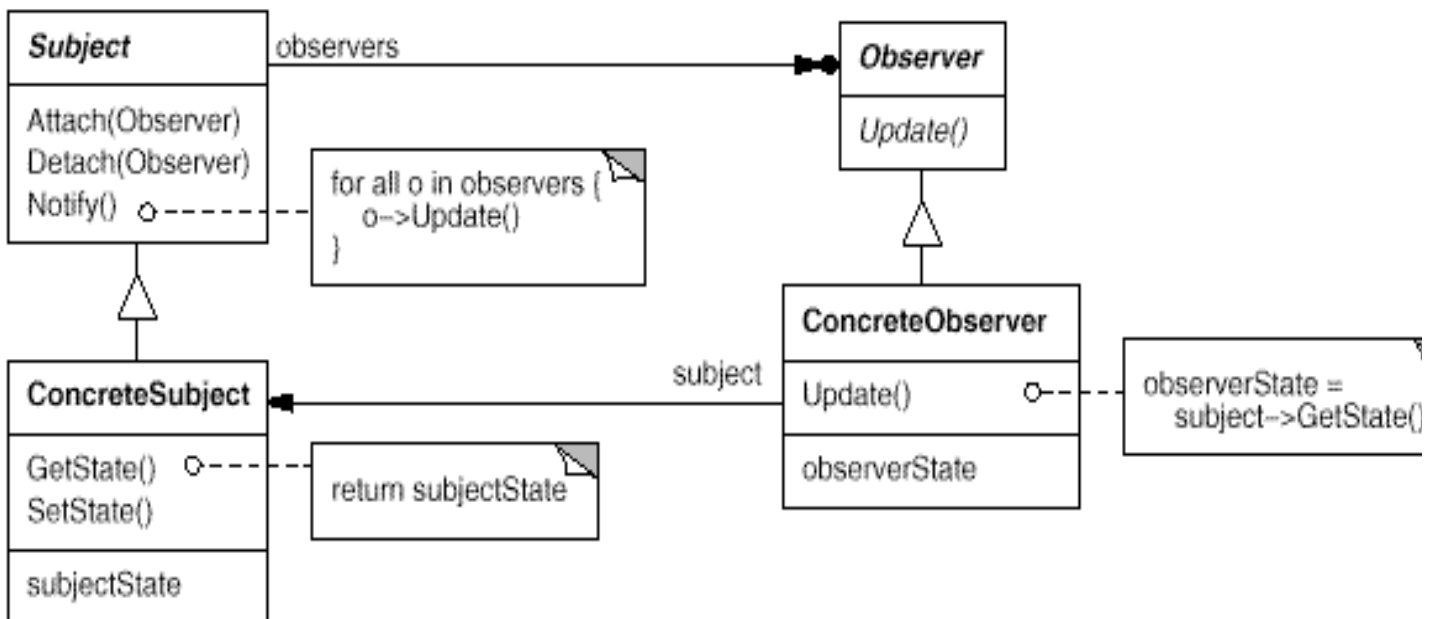
**Subject** è una classe che fornisce operazioni ad oggetti Observer, con metodi come: `attach()`, `detach()` e `notify()`; conosce solo la classe Observer

**Observer** è una classe che fornisce una interfaccia (operazione `update()`) comune a tutti gli oggetti che necessitano notifica

**ConcreteSubject** eredita da Subject, è la classe il cui stato deve essere notificato, conosce solo la classe Subject

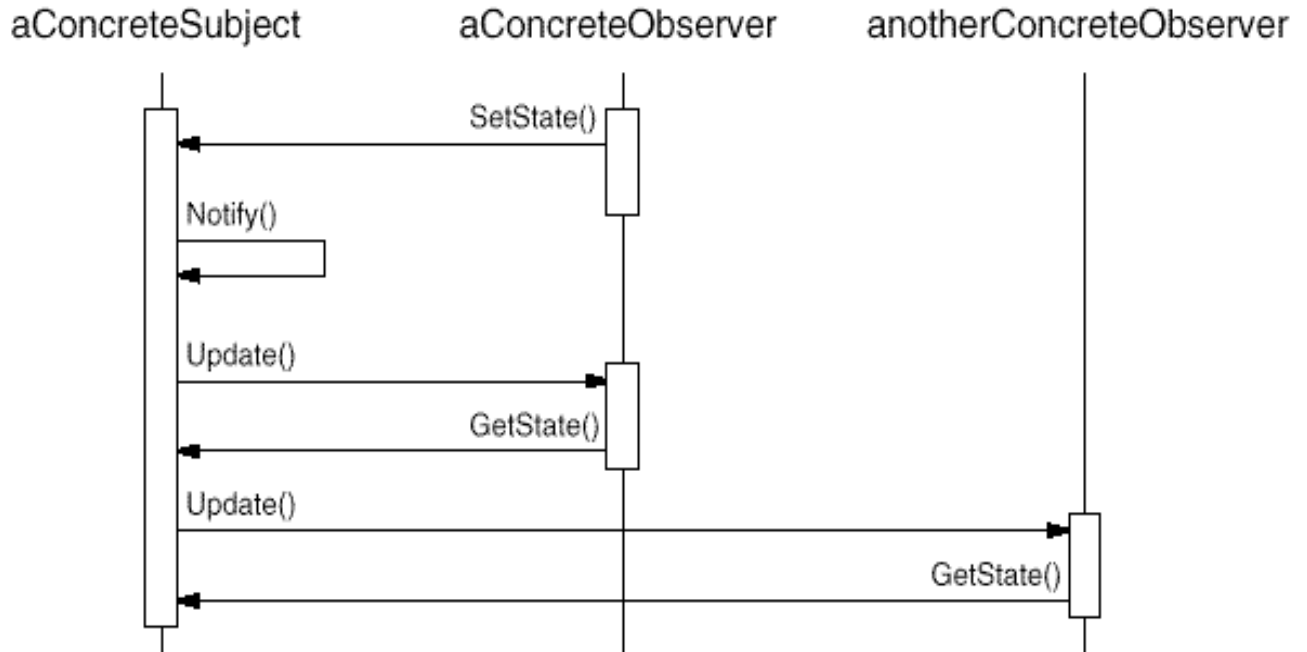
**ConcreteObserver** eredita da Observer, è la classe che vuole essere notificata, conosce solo la classe Observer

Struttura (class diagram):



# Il Design Pattern Observer - 3

Comportamento a runtime



## Conseguenze:

Il Subject conosce solo la classe Observer e non ha bisogno di conoscere le classi ConcreteObserver. ConcreteSubject e ConcreteObserver non sono accoppiati quindi più facili da riusare e modificare.

## Esempi di utilizzo:

Smalltalk Model View Controller (framework dell'interfaccia nell'ambiente Smalltalk)

X Window (ma non ad oggetti), ogni finestra registra il suo interesse ad un particolare evento e tutte le informazioni sono inviate indietro (callback) quando l'evento accade.

---

# Design Pattern Observer in Java

Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è parte della libreria `java.util`

In tale libreria ci sono due classi Java che permettono di implementare l'Observer: `Observable` ed `Observer`.

La classe `Observable` tiene traccia di tutti gli oggetti che vogliono essere informati quando accade un cambiamento. La classe `Observable` notifica il cambiamento di stato e permette agli osservatori di aggiornare il proprio stato, attraverso il metodo `notifyObservers()`.

La classe `Observable` ha una variabile (flag) che indica se lo stato è cambiato. Questo è settato dal metodo `setChanged()`. La chiamata a `setChanged()` è da implementare nella classe che la eredita, secondo la logica del programma.

`Observer` è una interfaccia che ha solo il metodo `update()`.

Questo metodo è chiamato dall'oggetto osservato. `update()` può avere un argomento che indica quale oggetto ha causato l'aggiornamento.

Esempio con codice: vedi Address Book

---

# Alcuni Design Pattern

Da Gamma:

## Abstract Factory

Fornisce una interfaccia per creare famiglie di oggetti dipendenti senza specificare le loro classi.

## Adapter

Converte l'interfaccia di una classe in un'altra interfaccia che il client si aspetta.

## Facade

Fornisce una interfaccia unificata per un set di interfacce di un sottosistema.

## Mediator

Definisce un oggetto che incapsula come un gruppo di oggetti interagiscono. Evita che gli oggetti riferiscano l'un l'altro.

## Proxy

Fornisce un surrogato per un altro oggetto per controllare l'accesso a quest'ultimo.

## Strategy

Definisce una famiglia di algoritmi, incapsulando ciascuno di essi, e rendendoli intercambiabili. Strategy lascia variare gli algoritmi indipendentemente dai client che li usano.