

# CORSO DESIGN PATTERN

Definizioni, concetti e  
classificazione dei design pattern

**Dott. Romina Fiorenza**  
[fiorenza.romina@gmail.com](mailto:fiorenza.romina@gmail.com)

# Obiettivi del corso



- Cosa sono i design pattern:
  - ▣ concetti fondamentali
  - ▣ ragioni di utilizzo
  - ▣ uso
  - ▣ terminologia
- I più comuni design pattern: i pattern del GoF
- Come si scelgono e si usano i design pattern

# Un problema di design

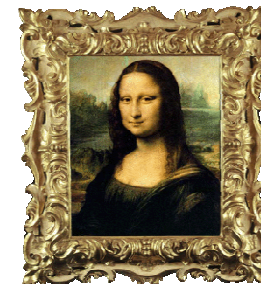


- La fase di design di un sistema è quella cruciale tra l'analisi e l'implementazione
- L'analisi segue metodologie
- Il design non segue un inquadramento canonico:
  - Introduce elementi e oggetti
  - Definisce ruoli e responsabilità
  - Usa concetti come indipendenza, flessibilità, riuso, ...
- Cosa si intende dunque per ***buon design***?

# Le metriche

- Affrontare un problema, in generale, significa risolverlo.
- Ma il come... può fare la differenza!!!
- Mentre esistono criteri precisi per determinare la “bontà” di un programma → prestazioni, modificabilità, eleganza...
- Non esistono parametri per valutare l’architettura generale di un sistema →

*design è una forma d’arte!*



# Un processo induttivo

- Deduzione logica & modelli per risolvere i problemi
- Ma il nostro cervello ha bisogno di “allenamento”
- Il cervello umano identifica in modo inconscio delle strutture simili = **patterns ricorrenti**
- Una situazione “già vissuta”
  - se riconosciuta
  - può essere gestita grazie all’esperienza maturata

ma non per tutti...



# Non si inventa nulla

- I pattern non hanno pretesa di inventare nulla!
- Sono solo pratiche di buon senso, frutto dell'esperienza
- **Processo induttivo:**
  1. Problemi simili
  2. Trovo una soluzione “comune”
  3. Analisi e astrazione della soluzione dal contesto
  4. Catalogazione →  
nome, problema, soluzione, conseguenze = **PATTERN**



# Perché usare i pattern?

1. I pattern permettono di sfruttare l'esperienza collettiva dei progettisti esperti

Sono frutto di esperienze reali di sviluppo e progettazione software

2. Ogni pattern riguarda un problema specifico e ricorrente

Inutile “reinventare la ruota”, basta studiare soluzioni già note

3. I pattern si possono combinare per costruire architetture sw con proprietà specifiche

Compresa una buona idea, è possibile combinarla con altre e realizzare intere architetture “ben fatte”

# Ruolo dei pattern

- Deposito di conoscenza
  - Esempio di buone pratiche
  - Vocabolario comune per i progettisti
  - Utili per la standardizzazione
  - Aiuto per la documentazione architettura sw
- 
- Sorgente di miglioramento continuo
  - Incoraggiamento alla generalità → astrazione
  - Schematizzazione principi complessi
  - Panorama di soluzioni diverse e confrontabili



# Ruolo dei pattern

□ Dal punto di vista delle architetture software, i design pattern

- diminuiscono i rischi
- incrementano la produttività
- aumentano la standardizzazione
- favoriscono un più alto livello di qualità



# Origini e storia

- **1977** → L'architetto Christopher Alexander introduce il concetto di design pattern
- **1987** → Beck e Cunningham, di Textronix, mostrano le idee di pattern per il design della GUI di Smalltalk alla **OOPSLA**
- **1991** → **Erich Gamma**, tesi di dottorato ET++
- **1992** → Pubblicazione di Johnson alla **OOPSLA**
- **1994** → Nasce prima conferenza **PLoP**
- **1995** → Gamma, Helm, Johnson, Vlissides ("Gang of Four" - GoF) presentano il libro *Design Patterns: Elements of Reusable Object Oriented Software*
- **1996** → Buschmann, Meunier, Rohnert, Sommerland, Stal, realizzano il lavoro *Pattern-Oriented Software Architecture: A System of Patterns*

# Conferenze internazionali

- **OOPSLA**: Conferenza annuale su **O**bject-**O**riented **P**rogramming, **S**ystems, **L**anguages & **A**pplications
- La prima OOPSLA si è tenuta a Portland (Oregon) nel **1986**.
- **PLOP**: Conferenza annuale su **P**attern **L**anguages **O**f **P**rograms, il cui obiettivo è incrementare l'espressione dei pattern. Qui gli autori hanno l'opportunità di rifinire e estendere i loro pattern.
- La prima si è tenuta in Illinois nel **1994** ed è stata la prima conferenza dedicata ai design pattern. Oggi viene organizzata in collaborazione con la conferenza OOPSLA.

# Alexander: un architetto

- **Christopher Alexander**

è un architetto, matematico  
e professore austriaco presso

la University of California

noto per la progettazione

di edifici complessi in California, Giappone e Messico.



- Famoso soprattutto per i suoi contributi in ambito teorico.
- Importante il suo libro "A Pattern language", 253 patterns che risolvono problemi comuni delle città
- Introduce il concetto di **design pattern** tra 1977-1979

# Esempio in architettura classica

## Pattern in architettura classica:

- **Contesto:** situazione problematica di progetto
- **Problema:** insieme di forze in gioco in tale contesto
- **Soluzione:** insieme di forme/regole da applicare per risolvere tali forze

## Esempio – Il Patio

- **Contesto**
  - Costruzione di una villetta
- **Forze**
  - Vorremmo disporre di uno spazio esterno
  - Piove spesso
- **Soluzione**
  - Realizzare un patio



# Design Pattern (in architettura)

Un pattern descrive il **nucleo** di una **soluzione** relativa un **problema** che compare **frequentemente** in un dato **contesto**

Il modello della soluzione deve essere strutturato in modo che

“si possa **usare** tale soluzione un **milione** di volte, senza **mai** farlo allo stesso modo”

*Definizione secondo Alexander nel libro “A Pattern language”*

# Gamma: un progettista

- **Erich Gamma**

è un informatico svizzero

noto come co-autore del libro Design Patterns:

*Elementi per il riuso di software ad oggetti.*



- Nel 1981 consegue un dottorato a Zurigo su Design Pattern.
- Nel 1995 guadagna vasta fama nell'ambiente della programmazione OO (pubblicazione del libro **Design Patterns** insieme alla **GoF**).
- Ha dato un importante contributo alla diffusione e lo sviluppo del concetto di design pattern.
- Gamma è anche il principale progettista di **JUnit**, un framework per il collaudo di moduli software (unit testing) Java (1998 in collaborazione con Kent Beck) e di **Eclipse**, piattaforma di sviluppo orientato agli oggetti del cui sviluppo è tuttora il principale responsabile.

# Un Design Pattern (Gamma)

- Un pattern software è
  - ▣ una **soluzione** provata ed ampiamente applicabile ad un particolare **problema di progettazione**, descritta in una forma **standard** che possa essere **facilmente riusata**
  
- *“Descrizione di classi e oggetti comunicanti adatti a risolvere un problema progettuale generale in un contesto particolare”*

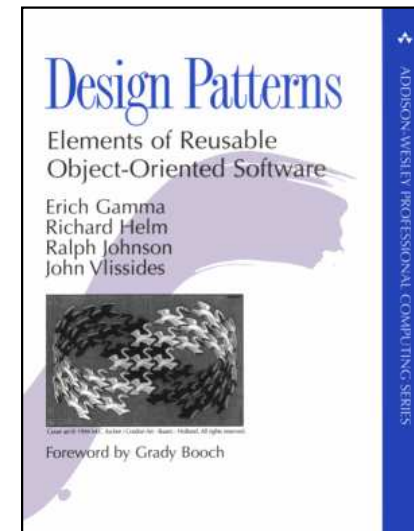
*Definizione secondo Gamma, Helm, Johnson, Vlissides in*

*“Design Patterns: Elements of Reusable Object-Oriented Software”, 1995*



# Il libro del GoF


- Il libro “**Design Patterns: Elements of Reusable Object-Oriented Software**”, è considerato la “**bibbia**” sui pattern.
- Primo libro che
  - Applica concetti di design pattern all'OOP
  - Individua i principali pattern
- E' un catalogo di 23 pattern, descritti per:
  - Nome
  - Problema
  - Soluzione
  - Conseguenze



# Tipi di pattern

Le principali categorie di pattern sw sono:

□ **architetturali**   
(o *stili architetturali*)

□ **progettuali**   
(*micro-architetture*) → **GoF**

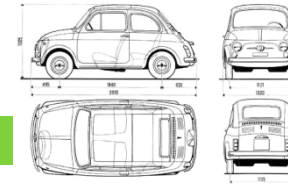
□ **idiomi**   
(*basso livello, legati al linguaggio*)

# Pattern architetturali



- Sono pattern di alto livello
- Guidano nella scelta della decomposizione del sistema in sottosistemi, indicando
  - ruoli e responsabilità
  - regole e criteri di comunicazione reciproca
- Esempi di pattern architetturali sono:
  - Client-Server
  - Model-View-Controller (MVC)
  - Peer-to-Peer

# Pattern progettuali



- Sono quelli che comunemente chiamiamo **design pattern**
- Sono utilizzati nella progettazione di dettaglio dei singoli elementi architettonici → micro architetture
- Forniscono inoltre uno schema per raffinare le comunicazioni fra gli elementi di un sistema sw
  
- Esempi di pattern progettuali sono i pattern del GoF:
  - Factory
  - Command
  - Proxy
  - Observer
  - ecc...

# Idiomi



- Sono pattern di basso livello
- Sono specifici per un linguaggio di programmazione
- Descrivono come implementare
  - *aspetti particolari di elementi e/o*
  - *relazioni tra elementi*usando le caratteristiche di un certo linguaggio di programmazione
- Esempio di idioma:
  - **Problema:** come implementare il Singleton in Java?
  - **Idioma:**
    - Definire un attributo privato e statico **instance** del tipo della classe
    - Implementare un metodo pubblico e statico **getInstance**
    - Impostare un costruttore privato

# Pattern vs Idioma

- Un pattern è legato al paradigma
  - propone un modello → soluzione generica
- Un idioma è legato alla tecnologia
  - propone soluzione specifica ad un linguaggio
- Un design idiomático rinuncia alla genericità della soluzione basata su un pattern a favore di una implementazione che poggia sulle caratteristiche (e le potenzialità) del linguaggio

# Linguaggio di Pattern

- Un *pattern language* (o *linguaggio di pattern*) è una famiglia di **pattern correlati**
- Essi hanno in comune le **finalità principali**
- Ne esistono di svariate specie:
  - specifici per certi tipi di sistemi
  - specifici per certi requisiti
- Esempi:
  - linguaggio di pattern per la sicurezza
  - linguaggio di pattern per sistemi distribuiti
  - linguaggio di pattern per web application
  - ecc.

# Indipendenza e collaborazione

I pattern non sono del tutto indipendenti tra loro

- Alcuni pattern sono **alternativi** → *uso uno oppure l'altro*
- Altri pattern sono **sinergici** → *se uso uno, è utile usare anche l'altro*
- Altri possono essere usati in **gruppi** più complessi

E' sempre utile:

- Conoscere svariati pattern
- Ragionare sulle relazioni tra pattern





# Forma canonica (Alexander)

- **Nome:** identificativo
  - **Problema:** descrizione del problema di progettazione
  - **Contesto:** una situazione dove ci si imbatte nel problema
  - **Forze:** descrizione di forze e vincoli rilevanti
  - **Soluzione:** descrizione della soluzione al problema
- 
- **Esempi:** applicazioni esemplificative del pattern
  - **Contesto risultante :** dopo la risoluzione delle forze
  - **Stato** del sistema dopo l'applicazione del pattern
  - **Pattern correlati:** relazioni statiche o dinamiche
  - **Uso noto:** occorrenza del pattern e sua applicazione su sistemi reali


# Formato canonico della GoF

- ❑ **Nome e classificazione**
- ❑ **Sinonimi** altri nomi noti del pattern (se ci sono)
- ❑ **Motivazione** il problema progettuale e come le classi lo risolvono
- ❑ **Intento** cosa fa il pattern / quando funziona la soluzione
- ❑ **Applicabilità** contesti in cui può applicare il pattern
  
- ❑ **Struttura** rappresentazione grafica delle classi entro il pattern
- ❑ **Partecipanti** classi e oggetti partecipanti e loro responsabilità
- ❑ **Collaborazioni** tra i partecipanti per condividere le responsabilità
- ❑ **Conseguenze** costi e benefici
  
- ❑ **Implementazione** suggerimenti e tecniche
- ❑ **Codice esemplificativo** che mostra una possibile implementazione
- ❑ **Usi noti** pattern trovati in sistemi reali
- ❑ **Pattern correlati** pattern fortemente collegati a questo

# Elementi chiave di un design pattern

- **Nome** del pattern:
  - ▣ Elemento del vocabolario di progetto, alto livello di astrazione
- **Problema:**
  - ▣ Quando si applica il pattern?
  - ▣ Problema e contesto, condizioni di applicabilità
- **Soluzione:**
  - ▣ Insieme degli elementi partecipanti
  - ▣ Relazioni, responsabilità e collaborazioni
  - ▣ Schema strutturale, senza implementazione dettagliata
- **Conseguenze**
  - ▣ Costi e benefici dell'applicazione del pattern
  - ▣ Riutilizzabilità, estensibilità, portabilità

# Classificazione dei pattern del GoF

Relazioni tra		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
	Class	Factory method	Adapter (Class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

# GoF: Creational



## Scopi generali:

- Astrazioni dell'istanziamento
  - Rendono un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati

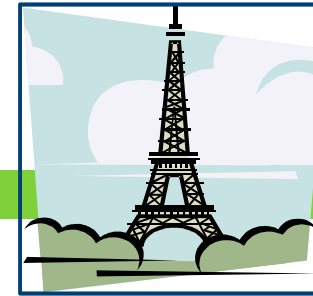
## Quando?

- Importante se il sistema evolve e dipende più dalla composizione di oggetti che dall'ereditarietà
  - comportamenti di creazione flessibili e ben ingegnerizzati

## Vantaggi:

- Disaccoppiano la creazione dall'uso degli oggetti
- Nascondono la creazione e la composizione delle istanze delle classi

# GoF: Structural



## Scopi generali:

- I pattern strutturali si usano per comporre classi e oggetti in strutture più grandi
- I pattern strutturali “di classe” usano **l’ereditarietà** per comporre interfacce o implementazioni

## Quando?

- Soprattutto se occorre far lavorare insieme librerie di classi sviluppate indipendentemente

## Vantaggi:

- Creare strutture affidabili e standardizzate.

# GoF: Behavioral



## Scopi generali:

- I pattern comportamentali si occupano di algoritmi e della distribuzione di responsabilità
- Quelli basati su **classi**, solitamente usano *l'ereditarietà*
- Quelli basati su **oggetti** sfruttano invece le **collaborazioni** fra oggetti

## Quando?

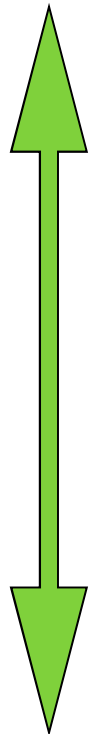
- Per ben ingegnerizzare le interazioni fra le classi

## Vantaggi:

- Migliore separazione dei ruoli all'interno di una collaborazione
- Alta collaborazione, ma Coupling contenuto → massimizzazione del riutilizzo e vantaggi nella manutenzione

# Frequenza media d'uso

*bassa*



*alta*

<b>3.Interpreter</b>	<b>1</b>
<b>6.Memento</b>	<b>1</b>
<b>4.Prototype</b>	<b>2</b>
<b>6.Flyweight</b>	<b>2</b>
<b>5.Mediator</b>	<b>2</b>
<b>11.Visitor</b>	<b>2</b>
<b>3.Builder</b>	<b>3</b>
<b>2.Bridge</b>	<b>3</b>
<b>1.Chain of Responsibility</b>	<b>3</b>
<b>1.Factory Method</b>	<b>4</b>
<b>2.Abstract Factory</b>	<b>4</b>
<b>1.Adapter</b>	<b>4</b>
<b>4.Decorator</b>	<b>4</b>
<b>2.Command</b>	<b>4</b>
<b>8.State</b>	<b>4</b>
<b>9.Strategy</b>	<b>4</b>
<b>5.Singleton</b>	<b>5</b>
<b>3.Composite</b>	<b>5</b>
<b>5.Façade</b>	<b>5</b>
<b>7.Proxy</b>	<b>5</b>
<b>4.Iterator</b>	<b>5</b>
<b>7.Observer</b>	<b>5</b>
<b>10.Template Method</b>	<b>5</b>



# E' una questione di principio!

- Nel mondo della progettazione e dello sviluppo si stanno affermando sempre di più alcuni principi interessanti che vale la pena conoscere
- La loro applicabilità non è totale, ma come le tecniche e i modelli del design, bisogna considerarle in funzione dei casi reali e ... delle proprie convinzioni personali
- Seguono alcuni tra i più famosi

# DRY principle



- **Acronimo di “Don't Repeat Yourself”** o anche conosciuto come **“Duplication is Evil” (DIE)**
- E' un principio base dello sviluppo software che promuove la riduzione delle ripetizioni di tutti i tipi di informazioni
- Afferma che *“In un sistema, ogni singolo pezzo di informazione deve avere un'unica, precisa e non ambigua rappresentazione”*
- E' molto usato nell'ambito delle architetture multitier, dove esiste una grossa mole di dati che transita da un tier all'altro.

# YAGNI principle



- ❑ Acronimo di "**You ain't gonna need it**" → non ne hai davvero bisogno!
- ❑ E' un principio di programmazione "estrema" secondo cui lo sviluppatore non dovrebbe aggiungere funzionalità, se non necessarie.
- ❑ Possibili conseguenze negative dovute allo sviluppo di codice non necessario sono:
  - maggior tempo per progettare/sviluppare/testare  
*oppure al contrario*
  - siccome non serve, la funzionalità non viene progettata bene, non viene documentata e magari non viene testata → pessima qualità del codice

# KISS principle



- Acronimo di “**Keep it simple stupid**” o anche e (forse più famoso) “**Keep it short and simple**”
- Il KISS principle afferma che la semplicità è l’obiettivo chiave del design e che tutte le complessità non necessarie devono essere evitate
- Una versione “ironica”, cambia leggermente il senso inserendo una virgola “**Keep it simple, stupid**” come fosse un imperativo scherzoso verso il progettista / programmatore
  - In forme diverse era stato formulato anche da Albert Einstein, Leonardo da Vinci e altri grandi della storia

# Gli Anti-Pattern

- Gli **anti-pattern** sono problemi che si incontrano frequentemente durante lo sviluppo dei programmi e che dovrebbero essere evitati e “non affrontati”
- Gli anti-pattern sono delle **trappole logiche** in cui il programmatore può facilmente cadere.
- Il termine fu “creato” dalla Gang of Four (GoF) nel loro libro
- Sono in contrasto con gli esempi di buona pratica di progettazione
- Sono un buon complemento ai design-pattern.



# A cosa serve un anti-pattern

- Ad **evitare** un errore già commesso da qualcuno → evitandoci di fare “sbagliando ... s’impara”
- Ad **evidenziare** perché quella soluzione sbagliata può *inizialmente* sembrare vincente per quel problema
- A **descrivere** come passare dalla soluzione sbagliata a quella adatta.
  - in questo caso si parla di **amelioration-pattern**.

# Esempi di anti pattern

- ❑ Reinventare la ruota → reinventing the wheel
- ❑ Reinventare la ruota quadrata → reinventing the Square Wheel
- ❑ Codice spaghetti → spaghetti code
- ❑ Complessità involontaria → accidental complexity
- ❑ Inferno delle DLL → DLL hell
- ❑ Attesa a vuoto - busy spin
- ❑ Anomalia della sottoclasse vuota- empty subclass failure
- ❑ Fede cieca - blind faith
- ❑ Ottimizzazione prematura- premature optimization
- ❑ Programmazione copia e incolla- copy and paste programming
- ❑ Punto di vista ambiguo- ambiguous viewpoint
- ❑ Vicolo cieco- Dead End
- ❑ ...
- ❑ ...