

---

# *Webinar:* **I principi SOLID**

Capire e padroneggiare i principi SOLID in Delphi

# Marco Breveglieri

*Software and Web Developer,  
Teacher and Consultant*

@ABLS Team - Reggio Emilia, ITALY

Homepage

<https://www.breveglieri.it>

Blog tecnico

<https://www.compilaquindiva.com>

Delphi Podcast

<https://www.delhipodcast.com>

Twitter

@mbreveglieri



# 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 ingiustificate



---

## Coupling

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



---

## High vs Loose Coupling

### High Coupling

- Moduli difficilmente separabili
- Moduli difficilmente sostituibili
- Implementazioni intrecciate tra loro
- Modifiche a un punto del codice possono produrre effetti devastanti in un altro
- Relazioni tra i moduli stessi meno comprensibili



### Loose Coupling

- Leggibilità più alta del codice
- Comprensione agevolata
- Riusabile perché connesso da strati più sottili (es. interfacce)
- Manutenibile (perché le modifiche sono isolate)
- Testabile!



---

## Cohesion

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



## High Cohesion

Si raggiunge quando le singole parti del sistema, sebbene diverse e con caratteristiche e funzionalità eterogenee, collaborano tra loro realizzando qualcosa di utile (ad alto valore per il cliente o per lo sviluppatore).



—  
**...altrimenti...**



# Introduzione



## 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?

Quando si percepisce la presenza di  
“Code & Design Smell”...

- Codice troppo difficile da modificare
- Codice facile da “rompere” al minimo tocco
- Codice non riutilizzabile in altre situazioni simili ma in contesti differenti
- Eccessivo sforzo nel far fare al codice il proprio compito
- Codice inutilmente complicato per il proprio scopo

---

## Perché ne parliamo?

- Principi più importanti da seguire nella scrittura del codice
- Non sono così complessi come possono apparire a prima vista
- Consentono di apportare modifiche al software con minimi sforzi (in barba ai clienti)
- Rendono in molti casi la programmazione... più divertente
- Sono l'unico strumento che permette di scrivere codice "testabile"
- Aprono la strada a refactoring, buon design del codice e infinite serie di possibilità

## Come applicarlo?



Katerina Borodina  
@ctrlshifti

SOLID code? no, my code is LIQUID: Low In Quality,  
Unrivaled In Despair

### ***Tecnica SDD\****

\*Schwartz Driven Development  
(usa lo Sforzo!)



# I principi S.O.L.I.D.



---

# Single Responsibility Principle

## *SRP*

---

## Single Responsibility Principle



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

---

## Single Responsibility Principle



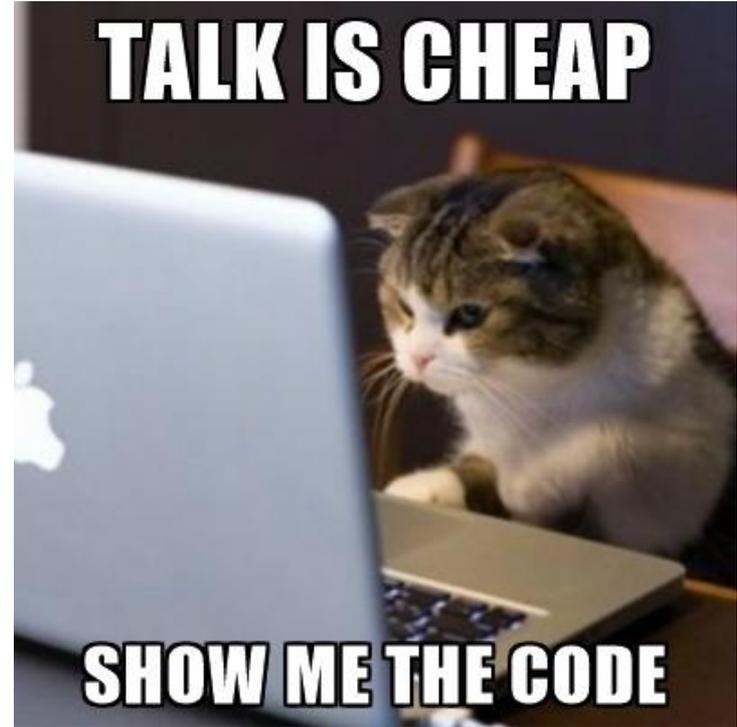
*“Ogni classe deve svolgere  
uno e un solo compito”*

---

## Demo



Vediamo un esempio “errato”...

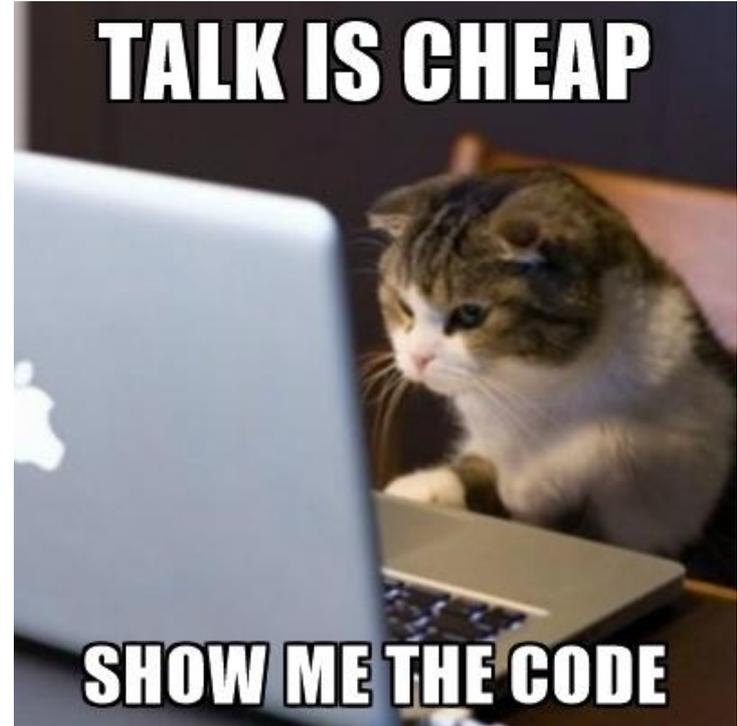


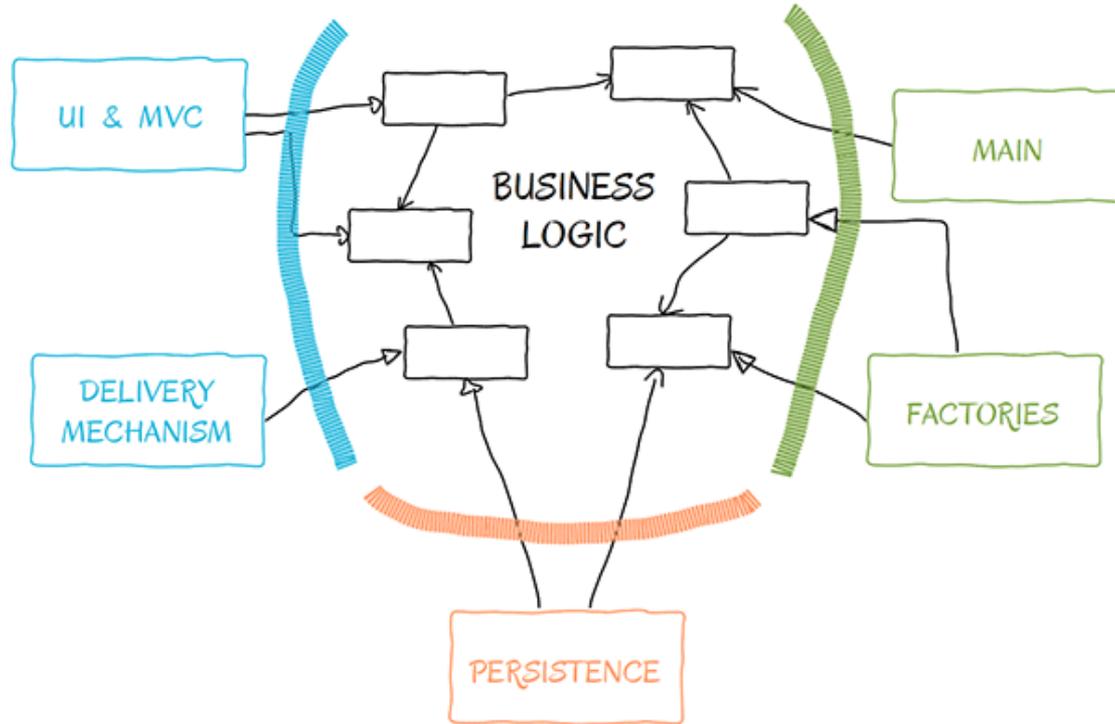
---

## Demo



Vediamo un esempio corretto...





Come si presenta il modello



## Recap

- Cercare di individuare i possibili soggetti che possono richiedere modifiche a una classe
- Cercare di individuare i possibili motivi che possono richiedere modifiche 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 degli oggetti

---

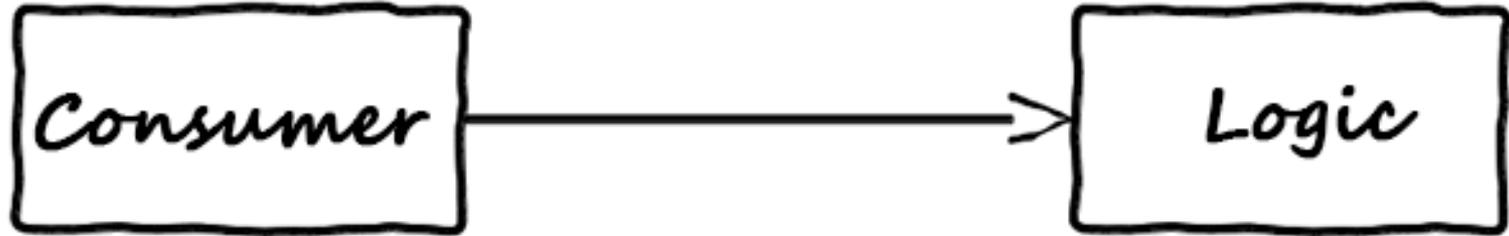
# Open/Closed Principle

## OCP

---

## Open Closed Principle

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



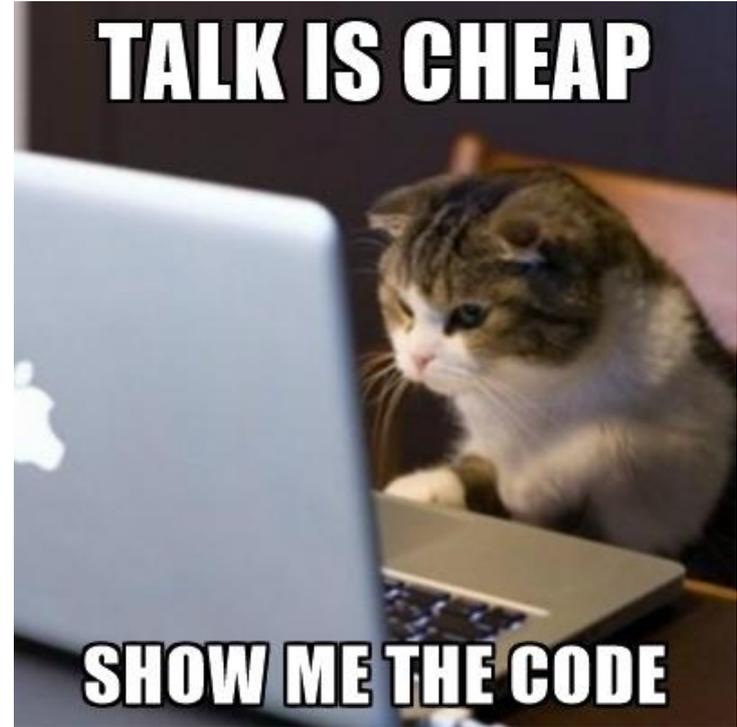
Lo scenario

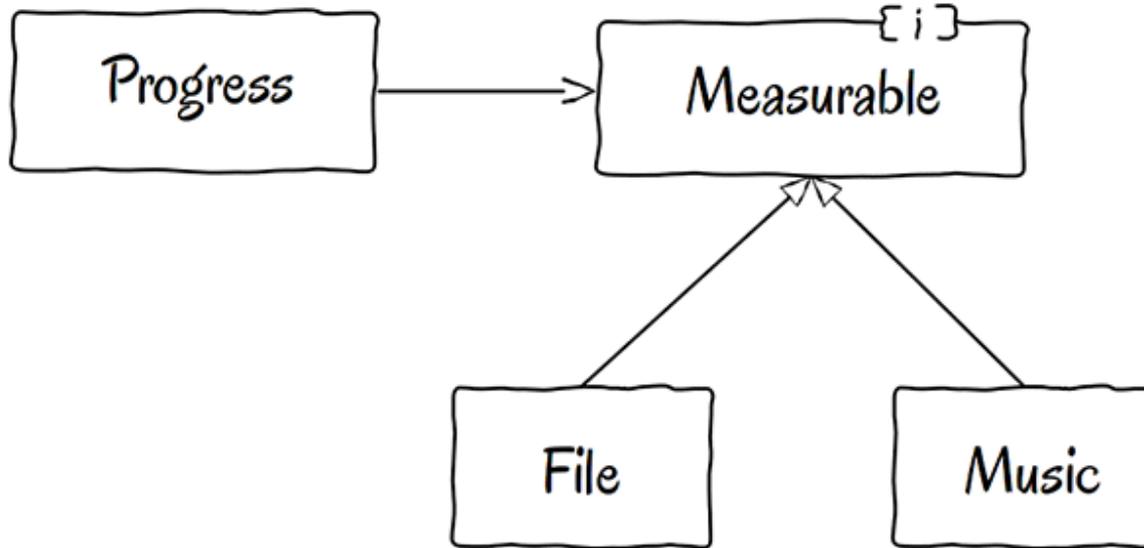
---

## Demo



Vediamo un esempio “errato”...





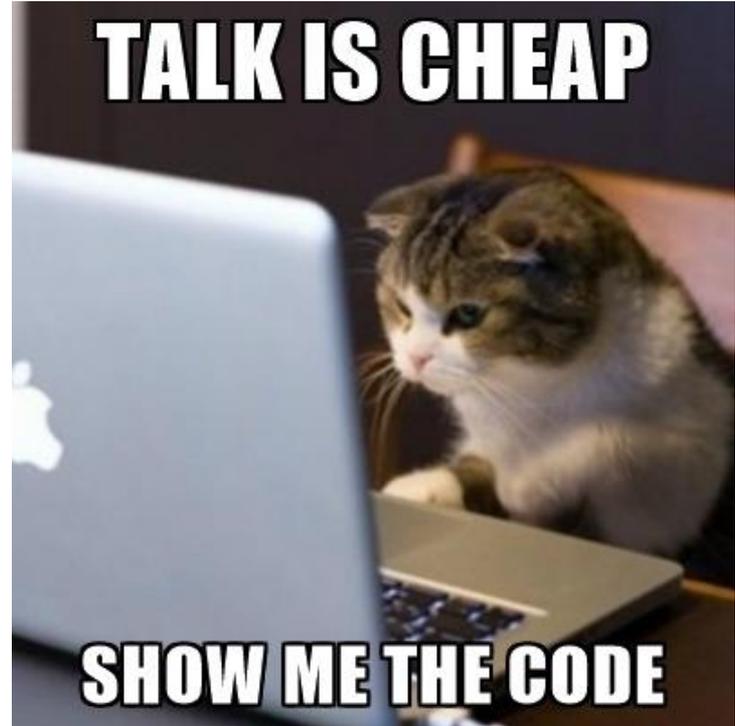
Applichiamo lo “Strategy Pattern”

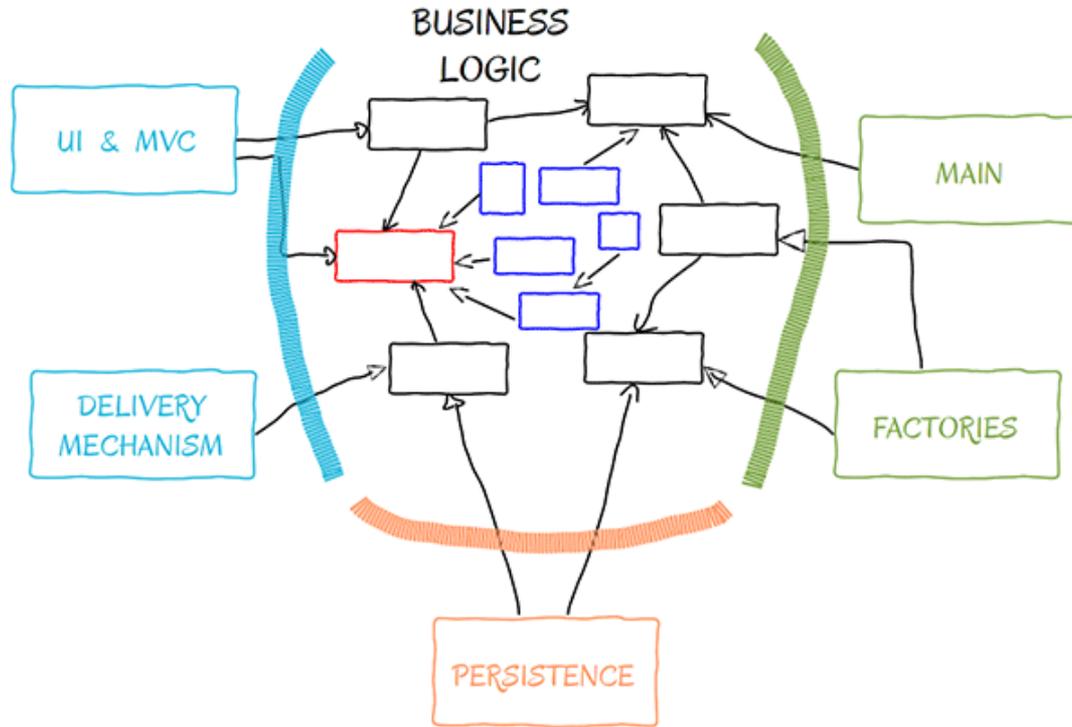
---

## Demo



Vediamo un esempio corretto...





Come evolve il modello



## Recap

- Utilizzare interfacce o classi astratte per creare “punti di estensione”
- Sostituire i costrutti `case...of` con oggetti esterni sfruttando ereditarietà
- Non esagerare con l’espansione, ovvero non esternalizzare fino ai minimi dettagli  
(valutare sempre prima i costi e i benefici)
- Se occorre referenziare una lista di oggetti esterni, utilizzare le classi “Container” con Generics

---

# Liskov Substitution Principle

## *LSP*

---

## Liskov's 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's Substitution Principle



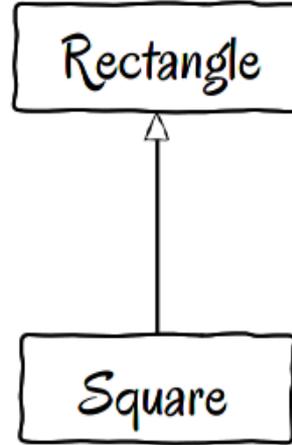
*“I sottotipi dovrebbero essere sempre sostituibili ai loro tipi di base”*

---

## Liskov's Substitution Principle



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



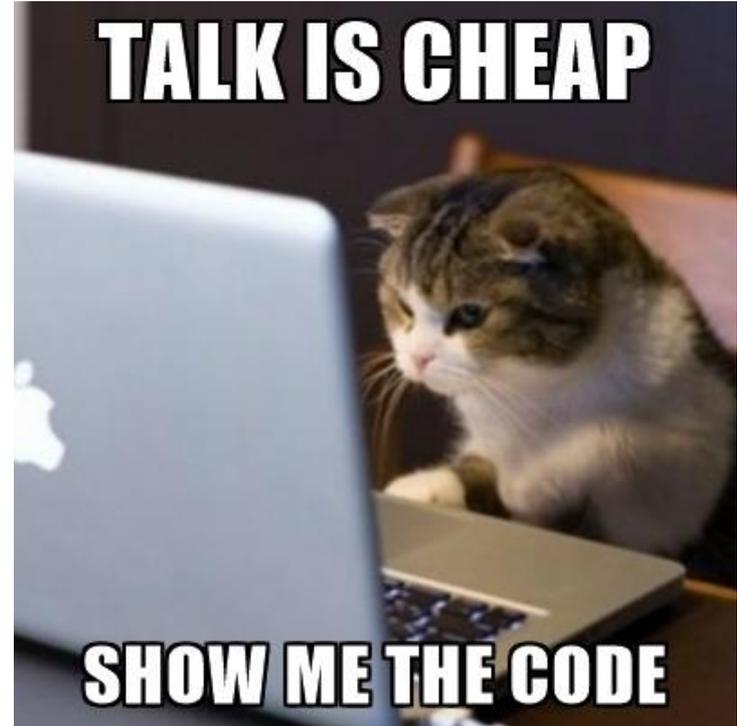
LSP Violation: l'esempio classico

---

## Demo



Vediamo un esempio “errato”...

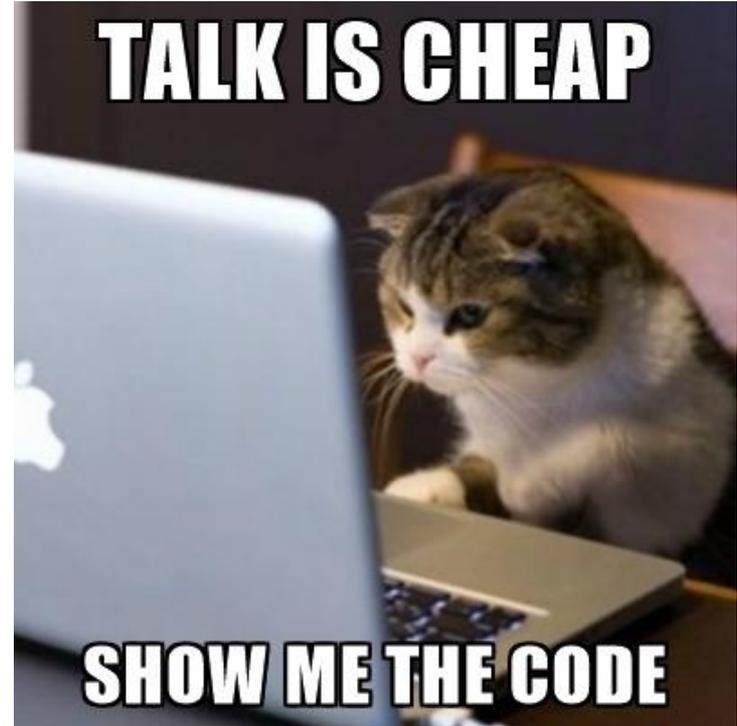


---

## Demo



Vediamo un esempio corretto...





## Recap

- Evitare gerarchie di classi logicamente inconsistenti
- Non condizionare la logica al tipo di oggetto passato (es. `if AObj is TSquare...`)
- Forzare uniformità nei comportamenti
  - Non restituire `nil` se non previsto nella logica principale
  - Non sollevare eccezioni tranne quelle lecite per la business logic

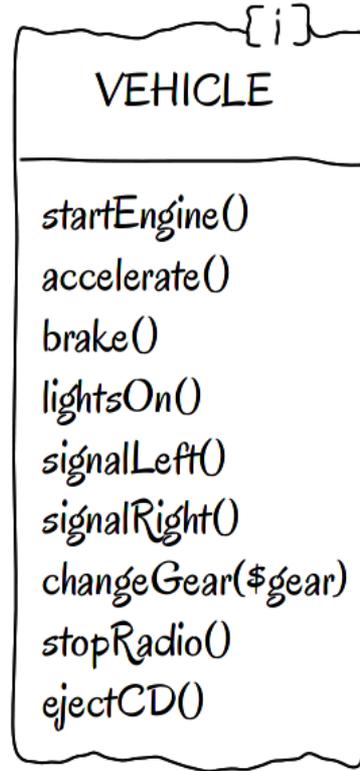
---

# **I**nterface Segregation Principle *ISP*

---

## Interface Segregation Principle

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



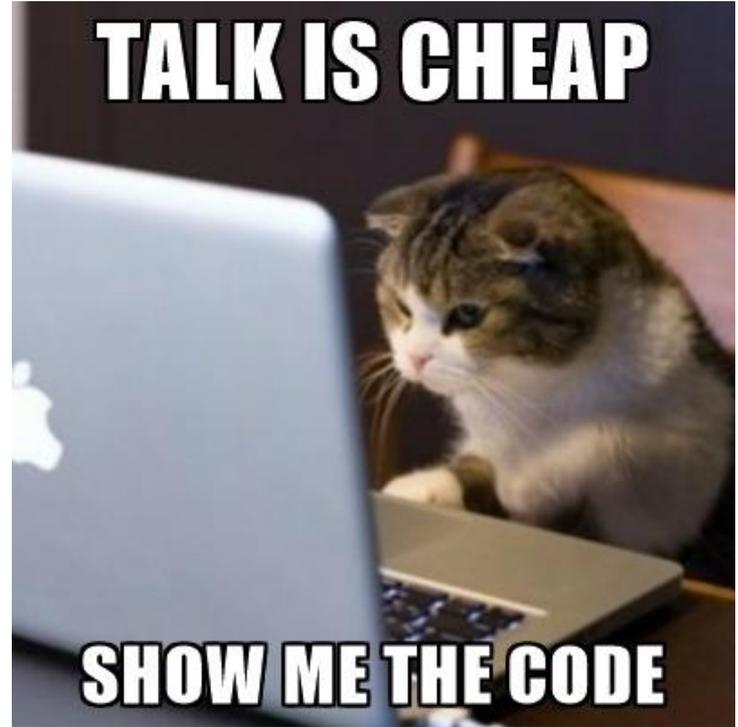
ISP Violation: un esempio

---

## Demo



Vediamo un esempio “errato”...

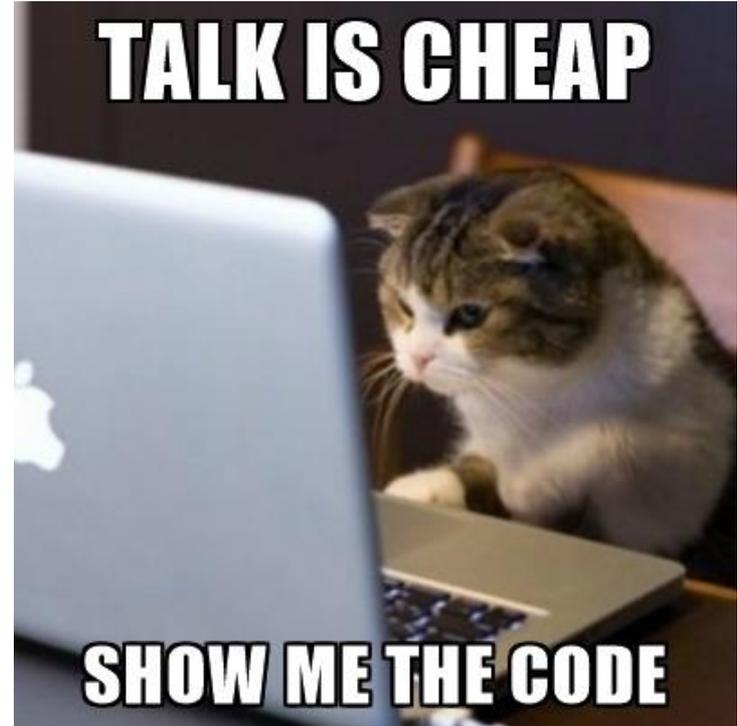


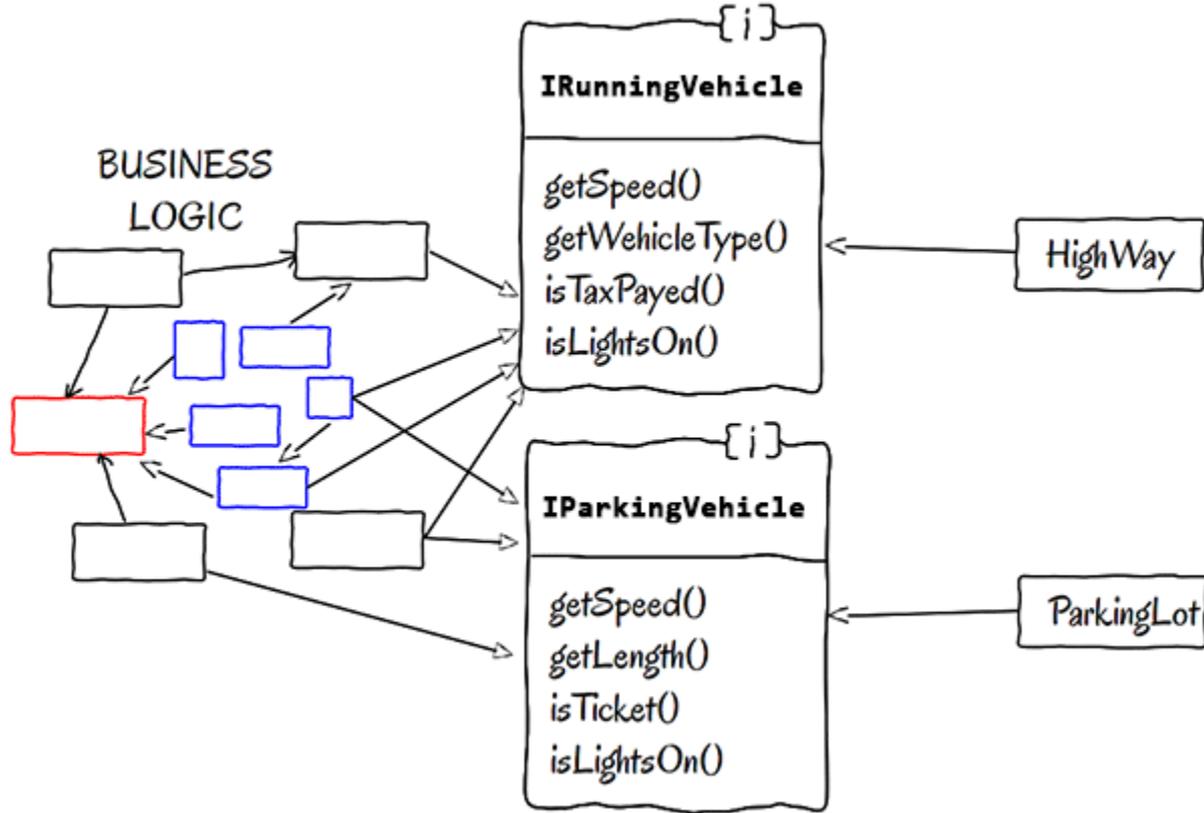
---

## Demo



Vediamo un esempio corretto...





ISP Violation: un esempio

---

## Recap

- Ricordare che ogni interfaccia (interface) rappresenta un contratto
- Non creare interfacce cosiddette “asfittiche” (copia astratta della classe)
- Non aggiungere metodi non necessari alle interfacce ma disegnarle opportunamente
- Non estendere (se possibile) le interfacce esistenti, ma crearne di nuove

---

# **D**ependency Inversion Principle

## *DIP*

---

## Dependency Inversion Principle



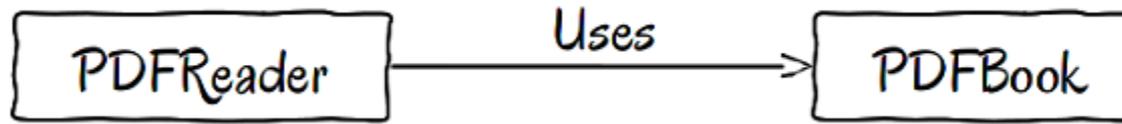
*“Occorre sempre dipendere da una interfaccia e non da una implementazione”*

---

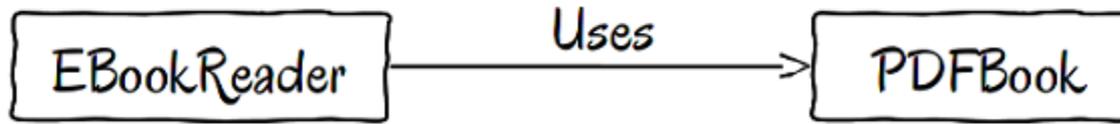
## Dependency Inversion Principle



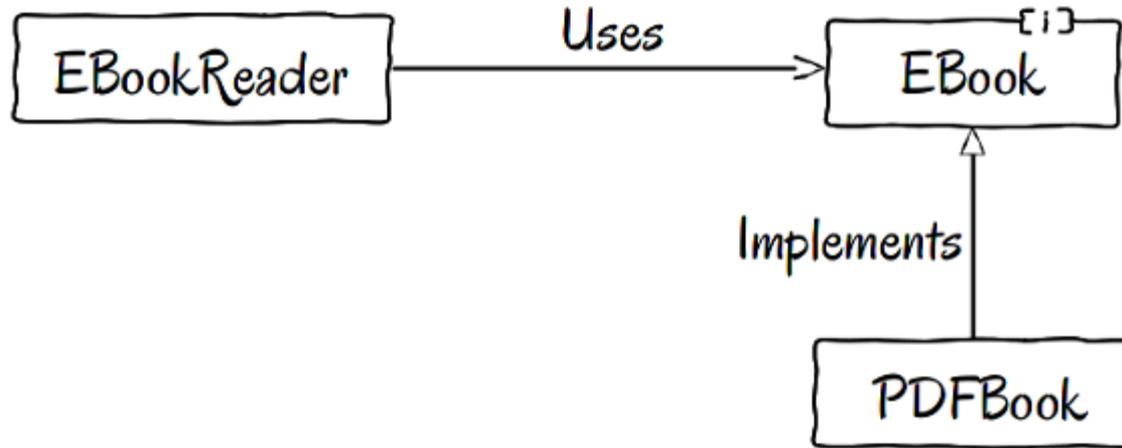
- A. *Moduli di alto livello non devono dipendere da moduli di basso livello, ed entrambi devono dipendere da astrazioni.*
- B. *Le astrazioni non devono dipendere dai dettagli. I dettagli devono dipendere dalle astrazioni.*



Consideriamo questo scenario iniziale.



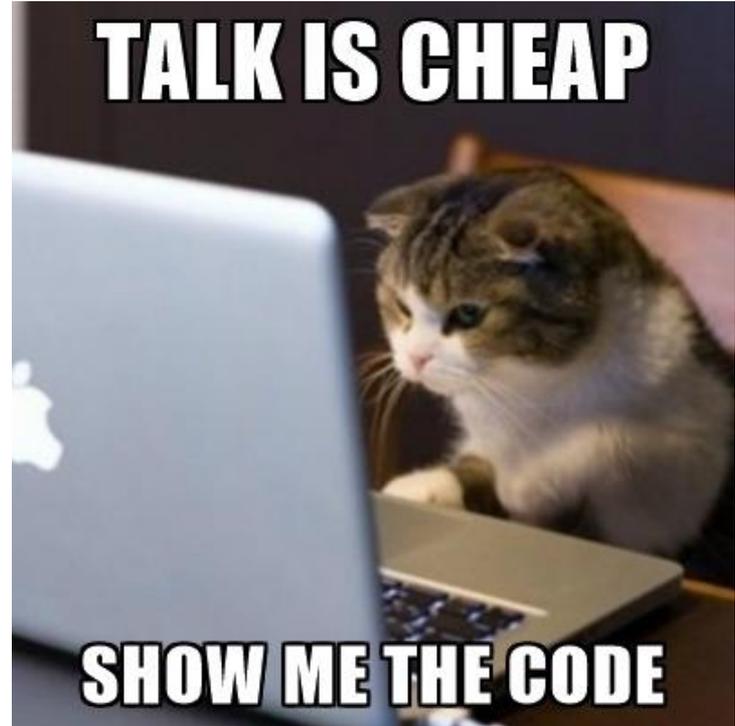
Nessun cambiamento funzionale, ma effetto di design ben visibile:  
il Reader diventa più astratto, più generico, ma usa un tipo di Book molto specifico (PdfBook).



Soluzione: introduciamo un'ulteriore astrazione (*EBook*) applicando il principio ISP.



## Demo





## Recap

- Applicare il DIP automaticamente consente di abilitare l'uso degli altri principi
- ti forza quasi a usare correttamente il principio Open/Closed;
  - ti permette di separare le responsabilità
  - incentiva l'implementazione corretta dei sottotipi
  - offre l'opportunità di segregare le interfacce

# Conclusioni



## S.O.L.I.D. non è così difficile...

- E' sufficiente vedere i problemi esposti da una prospettiva differente nella stesura del codice
- Avremo come risultato codice ben scritto e facilmente manutenibile
- Non dobbiamo preoccuparci se avremo molte classi e interfacce: non c'è un limite
- Classi più piccole e mirate possono essere combinate più facilmente per creare sistemi complessi
- L'approccio semplifica il testing e la collaborazione tra sviluppatori

Non è male, no? 😊

  
**A full demo**



# Questions?

Domande?



# Thanks!

---

#### Riferimenti e fonti

Unsplash - Beautiful Free Images & Pictures (<https://unsplash.com/>)

Meme Generator (<https://imgflip.com/memegenerator>)

EnvatoTuts+ (<https://code.tutsplus.com/series/the-solid-principles--cms-634>)