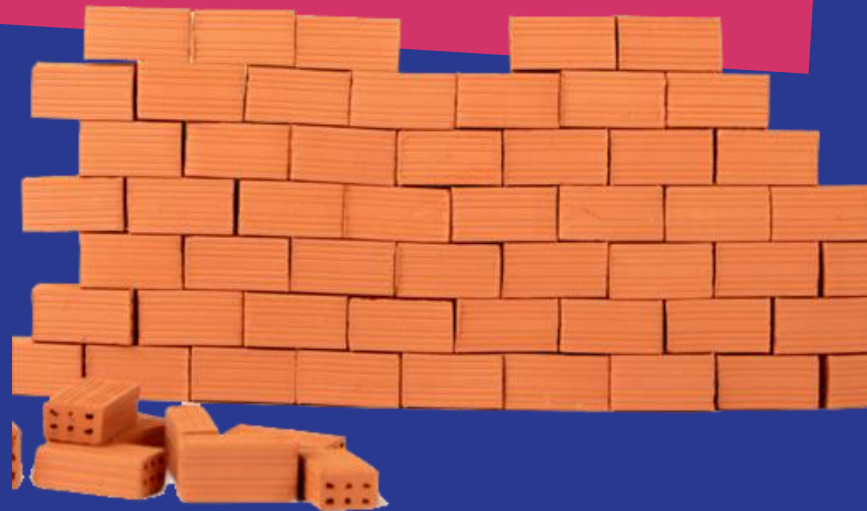


# I principi S.O.L.I.D. con il linguaggio C#

*Webinar*



# Marco Breveglieri

*Sviluppatore software  
consulente e trainer*

👉 <https://www.breveglieri.it>





Una premessa:  
il codice “cattivo”

# Cosa rende il codice “cattivo”?

- Classi che fanno troppe cose
- Metodi troppo lunghi e complessi
- Eccessiva dipendenza da altro codice
- Astrazioni esigenti e non fattorizzate
- Impossibilità di eseguire test automatici
- Gerarchie OOP illogiche



# Coupling

“La misura in cui il tuo codice dipende da altri moduli, e viceversa”.



# Cohesion

“Il grado con cui gli elementi di un modulo possono stare assieme”.



# Cosa succede se...

- Sì “high coupling”?
- No “loose coupling”?
- No “high cohesion”?





Welcome SOLID!



# S.O.L.I.D. è un acronimo



**SRP** Single Responsibility Principle

**OCP** Open/Closed Principle

**LSP** Liskov Substitution Principle

**ISP** Interface Segregation Principle

**DIP** Dependency Inversion Principle

# Quando sono necessari?

Per evitare (o quando si inizia a percepire la presenza) di  
“Code & Design Smell”...

- Codice troppo difficile da modificare
- Codice fragile (facile da “rompere”)
- Codice non riutilizzabile in altre situazioni simili, anche se in contesti differenti
- Eccessivo sforzo nel far svolgere al codice il compito per cui è stato scritto
- Codice che appare inutilmente complicato per lo scopo a cui deve adempiere

# Perché ne parliamo?

- Sono i principi più importanti da seguire nella scrittura del codice (per me)
- Non sono così complessi come possono apparire a prima vista
- Consentono di apportare modifiche al software con minimi sforzi
- Rendono in molti casi la programmazione più... divertente
- Sono l'unico strumento che garantisce la scrittura di codice testabile
- Aprono la strada a buon design del codice, alla possibilità di applicarvi agevolmente il refactoring, e a una infinita serie di altre possibilità

...e come si applicano?



*Usa lo “Sforzo”...*



Nel dettaglio...



# Single Responsibility Principle

*SRP*

# Single Responsibility Principle



*“Ogni modulo software deve avere **una e una sola ragione** per essere modificato”*

# Single Responsibility Principle



*“Ogni classe deve svolgere  
uno e un solo compito”*





*«Vediamo un po' codice...»*

# Recap

- Cercare di individuare i possibili **soggetti** che possono richiedere modifiche future a una classe
- Cercare di individuare i possibili **motivi** che possono richiedere modifiche future a una classe
- Valutare costi e benefici prima di separare due implementazioni
- Tante classi “piccole” sono meglio di poche classi “giganti” (anche per l'IDE)
- Sfruttare la Dependency Injection per l'iniezione di oggetti e dipendenze

# Open/Closed Principle

*OCP*

# Open/Closed Principle

*“Ogni classe deve essere  
**aperta** a estensioni  
ma **chiusa** al cambiamento”*



*«Vediamo un po' codice...»*

# Recap

- Utilizzare **interfacce** o **classi astratte** per creare “punti di estensione”
- Sostituire i costrutti switch { } con oggetti, sfruttando l'ereditarietà
- Non esagerare con la fattorizzazione, ossia
  - Non “esternalizzare” nei minimi dettagli
  - Valutare sempre prima il rapporto costi/benefici
- Utilizzare ove possibile le Collection che fanno uso di Generics

# Liskov Substitution Principle

*LSP*

# Liskov Substitution Principle



*“Let  $q(x)$  be a property provable about  
objects  $x$  of type  $T$ ,  
then  $q(y)$  should be provable for objects  $y$  of  
type  $S$  where  $S$  is a subtype of  $T$  ”*



# Liskov Substitution Principle



*“I sottotipi dovrebbero essere sempre sostituibili ai loro tipi di base ”*

# Liskov Substitution Principle



*“Un client dovrebbe consumare qualsiasi implementazione di un'interfaccia senza che questo modifichi la correttezza del sistema”*



*«Vediamo un po' codice...»*

# Recap

- Evitare di creare gerarchie di classi logicamente inconsistenti
- Non condizionare la logica del codice al tipo specifico di oggetto passato (ad esempio, scrivendo qualcosa del tipo `if (obj is Square)`)
- Prediligere l'uniformità dei comportamenti nelle implementazioni
  - Non restituire `null` se non è tollerato dalla logica principale
  - Non sollevare eccezioni tranne quelle necessarie e previste dalla "business logic"

# Interface Segregation Principle

*ISP*

# Interface Segregation Principle

*“I client non devono essere forzati a dipendere da metodi che non utilizzano”*



*«Vediamo un po' codice...»*

# Recap

- Ricordare che ogni interfaccia (**interface**) rappresenta un “contratto”
- Non creare interfacce cosiddette “asfittiche”, ovvero che costituiscono una mera copia astratta dei metodi della classe che la implementano
- Non aggiungere nuovi metodi alle interfacce quando non necessari
  - Meglio creare nuove interfacce, disegnandole opportunamente
- Non estendere (se possibile) le interfacce esistenti, ma crearne di nuove



# Dependency Inversion Principle

*DIP*

# Dependency Inversion Principle



*“Occorre sempre dipendere da una interfaccia e non da una implementazione ”*

# Liskov Substitution Principle



*“I moduli di alto livello non devono dipendere da moduli di basso livello, ed entrambi devono dipendere da astrazioni ”*

# Liskov Substitution Principle



*“Le astrazioni non devono dipendere dai dettagli, i dettagli devono dipendere dalle astrazioni”*



*«Vediamo un po' codice...»*

# Recap

- Si tratta del principio più importante: applicare questo abilita l'uso degli altri
  - Forza a usare correttamente il principio Open/Closed
  - Permette di separare efficacemente le responsabilità
  - Incentiva l'implementazione corretta dei sottotipi
  - Offre l'opportunità di "segregare" le interfacce

# Conclusioni

# SOLID non è così difficile...

- E' sufficiente vedere i problemi esposti da una prospettiva differente
- Otterremo codice ben scritto e facilmente manutenibile come risultato
- Non dobbiamo preoccuparci se si creano moltissime classi e interfacce
  - Anche le classi più piccole possono essere combinate per creare sistemi complessi
- L'approccio semplifica il testing e la collaborazione tra sviluppatori

*Hai ancora dubbi? Parliamone assieme..* 



## Q & A



# Risorse e approfondimenti

- Pagina (voce) su **Wikipedia** (in inglese)  
<https://en.wikipedia.org/wiki/SOLID>
- Ulteriori **esempi pratici** in linguaggio C#  
<https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/>
- **Dependency Injection** con .NET  
<https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>
- “**Dependency Injection Principles, Practices, and Patterns**” (Libro)  
<https://www.manning.com/books/dependency-injection-principles-practices-patterns>

# Grazie!

