

# Principi SOLID

Anche il codice deve essere agile



# SOLID

Software Development is not a Jenga game

## Che cosa sono?

- **SOLID** è un acronimo che ci aiuta a ricordare cinque principi di software design.
- Si applicano nello specifico alla **programmazione orientata agli oggetti** e si prestano bene ad essere adottati come filosofia di design del software all'interno di un contesto di lavoro *Agile*.

## Per cosa sta l'acronimo SOLID?

- **(S)**ingle responsibility principle
- **(O)**pen/Closed principle
- **(L)**iskov substitution principle
- **(I)**interface segregation principle
- **(D)**ependency inversion principle

- Sono introdotti da Robert C. Martin(Uncle Bob per gli amici), famoso per essere uno degli autori dell'*Agile Manifesto*, nel suo paper *Design principles and design patterns*, del 2000.
- L'acronimo **SOLID** nasce successivamente, coniato da Michael Feathers per identificare un sottoinsieme dei principi di software design trattati da Martin.

Ma...

Perché dovrei preoccuparmi del  
design?

A still from the movie Toy Story showing Woody and Buzz Lightyear. Woody is on the left, looking concerned with a worried expression. Buzz is on the right, sitting in his green and purple space suit, looking excited and gesturing with his right hand. The background is a simple indoor setting with a window showing a night sky with stars.

# Breaking Changes

## Breaking Changes Everywhere

- Nello sviluppo del software, è possibile, anzi **quasi certo** che le specifiche cambino in corso d'opera e che nuove funzionalità debbano essere aggiunte una volta terminato lo sviluppo.
- Il design iniziale potrebbe non aver anticipato alcuni dei cambiamenti richiesti.  
Un buon design deve quindi essere **adattabile ai cambiamenti**.



Il software costruito sulla base di un buon design tende ad avere queste caratteristiche:

- Flessibile
- Facilmente manutenibile
- I componenti sono riusabili
- I componenti sono coesi
- I componenti sono disaccoppiati
- I componenti sono facilmente testabili

In *Design principles and design patterns*, Martin elenca quattro sintomi di “**bad design**”:

- Rigidity
- Fragility
- Immobility
- Viscosity

- Il software è **difficile da modificare**.
- Ogni modifica forza una serie di **modifiche a cascata nei componenti dipendenti** da quello interessato.
- Gli sviluppatori devono inseguire gli effetti della modifica per tutta l'applicazione e diventa problematico **stimare i tempi degli sviluppi**.
- I manager iniziano ad avere timore di allocare tempo per risolvere problemi non critici.

- Il software “**si rompe**” **facilmente** in molti punti quando viene modificato.
- Spesso i problemi derivanti dalle modifiche avvengono in componenti o aree **non concettualmente legate** a quella interessata dalla modifica.
- I manager esitano a permettere di aggiustare qualcosa, con il timore che qualcos'altro smetta di funzionare e perdono fiducia negli sviluppatori.

- E' difficile riutilizzare componenti del software in più parti del progetto.
- I singoli componenti sono **troppo legati ad altri moduli** per poter essere riusabili.
- Si arriva al punto in cui gli sviluppatori realizzano che il rischio e la quantità di lavoro necessari a disaccoppiare le parti necessarie dal resto sono troppo elevati. Si preferisce quindi **riscrivere** anziché **riusare**.

- E' più difficile implementare una modifica in un modo che **rispetta il design** anziché utilizzando delle scorciatoie o “hack”.
- Diventa facile adottare gli approcci “sbagliati” e difficile adottare quelli “giusti”.

- L'ambiente di sviluppo è **lento e inefficiente**.
- Diventa difficile compilare o in generale applicare nuove modifiche al software e permettere rilasci frequenti.
- Gli sviluppatori tendono a prediligere modifiche che non impiegano troppo tempo per essere rilasciate, indipendentemente dal fatto che esse siano o meno adeguate dal punto di vista del design

- Nelle prime fasi dello sviluppo sono trascurabili. In seguito, al crescere del progetto, molto spesso cresce anche il costo necessario a fare una modifica.
- A un certo punto è necessario un “**redesign**”. Ciò spesso non è fattibile a causa del costo in risorse tendenzialmente alto che queste operazioni richiedono. Costa molto meno **prevenire il problema** pensando un design reattivo.



*- “Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.”*

*- “I processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitivamente un ritmo costante”.*

*- “La continua attenzione all'eccellenza tecnica e alla buona progettazione esaltano l'agilità. ”*

*- The Agile manifesto -*

# Principi SOLID in dettaglio

# Single Responsibility Principle(SRP)



## SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

## Single Responsibility principle

*"A class should have one, and only one, reason to change."*

- Reason to change = responsabilità
- Una classe ha tante **ragioni per cambiare** quante sono le sue **responsabilità**.
- Ogni classe deve quindi **avere una ed una sola responsabilità**.

- Il principio si basa sulla **coesione** e sul **disaccoppiamento**.
- Le classi hanno lo scopo raggruppare ciò che **cambia per lo stesso motivo** e separare ciò che **cambia per ragioni diverse**.

- Limita l'impatto dei **cambiamenti** tramite il **disaccoppiamento** e la **coesione**.
- Classi disaccoppiate sono facilmente **riusabili**
- Classi disaccoppiate sono facilmente **testabili**



- Classi denominate “handler”, “manager” e così via. Nomi generici indicano che non è chiaro quale sia la responsabilità della classe.
- Classi che includono la logica per salvarsi su un database o per essere “scrivere sé stesse”.

- Applicare l' SRP **non** vuol dire atomizzare ogni componente in tante piccole classi, bensì raggruppare cosa cambia per la stessa ragione e separare cosa non lo fa.

# Open/Closed Principle(OCP)



## Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

*“Software entities (classes, modules, functions, etc.)  
should be open for extension, but closed for modification.”*

- Bisogna essere in grado di estendere il comportamento di una classe senza modificarla.
- Ciò è possibile grazie al **polimorfismo**, elemento chiave della OOP e su cui si basa l'OCP.

- L'utilizzo di **interfacce** aggiunge livelli di astrazione, riducendo l'**accoppiamento**.
- Diminuisce il rischio di introdurre bug nei componenti esistenti e in quelli dipendenti da essi.
- E' necessario testare e/o deployare soltanto i nuovi componenti, senza preoccuparsi di quelli già esistenti.

- A volte i moduli che abbiamo sono semplicemente impossibili da estendere.
- Non c'è nulla di male nel modificare i componenti esistenti in questo caso. Cercando comunque di introdurre le modifiche in modo più isolato possibile all'interno del componente.



# Liskov Substitution Principle(LSP)



If it looks like a duck and quacks like a duck but it needs batteries, you probably have the wrong abstraction.

# Liskov Substitution Principle

*"Let  $\Phi(x)$  be a property provable about objects  $x$  of type  $T$ .  
Then  $\Phi(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is  
a subtype of  $T$ ."*

*"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."*

*Robert C. Martin*

In pratica: le classi derivate devono essere **completamente sostituibili** con le classi base da cui derivano.

- Estensione naturale dell'OCP.
- Fa in modo che **l'ereditarietà** venga applicata in modo corretto, stabilendo quando due entità devono o non devono avere una relazione **“is a”**.

- -Le **precondizioni** di un metodo di una classe derivata non possono essere **più restrittive** della classe base.
- -Le **postcondizioni** di un metodo derivato non possono essere **meno restrittive** della classe base.
- -Le **invarianti** della classe base devono essere rispettate nella classe derivata.
- -Rispettare l' "history constraint".
- -Il tipo derivato non può lanciare eccezioni che non siano derivate dalle eccezioni della classe base.

- Una relazione “is a” corretta da un punto di vista logico potrebbe non rispettare l’LSP.
- Esempio: matematicamente parlando un quadrato è un rettangolo. Tuttavia, nella OOP ciò potrebbe non essere vero...

# Interface Segregation Principle(ISP)





## Interface Segregation Principle

If IRequireFood, I want to Eat(Food food) not,  
LightCandelabra() or LayoutCutlery(CutleryLayout preferredLayout)

## Interface Segregation Principle

*“Clients should not be forced to depend upon interfaces that they do not use.”*

- Interfacce “*polluted*” o “*fat*”.
- Se ci sono componenti che implementano **interfacce complesse con molti metodi**, spesso capita di dover implementare in modo “*dummy*” i metodi che non sono necessari ai componenti in questione.

Applicare l'ISP:

- Dividere le interfacce con molti metodi in **più piccole interfacce specializzate**, ciascuna che raggruppa metodi concettualmente legati tra loro.
- Non forzare i “*client*” ad essere accoppiati a metodi che non utilizzano.

- Come nel caso dell' SRP, evitare la frammentazione eccessiva delle interfacce.

Le interfacce devono essere **coese**, quindi raggruppare metodi che sono concettualmente collegati.

# Dependency Inversion Principle(DIP)



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

*High-level modules should not depend on low-level modules. Both should depend on abstractions.*

*Abstractions should not depend on details. Details should depend on abstractions.*



- *Modulo di basso livello*: modulo con poche dipendenze
- *Modulo di alto livello*: modulo con molte dipendenze
- *Astrazioni*: interfacce
- *Dettagli*: moduli concreti

- I moduli di alto livello non devono essere obbligati a cambiare in seguito a una modifica nei moduli di basso livello.
- Non dipendere da componenti concreti, ma da astrazioni.

In pratica:

L'interazione tra moduli di alto e basso livello deve avvenire solo tramite **astrazioni**.

- I moduli diventano **disaccoppiati**.
- Se voglio utilizzare un modulo concreto al posto di un altro non ho bisogno di modificare il software in molti punti, ma solo dove il modulo concreto viene effettivamente istanziato.

- Il DIP non è la **dependency injection**.
- La DI è soltanto il modo più comune di rispettare il DIP, ma non è l'unico possibile.
- Da qualche parte il software dovrà comunque istanziare moduli concreti, il DIP non permette di evitare ciò, ma di centralizzarlo.

- *Clean Code (Robert C. Martin)*
- *Clean Architecture (Robert C. Martin)*
- *Design Principles and Design Patterns (Robert C. Martin)*
- *Adaptive Code: Agile Coding with Design Patterns and SOLID Principles (Gary McLean Hall)*



GRAZIE!