DESIGN PATTERN LEZIONE 1

Davide Caio

INTRODUZIONE

Cos'è l'ingegneria del software?

L’ingegneria del software è la disciplina che si occupa di progettare, sviluppare e mantenere software in modo sistematico, efficiente e scalabile.

- Scrivere codice **funzionante** non basta, il codice deve essere:
  - **Manutenibile** (facile da aggiornare e correggere).
  - **Scalabile** (adattabile a progetti più grandi).
  - **Riutilizzabile** (senza riscrivere tutto da capo).
  - **Chiaramente leggibile** (per chi lo legge dopo di noi).

Obiettivo: evitare il caos e garantire un codice pulito e strutturato.

INTRODUZIONE

Cos'è un design pattern?

Un **design pattern** è una soluzione riutilizzabile a un problema comune nello sviluppo software.

- Non è un codice pronto all’uso, ma una **struttura concettuale** che aiuta a scrivere codice più ordinato e scalabile.
- Sono nati grazie all’esperienza di programmatori che hanno individuato schemi ricorrenti nella progettazione del software.
- **Esempio nella vita reale:** i progetti architettonici. Un ponte ha sempre gli stessi principi strutturali, anche se ogni ponte è diverso.

🎯 **Obiettivo**: evitare di reinventare la ruota e usare soluzioni collaudate.

SOLID

I PRINCIPI SOLID

> **“Il modo migliore per scrivere codice di qualità è mettersi nelle condizioni per scrivere codice di qualità” - Davide Caio**

I principi **SOLID** aiutano a rendere il codice più **modulare, scalabile e facile da mantenere**.

Cosa sono i principi SOLID?

Sono **cinque linee guida** per la programmazione orientata agli oggetti, formulate da **Robert C. Martin** ("Uncle Bob").
L’obiettivo è **evitare il codice fragile e monolitico**, creando software più strutturato e riutilizzabile.

S - Single Responsibility Principle (SRP)

Ogni classe deve avere una sola responsabilità.

✅ **Vantaggio**: il codice è più semplice da mantenere e modificare.

❌ **Errore comune**: una classe che fa troppe cose diventa difficile da aggiornare senza rompere altre parti del codice.

O - Open/Closed Principle (OCP)

O - Open/Closed Principle (OCP)

Il codice deve essere aperto all'estensione ma chiuso alla modifica.
✅ **Vantaggio**: permette di aggiungere funzionalità senza toccare il codice esistente (evitando bug).
❌ **Errore comune**: modificare classi esistenti per aggiungere nuove funzionalità.

> **Esempio**: un sistema di gestione dei nemici che deve supportare nuovi tipi di AI senza modificare il codice base.

L - Liskov Substitution Principle (LSP)

Le sottoclassi devono poter sostituire la classe base senza problemi.
✅ **Vantaggio**: codice più flessibile e meno bug dovuti a comportamenti inaspettati.
❌ **Errore comune**: sottoclassi che modificano il comportamento della classe base in modo inaspettato.

Esempio: una classe `Bird` con il metodo `Fly()`, ma una sottoclasse `Penguin` non può volare.

I - Interface Segregation Principle (ISP)

Meglio tante interfacce piccole che una grande interfaccia generica.

✅ **Vantaggio**: le classi implementano solo ciò di cui hanno bisogno.

❌ **Errore comune**: un'interfaccia con troppi metodi obbliga le classi a implementare cose che non servono.

D - Dependency Inversion Principle (DIP)

Le classi devono dipendere da astrazioni, non da implementazioni concrete.
✅ **Vantaggio**: codice più flessibile e testabile.
❌ **Errore comune**: dipendere direttamente da classi specifiche invece di usare interfacce o astrazioni.

SOLID + Design Pattern = CODICE DI QUALITA!

OBSERVER

Il pattern Observer: cos'è e perché usarlo?

Definizione

Il **pattern Observer** è un pattern **comportamentale** che permette a un oggetto (**Subject**) di notificare automaticamente più oggetti (**Observers**) quando cambia stato, senza creare dipendenze rigide.

Perché è utile?

✅ Disaccoppia i componenti: il Subject non ha bisogno di conoscere direttamente gli Observer.
✅ Evita il codice spaghetti: non dobbiamo aggiornare ogni oggetto manualmente.
✅ Perfetto per eventi in tempo reale: interfacce utente, notifiche di gioco, gestione AI.

🎮 **Esempi nei videogiochi**

- Sistema di **punteggio e UI**: ogni volta che il punteggio cambia, la UI si aggiorna automaticamente.
- **Gestione della salute del giocatore**: quando l'HP cambia, UI, effetti sonori e animazioni reagiscono subito.
- **Notifiche in un quest system**: quando il giocatore raccoglie un oggetto, l'HUD si aggiorna.

Struttura del pattern Observer

Definizione

Il **pattern Observer** è un pattern **comportamentale** che permette a un oggetto (**Subject**) di notificare automaticamente più oggetti (**Observers**) quando cambia stato, senza creare dipendenze rigide.

Perché è utile?

✅ Disaccoppia i componenti: il Subject non ha bisogno di conoscere direttamente gli Observer.
✅ Evita il codice spaghetti: non dobbiamo aggiornare ogni oggetto manualmente.
✅ Perfetto per eventi in tempo reale: interfacce utente, notifiche di gioco, gestione AI.

🎮 **Esempi nei videogiochi**

- Sistema di **punteggio e UI**: ogni volta che il punteggio cambia, la UI si aggiorna automaticamente.
- **Gestione della salute del giocatore**: quando l'HP cambia, UI, effetti sonori e animazioni reagiscono subito.
- **Notifiche in un quest system**: quando il giocatore raccoglie un oggetto, l'HUD si aggiorna.

Vantaggi e svantaggi dell’Observer

✅ Vantaggi

✔️ Disaccoppiamento totale tra Subject e Observer.
✔️ Permette di gestire eventi in modo pulito e scalabile.
✔️ Ottimo per UI, AI e sistemi di eventi complessi.

❌ Svantaggi

❌ Può essere inefficiente se ci sono troppi Observer.
❌ Se un Observer ha un bug, può bloccare l'intera notifica.
❌ Difficile da debuggare se ci sono troppi eventi concorrenti.

Un semplice esempio

Facciamo un esempio per lo score nella UI e nel game controller.

Finito questo esempio, ragioniamo su come potremmo ampliare il sistema.

OBSERVER - 2.0

Un sistema come quello che abbiamo visto prima è molto utile per disaccoppiare classi nello stesso modulo. Ma per notifiche cross-moduli, ci rimane una dipendenza (tra observer e subject).

Come potremmo rimuoverla, per usare questo pattern anche quando vogliamo comunicare tra più moduli?

EVENT MANAGER

✅ Il **Subject** notifica un Event Manager centrale.
✅ L'**Observer** si iscrive all'Event Manager e riceve solo gli eventi che gli interessano.
✅ Nessun collegamento diretto tra Subject e Observer!

Esempio nei videogiochi 🎮

- Quando il giocatore raccoglie una moneta → il sistema di punteggio viene aggiornato, la UI cambia e un suono viene riprodotto.
- Senza Observer: la classe Player dovrebbe conoscere ScoreManager, UI e AudioManager.
- Con Observer: il Player non conosce ScoreManager, UI e AudioManager (ma il contrario si).
- Con Observer - Event Manager: devono solo conoscere l'Event Manager

EVENT MANAGER - VANTAGGI E SVANTAGGI

🎯 **Vantaggi dell’uso di un Event Manager**
✅ **Decoupling**: il **Subject non deve conoscere gli Observer**.
✅ **Facile da estendere**: possiamo aggiungere nuovi Observer senza toccare il codice esistente.
✅ **Codice più pulito**: ogni classe ha una responsabilità chiara.
✅ **Debug più semplice**: possiamo attivare eventi manualmente e vedere chi li ascolta.

❌ **Possibili svantaggi**

- Troppi eventi potrebbero rallentare il gioco (ma possiamo ottimizzarlo con event pooling).
- Più difficile tracciare chi ascolta cosa (ma possiamo loggare gli eventi per debugging)