Best Practice

Design Pattern – SOLID Principle

Academy

GRUPPO EURIS

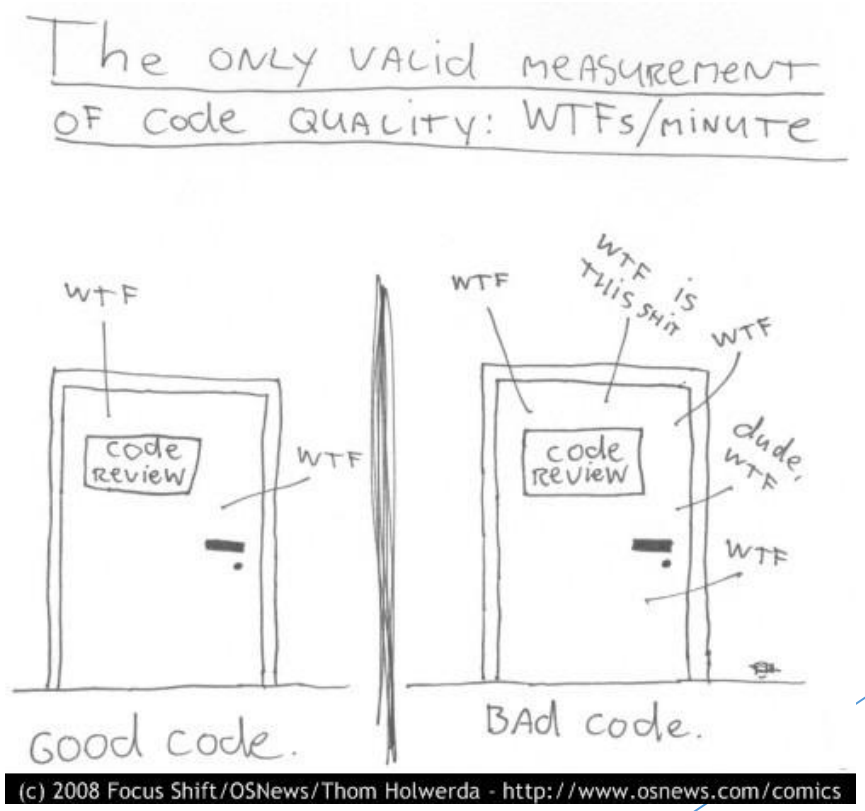
Di cosa abbiamo bisogno...

- Visual Studio 2019
- .NET Core 3.1+
- Pazienza ... e tanta attenzione !! ☺ ☺

Perchè «Best Practice»?

Il bravo programmatore non sa scrivere codice ma sa scrivere BUON codice!

Cosa vuol dire scrivere buon codice?



GRUPPO EURIS

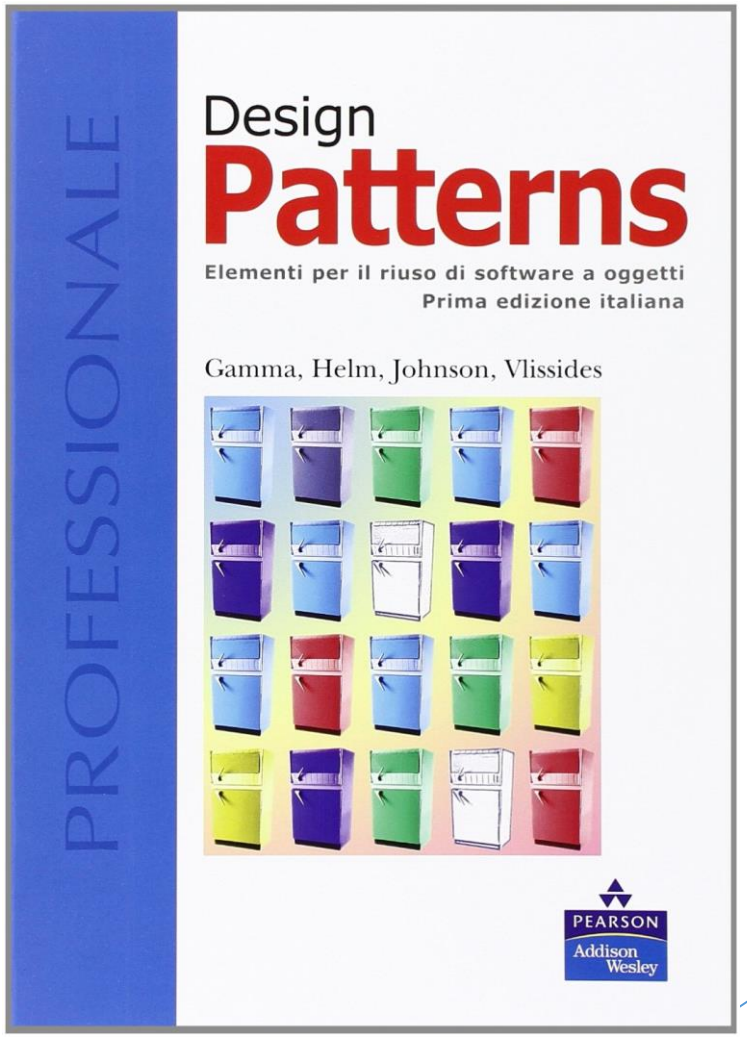
Approcci alle Best Practice

- **Approccio Clean Code:**
  - Non li tratteremo in questo corso ma vi consiglio un'ottima lettura: Clean Code
- **Approcci Architetturali:**
  - **Design Pattern**
  - **Principi Solid**

Design Pattern

Design Pattern - Indice

- Cos'è un Pattern
- Scopo dei Pattern
- Definizione
- Tipologia dei design pattern
  - Creazionali
    - Factory Method
  - Strutturali
    - Decorator
  - Comportamentali
    - Chain of Responsability
- Qualche esempio pratico



GRUPPO EURIS

Design Pattern – Cos'è un Pattern

- È un'IDEA, uno schema GENERALE E RIUSABILE
- NON un **componente** riusabile perchè
  - non è un *oggetto* fisico
  - non può essere usato così come è stato definito, ma deve essere contestualizzato all'interno del particolare problema applicativo

Design Pattern – Scopo dei Patterns

- Catturare l'esperienza degli esperti
- Evitare di reinventare ogni volta le stesse cose
- Cosa fornisce un design pattern al progettista software?
  - Una soluzione codificata e consolidata per un problema ricorrente
  - Un'astrazione di granularità e livello di astrazione più elevati di una classe
  - Un supporto alla comunicazione delle caratteristiche del progetto
  - Un modo per progettare software con caratteristiche predefinite
  - Un supporto alla progettazione di sistemi complessi

Design Pattern – Definizione

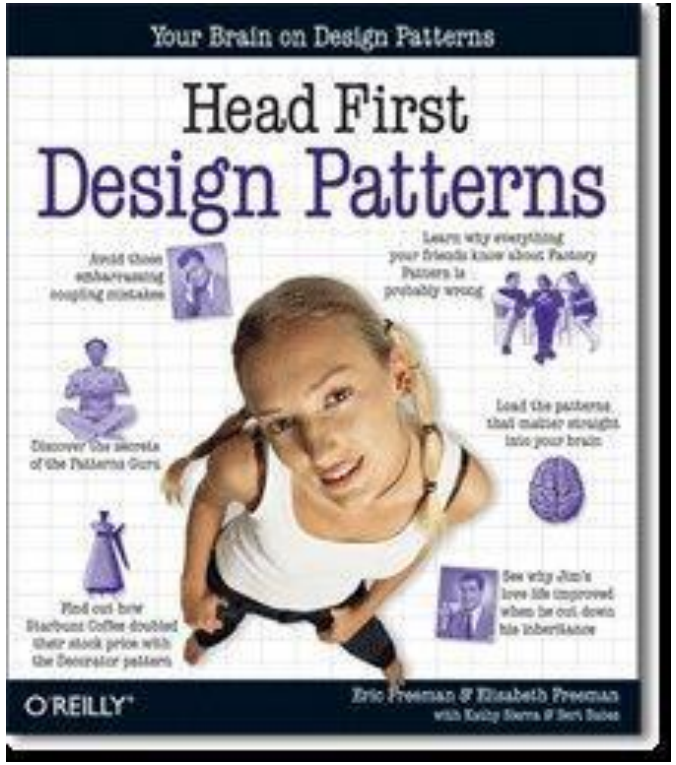
- **Ogni pattern descrive un problema specifico che ricorre più volte e descrive il nucleo della soluzione a quel problema, in modo da poter utilizzare tale soluzione un milione di volte, senza mai farlo allo stesso modo.**

- Un pattern è formato da quattro elementi essenziali:

1. Il **nome** del pattern, è utile per descrivere la sua funzionalità in una o due parole.
2. Il **problema** nel quale il pattern è applicabile. Spiega il problema e il contesto, a volte descrive dei problemi specifici del design mentre a volte può descrivere strutture di classi e oggetti. Può anche includere una lista di condizioni che devono essere soddisfatte precedentemente perché il pattern possa essere applicato.
3. La **soluzione** che descrive in modo astratto come il pattern risolve il problema. Descrive gli elementi che compongono il design, le loro responsabilità e le collaborazioni.
4. Le **conseguenze** portate dall'applicazione del pattern. Spesso sono tralasciate ma sono importanti per poter valutare i costi-benefici dell'utilizzo del pattern.

Design Pattern – Definizione

- **Nome** e classificazione del pattern
- **Sinonimi**: altri nomi del pattern
- **Scopo**: cosa fa il pattern? a cosa serve?
- **Motivazione**: scenario che illustra un design problem
- **Applicabilità**: situazioni in cui si applica il pattern
- **Struttura**: rappresentazione delle classi in stile OMT
- **Partecipanti**: classi e oggetti inclusi nel pattern
- **Collaborazioni**: come i partecipanti collaborano
- **Conseguenze**: come consegue i suoi obiettivi il pattern?
- **Implementazione**: che tecniche di codifica sono necessarie?
- **Codice di esempio:** scritto in un linguaggio a oggetti
- **Usi noti:** esempi d'applicazione del pattern in sistemi reali
- **Pattern correlati:** con quali altri pattern si dovrebbe usare?



Design Pattern – Tipologia di Design Patterns

Esistono diverse categorie di pattern, che descrivono la funzione (purpose) e il dominio (scope) del pattern.

- Funzione (purpose), ovvero cosa fa il pattern:
  - Creazionali (**creational**): forniscono meccanismi per la creazione di oggetti
  - Strutturali (**structural**): gestiscono la separazione tra interfaccia e implementazione e le modalità di composizione tra oggetti
  - Comportamentali (**behavioral**): consentono la modifica del comportamento degli oggetti

Design Pattern – Creazionali

- I pattern di questa categoria sono dedicati alla composizione di classi e oggetti per creare delle strutture più grandi.

- È possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti.

Design Pattern – Creazionali

Nome	Descrizione
Builder	Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo da poter usare lo stesso processo di costruzione per altre rappresentazioni
Abstract Factory	Provvede ad un interfaccia per creare famiglie di oggetti in relazione senza specificare le loro classi concrete
Factory Method	Definisce un interfaccia per creare un oggetto ma lascia decidere alle sottoclassi quale classe istanziare
Prototype	Specifica il tipo di oggetto da creare usando un istanza prototipo e crea nuovi oggetti copiando questo prototipo
Singleton	Assicura che la classe abbia una sola istanza e provvede un modo di accesso

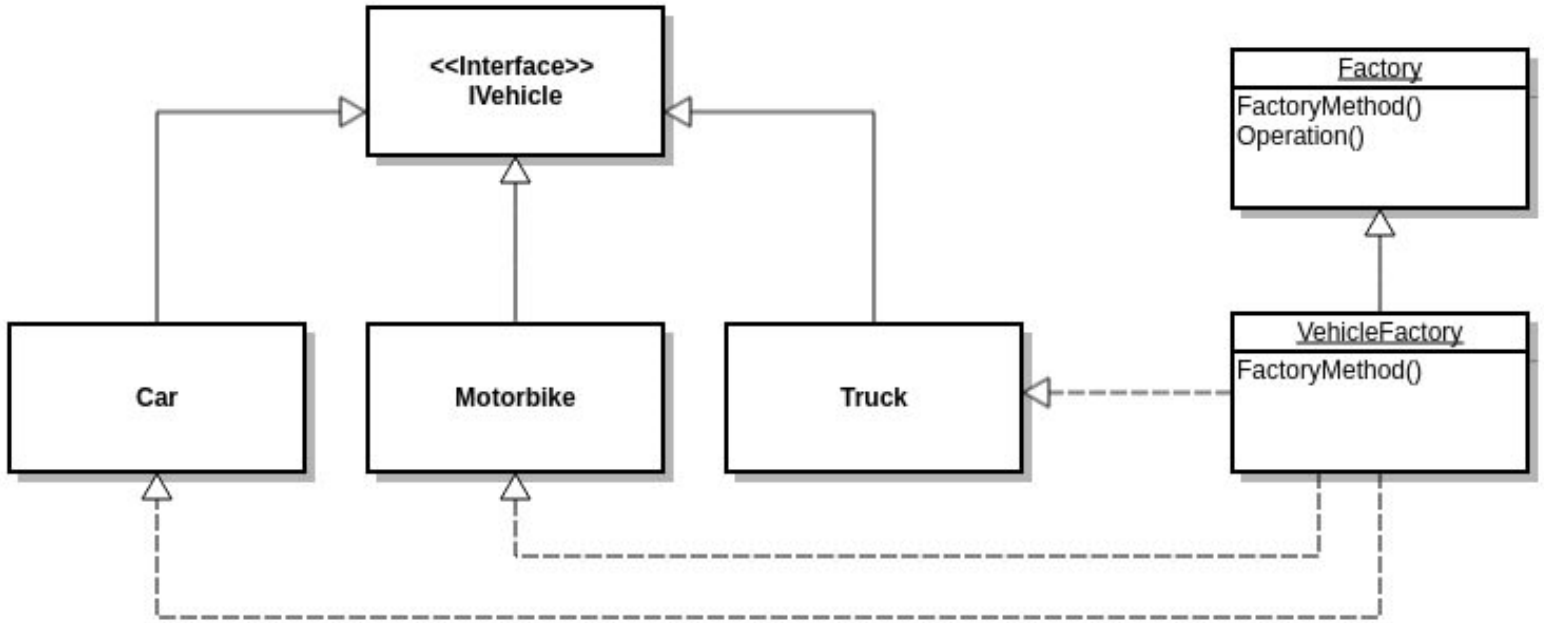
Design Pattern – Factory Method

Esigenza: vogliamo costruire un veicolo sulla base del numero di ruote richieste.

Caso d'uso:

- vogliamo un mezzo a 4 ruote -> macchina
- vogliamo un mezzo a 2 ruote -> motocicletta
- vogliamo un mezzo da 6 ruote in sù -> Camion

Design Pattern – Factory Method



GRUPPO EURIS

Design Pattern – Factory Method

```
public interface IVehicle
{
}
public class Car: IVehicle
{
}
public class Motorbike: IVehicle
{
}
public class Truck: IVehicle
{
}
```

Design Pattern – Factory Method

```
4 references | 0 changes | 0 authors, 0 changes
public class VehicleFactory
{
    4 references | 0 changes | 0 authors, 0 changes
    public static IVehicle Build(int numberOfWheels, int cilindrata) => numberOfWheels switch
    {
        2 => cilindrata > 125 ? new Motorbike(cilindrata) : new Scooter(cilindrata),
        4 => new Car(cilindrata),
        6 => new Truck(cilindrata),
        _ => throw new NumberOfWheelsNotSupported()
    };
}
```

Spunti:

- E se volessimo gestire la differenza tra Camion e TIR?
- E i sidecar?
- E la differenza tra motocicletta e scooter?

Link al codice completo

GRUPPO EURIS

Design Pattern – Strutturali

- I pattern di questa categoria sono dedicati alla composizione di classi e oggetti per creare delle strutture più grandi.
- È possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti.
- Ad esempio permettono di far funzionare insieme delle librerie indipendenti.

Design Pattern – Strutturali

Nome	Descrizione
Adapter	Converte l'interfaccia di una classe in un'altra permettendo a due classi di lavorare assieme anche se hanno interfacce diverse.
Bridge	Disaccoppia un'astrazione dalla sua implementazione in modo che possano variare in modo indipendente.
Decorator	Aggiunge nuove responsabilità ad un oggetto in modo dinamico, è un alternativa alle sottoclassi per estendere le funzionalità
Composite	Compone oggetti in strutture ad albero per implementare delle composizioni ricorsive
Facade	Provvede un interfaccia unificata per le interfacce di un sottosistema in modo da rendere più facile il loro utilizzo

Design Pattern – Strutturali

Nome	Descrizione
Facade	Provvede un interfaccia unificata per le interfacce di un sottosistema in modo da rendere più facile il loro utilizzo
Proxy	Provvede un surrogato di un oggetto per controllarne gli accessi
Flyweight	Usa la condivisione per supportare in modo efficiente un gran numero di oggetti con fine granularità

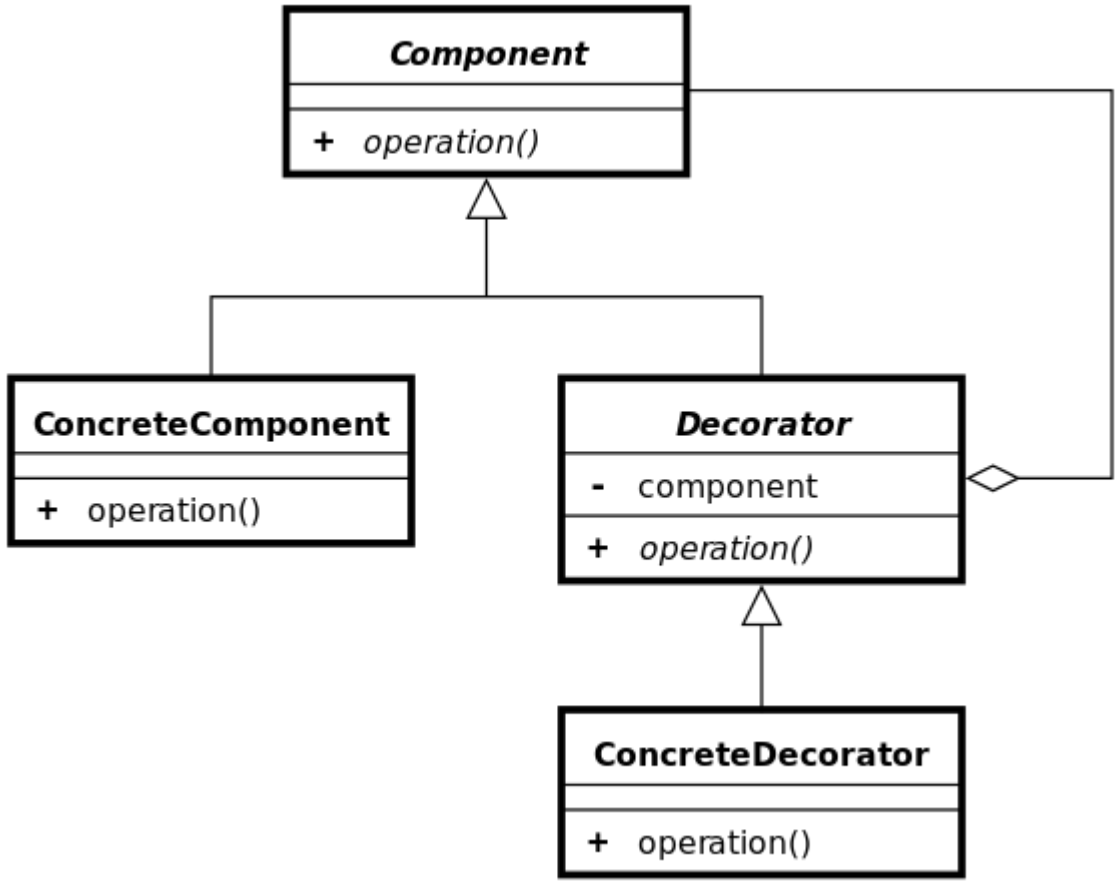
Design Pattern – Decorator

Esigenza: vogliamo gestire il software di una gelateria che permetta di realizzare sia gelati semplici, sia gelati con il topping o altri ingredienti.

Caso d'uso:

- vogliamo un gelato semplice
- vogliamo un gelato con le noccioline sopra
- vogliamo un gelato al miele

Design Pattern – Decorator



GRUPPO EURIS

Design Pattern – Decorator

```
public abstract class IceCream
{
  public abstract MakeIceCream();
}
public class SimpleIceCream: IceCream
{
  public override string MakeIceCream() => ‘’base Icecream’’;
}
```

Design Pattern – Decorator

```
public abstract class IceCreamDecorator: IceCream
{
    private readonly _icecream;

    public string IceCreamDecorator(IceCream icecream)
    {
        _icecream = icecream;
    }
   public override string MakeIceCream() => _icecream.MakeIceCream()
}
```

Design Pattern – Decorator

```
public class HoneyIceCream: IceCreamDecorator
{
    public string HoneyIceCream(IceCream icecream): base(icecream)
    {
        _icecream = icecream;
    }

    public override string MakeIceCream() => $''{_icecream.MakeIceCream()} with Honey'';
}
```

Link al codice completo

Design Pattern – Comportamentali

- Questi pattern sono dedicati all'assegnamento di responsabilità tra gli oggetti e alla creazione di algoritmi.
- Una caratteristica comune in questi pattern è il supporto per seguire le comunicazioni che avvengono tra le classi.
- L'utilizzo di questi pattern permette di dedicarsi principalmente alle connessioni tra oggetti lasciando in disparte la gestione dei flussi di controllo.

Design Pattern – Comportamentali

Nome	Descrizione
Chain of Responsability	Evita l'accoppiamento di chi manda una richiesta con chi la riceve dando a più oggetti la possibilità di maneggiare la richiesta.
Command	Incapsula una richiesta in un oggetto in modo da poter eseguire operazioni che non si potrebbero eseguire.
Interpreter	Dato un linguaggio, definisce una rappresentazione per la sua grammatica ed un interprete per le frasi del linguaggio.
Iterator	Fornisce un modo di accesso agli elementi di un oggetto aggregato in modo sequenziale senza esporre la sua rappresentazione sottostante
Mediator	Definisce un oggetto che incapsula il modo in cui un insieme di oggetti interagisce in modo da permettere la loro indipendenza

Design Pattern – Comportamentali

Nome	Descrizione
Memento	Cattura e porta all'esterno lo stato interno di un oggetto senza violare l'incapsulazione in modo da ripristinare il suo stato più tardi
Observer	Definisce una dipendenza 1:N tra oggetti in modo che se uno cambia stato gli altri siano aggiornati automaticamente
State	Permette ad un oggetto di cambiare il proprio comportamento a seconda del suo stato interno, come se cambiasse classe di appartenenza
Strategy	Definisce una famiglia di algoritmi, li incapsula ognuno e li rende intercambiabili in modo da cambiare in modo indipendente dagli utilizzatori
Template method	Definisce lo scheletro di un algoritmo in un'operazione lasciando definire alcuni passi alle sottoclassi

Design Pattern – Comportamentali

Nome	Descrizione
Visitor	Rappresenta un'operazione da fare sugli elementi della struttura di un oggetto. Lascia definire nuove operazioni senza cambiare classe degli elementi

Consiglio: i DP riportati sopra sono tanti e complessi. Un professionista NON li impara a memoria ma ne capisce l'utilità e sa approcciarsi facilmente ai più comuni. Essere un professionista vuoldire anche riprendere un libro quando necessario per vedere se uno dei DP può fare al caso suo!

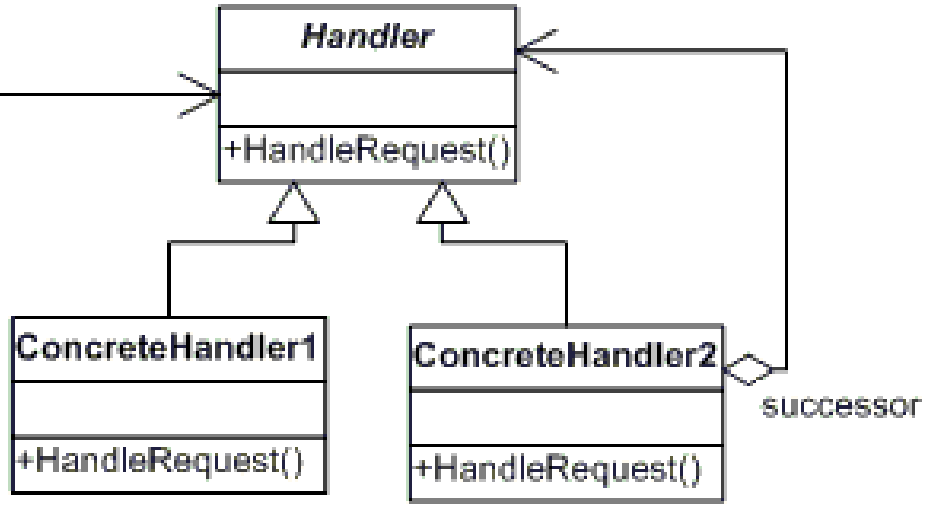
Design Pattern – Chain of Responsability

Esigenza: vogliamo simulare il processo di approvazione di un prestito dalla richiesta all'accettazione. Maggiore sarà la cifra, più in alto l'impiegato della banca dovrà scalare per l'apporvazione.

Caso d'uso:

- Il cliente fà la richiesta
- L'impiegato in banca l'approva se è inferiore a 10.000€ altrimenti la manda al suo superiore
- Il vice-direttore l'approva se la cifra è inferiore a 25.000€ altrimenti la manda al suo superiore
- Il direttore l'approva!

Design Pattern – Chain of Responsability



GRUPPO EURIS

Design Pattern – Chain of Responsability

```
public abstract class Approver
{
    protected Approver _boss;

    public void SetSuccesor(Approver boss)
    {
        _boss= boss;
    }

     public abstract void ProcessLoan(Loan loan);
}
```

Design Pattern – Chain of Responsability

Link al codice completo

```
public class Clark : Approver
{
    public override void ProcessLoan(Loan loan)
    {
        if(loan.Amount < 10000.0)
            Console.Writeline(‘’Approved!’’);
        else
            _boss.ProcessLoan(loan)
    }
}
```

Design Pattern – Esempi pratici

Mano al codice!!

Esercizo proposto: la nota pizzeria Sdomino vuole automatizzare il processo di ordinazione pizze e chiede a software che gestisca gli ordini in arrivo. In particolare:

- Le ordinazioni arrivano tramite file CSV, ogni riga contiene una pizza ordinata, nel seguente formato:

BasePizza;Impasto;Aggiunte

Esempio: «Margherita;Integrale;Prosciutto Cotto,Funghi»

Dove:

Base pizza può avere i seguenti valori: *Margherita (5€)*, *Pepperoni(7€)*, *Napoletana(3€)*

Impasto: normale (0€), integrale (+1€)

Le aggiunte possono essere più di una, separatio da «,» e sono: Prosciutto cotto (+1€), Funghi (+2€), Crudo (2€), Ananas (!!)

Lo scopo del software è:

-All'avvio leggere tutti gli ordini presenti e per ognuno di essi creare uno scontrino (ogni scontrino deve avere un identificativo progressivo) con il prezzo totale dell'ordine.
Attenzione: se una pizza contiene l'aggiunta Ananas allora la pizza è GRATIS!

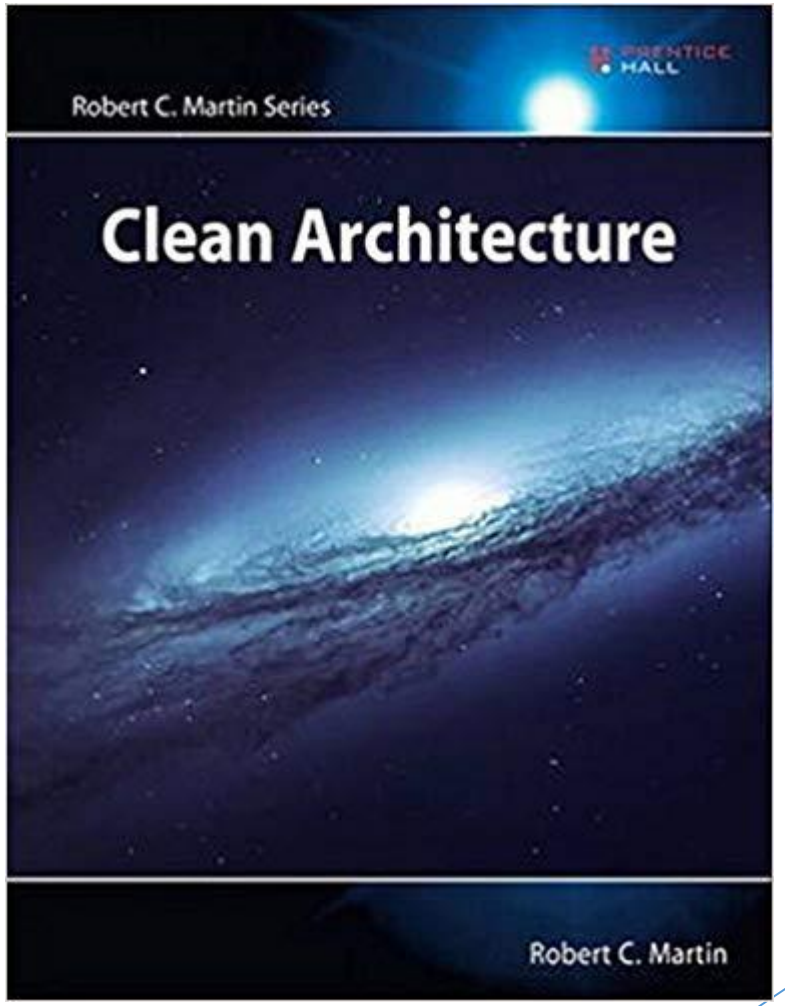
-Loggare lo scontrino su file e inserirlo a DB, in modo che gli ordini siano consultabili

Obbligo: lo scopo dell'esercizio è utilizzare al meglio i design pattern!!

S.O.L.I.D
Principle

S.O.L.I.D. - Indice

- Cosa significa?
- I cinque principi
- **S**RP
- **I**SP
- **D**IP
- Qualche esempio pratico



GRUPPO EURIS

S.O.L.I.D – Cosa significa?

- SOLID indica l'acrostico dei «cinque principi» della programmazione OOP, descritti da Robert C. Martin nei primi anni 2000.
- I principi SOLID (SOLID PRINCIPLES) sono intesi come linee guida per lo sviluppo di codice
  - **Leggibile**
  - **Estendibile**
  - **Manutenibile**
  - **Predisposto al Refactoring**

S.O.L.I.D. – I cinque principi

S ingle Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

O pen / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

L iskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

I nterface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

D ependency Inversion Principle
Program to an interface, not to an implementation.

S.O.L.I.D. – I cinque principi

Lettera	Nome	Acronimo	In pillole
S	Principio di singola responsabilità (*single responsability principle*)	SRP	Ogni classe deve avere una e una sola responsabilità, interamente incapsulata al suo interno
O	Principio aperto/chiuso (*open/closed principle*)	OCP	Un componente software deve essere aperta alle estensioni ma deve proteggersi da modifiche
L	Principio di sostituzione di Liskov (*Liskov substitution principle*)	LSP	Gli oggetti devono essere interscambiabili con dei sottotipi, senza alterare il comportamento del software.
I	Princio di segregazione delle interface (*Interface segregation principle*)	ISP	Sarebbero preferibili interfacce specifiche, che una singola generica.
D	Principio di inversione delle dipendeze (*Dependency inversion principle*)	DIP	Una classe dovrebbe dipendere dalle astrazioni e non da classi concrete.

S.O.L.I.D – Qualche esempio pratico

Mano al codice!!

- Esempi di SRP qui
- Esempi di ISP qui
- Esempi di DIP qui

- **Esercizo proposto:** riprendiamo il nostro **Main Project** e chiediamoci:
  - **Abbiamo applicato i Design Pattern? Se si quali?**
  - **Abbiamo usato i pincipi di SOLID?**

Thank you!