

# CORSO DESIGN PATTERN

Design Principles

**Dott. Romina Fiorenza**  
**[fiorenza.romina@gmail.com](mailto:fiorenza.romina@gmail.com)**

# E' una questione di principio!

- Nel mondo della progettazione e dello sviluppo si sono affermati sempre di più alcuni principi interessanti che sono un buon complemento ai design pattern
- La loro applicabilità non è totale e non seguono schemi precisi, sono essenzialmente **linee guida** che però possono fare la differenza!

# Design Pattern vs Design principles (1)

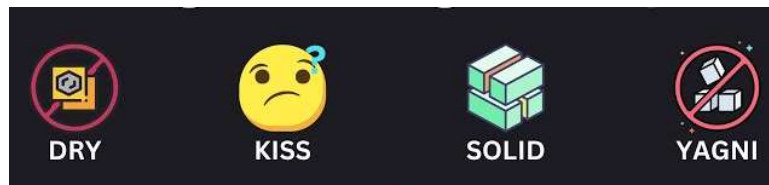
- I **Design Pattern** sono tecniche di progettazione
- Rappresentano soluzioni (**low level**) riutilizzabili relative a problemi di progettazione software ricorrenti.
- Caratteristiche:
  - ▣ Hanno un caso d'uso e uno scenario specifico
  - ▣ Spesso vengono forniti dettagli di implementazione.
  - ▣ Potrebbero dipendere dalla tecnologia o dal linguaggio
  - ▣ Rappresentano uno strumento preciso e ben collaudato che conferisce affidabilità all'architettura che lo utilizza
  - ▣ Bisognerebbe sempre giustificare quando li si utilizza

# Design Pattern vs Design principles (2)

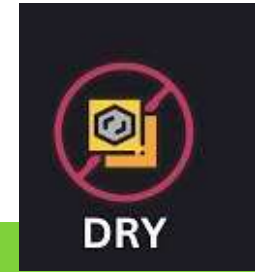
- I **Design principles**, a differenza dei design pattern, sono linee guida generali (*high level*) che aiutano a realizzare architetture software non solo funzionali, ma eleganti, affidabili e scalabili
  
- Caratteristiche:
  - ▣ Sono più astratti e meno prescrittivi dei design pattern
  - ▣ In genere non dipendono dalla tecnologia/linguaggio
  - ▣ E' consigliato applicarli per aumentare la qualità del codice
  - ▣ Bisognerebbe sempre giustificare quando NON li si utilizza

# Alcuni principles più famosi

1. DRY (Don't Repeat Yourself)
2. KISS (Keep it simple, stupid)
3. YAGNI (You ain't gonna need it)
4. SOLID (acronimo per 5 principles)

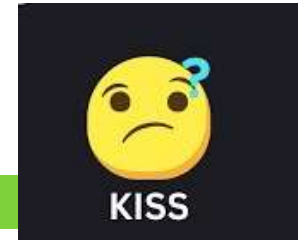


# DRY principle



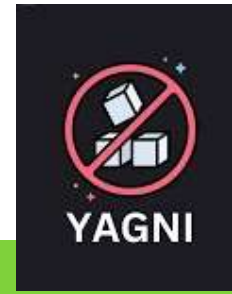
- **Acronimo di “Don't Repeat Yourself”** o anche conosciuto come **“Duplication is Evil” (DIE)**
- E' un principio base dello sviluppo software che promuove la riduzione delle ripetizioni di tutti i tipi di informazioni
- Afferma che *“In un sistema, ogni singolo pezzo di informazione deve avere un'unica, precisa e non ambigua rappresentazione”*
- E' molto usato nell'ambito delle architetture multitier, dove esiste una grossa mole di dati che transita da un tier all'altro.

# KISS principle



- Acronimo di “**Keep it simple stupid**” o anche e (forse più famoso) “**Keep it short and simple**”
- Il KISS principle afferma che la semplicità è l’obiettivo chiave del design e che tutte le complessità non necessarie devono essere evitate
- Una versione “ironica”, cambia leggermente il senso inserendo una virgola “**Keep it simple, stupid**” come fosse un imperativo scherzoso verso il progettista / programmatore
  - ▣ In forme diverse era stato formulato anche da Albert Einstein, Leonardo da Vinci e altri grandi della storia

# YAGNI principle



- Acronimo di "**You ain't gonna need it**" → non ne hai davvero bisogno!
- E' un principio di programmazione "estrema" secondo cui lo sviluppatore non dovrebbe aggiungere funzionalità, se non necessarie.
- Possibili conseguenze negative dovute allo sviluppo di codice non necessario sono:
  - maggior tempo per progettare/sviluppare/testare  
*oppure al contrario*
  - siccome non serve, la funzionalità non viene progettata bene, non viene documentata e magari non viene testata → pessima qualità del codice



# SOLID



- SOLID è l'acronimo di 5 principi:



# Single Responsibility Principle

- SRP afferma che un modulo o una classe dovrebbe avere una **singola responsabilità**
  - ▣ Cioè tutti i servizi offerti dall'elemento dovrebbero essere strettamente allineati a tale responsabilità.
- Una responsabilità è un **motivo per cambiare**.
- Corollario di SRP è un elemento dovrebbe avere uno e un solo motivo per cambiare.
- Questo principio favorisce una manutenzione più semplice e minor impatto durante le modifiche

# Esempio SRP

- Consideriamo questa classe che compila e stampa un report

```
public class GestoreReport {  
  
    private String report;  
  
    public void generaReport() {  
        report = "questo è il testo del report";  
    }  
  
    public void stampaReport() {  
        System.out.println(report);  
    }  
  
}
```

Il contenuto del report  
potrebbe cambiare

La visualizzazione del report  
potrebbe cambiare

- Queste due cose cambiano per ragioni molto diverse: SRP le considera 2 responsabilità distinte che andrebbero gestite in 2 classi differenti

# Soluzione SRP

- Ogni classe ha UNA responsabilità e un SOLO motivo per cambiare

```
public class CreatorReport {  
  
    public String generaReport() {  
        return "questo è il testo del report";  
        // se cambia il contenuto del report, le modifiche ricadono solo su questa classe  
    }  
  
}
```

```
public class PrinterReport {  
  
    public void stampaReport(String report) {  
        System.out.println(report);  
        // se cambia modalità di stampa, le modifiche ricadono solo su questa classe  
    }  
  
}
```

# Open-Closed Principle

- Open-Closed Principle afferma che una classe dovrebbe essere **Open** for extension e **Closed** to modification.
- Quindi l'aggiunta o modifica dei comportamenti dovrebbe passare sempre attraverso le estensioni, senza apportare modifiche al codice preesistente.
- Per realizzare questo è necessario
  - ▣ separare correttamente le responsabilità (SRP è prerequisito per l'Open-Closed Principle)
  - ▣ definire un'interfaccia per i comportamenti sui quali ci si vuole predisporre alle modifiche

# Esempio Open-Closed

- Supponiamo di avere una classe Account che gestisce i dati del conto e calcola gli interessi in base alla tipologia di contratto (tipo di conto)

```
public class Account {  
  
    private String titolare;  
    private double saldo;  
  
    // getters & setters  
  
    public double calcolaInteressi(String tipoAccount) {  
        if(tipoAccount.equals("young"))  
            return saldo * 0.5;  
        else // standard account  
            return saldo * 0.7;  
    }  
}
```

Se si vogliono prevedere nuovi tipi di conto e nuovi calcoli dell'interesse, bisognerebbe agire sull'if → NON è corretto per l'Open-Closed

- Innanzitutto questa soluzione non applica l'SRP e quindi bisogna separare la responsabilità del calcolo degli interessi dalla classe Account

# Soluzione Open-Closed

(1 / 2)

- Si crea una interfaccia che astrae il calcolo degli interessi e si aggiungono le classi concrete in funzione dei vari tipi di conto (low coupling)

```
public class Account {  
  
    private String titolare;  
    private double saldo;  
    public String getTitolare() {  
        return titolare;  
    }  
    public double getSaldo() {  
        return saldo;  
    }  
}
```



```
public interface ICalculatorInterest {  
  
    public double calcolaInteresse(Account account);  
}
```



# Soluzione Open-Closed

(2/2)

- Le varie classi, tutte indipendenti, che eseguono l'overriding del metodo dell'interfaccia

```
public class CalculatorInterestYoungAcc implements ICalculatorInterest {  
    @Override  
    public double calcolaInteresse(Account account) {  
        return account.getSaldo() * 0.5;  
    }  
}
```

```
public class CalculatorInterestStandardAcc implements ICalculatorInterest {  
    @Override  
    public double calcolaInteresse(Account account) {  
        return account.getSaldo() * 0.7;  
    }  
}
```

```
public class CalculatorInterestOver60Acc implements ICalculatorInterest {  
    @Override  
    public double calcolaInteresse(Account account) {  
        return account.getSaldo() * 0.1;  
    }  
}
```



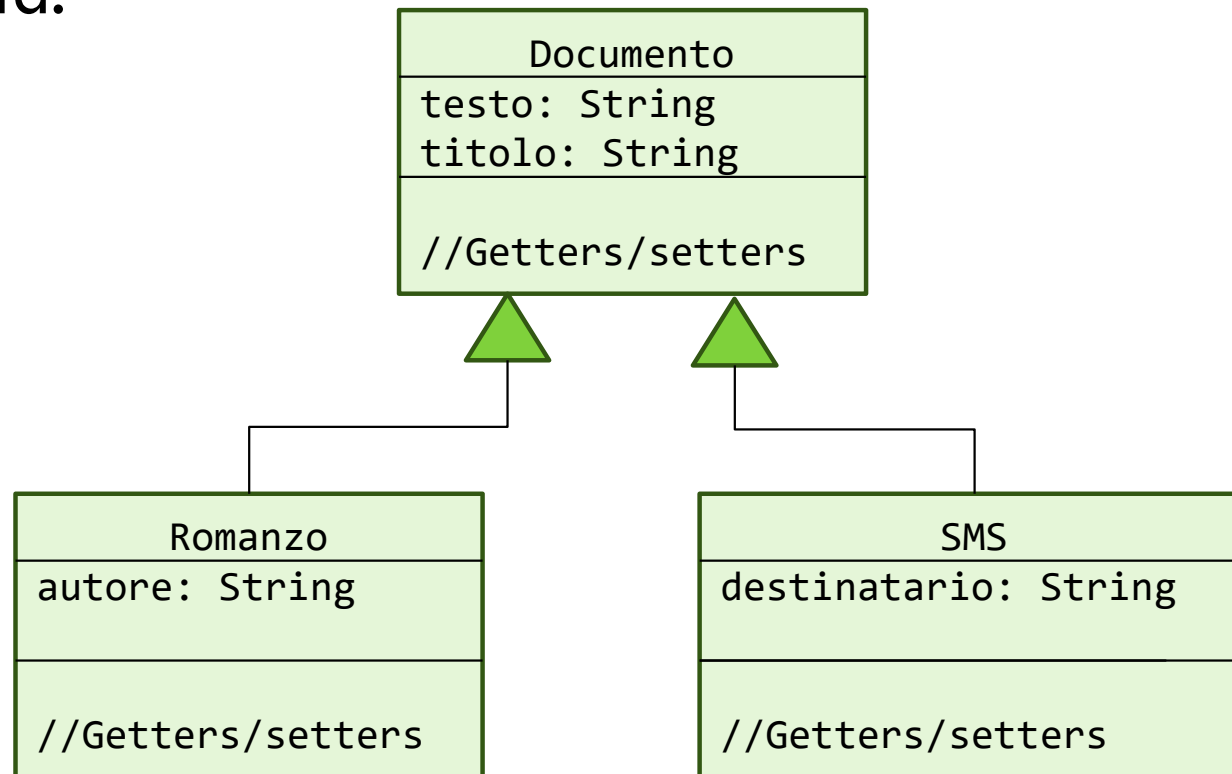
# Liskov Principle

- Deve il nome a Barbara Liskov, informatica statunitense e costituisce una particolare definizione di sottotipo.
- Afferma precisamente che se Sub è sotto classe di Sup, allora in qualunque parte del codice dove è possibile usare Sup, deve essere possibile utilizzare Sub, senza alterare il funzionamento del programma.
- Piu sinteticamente: Utilizzare una classe figlia al posto di una classe padre non cambia il comportamento del programma
- Questo porta inevitabilmente a scoraggiare la tecnica del subclassing classico a favore di altre tecniche tra cui quella di composizione

# Esempio Liskov

(1 / 3)

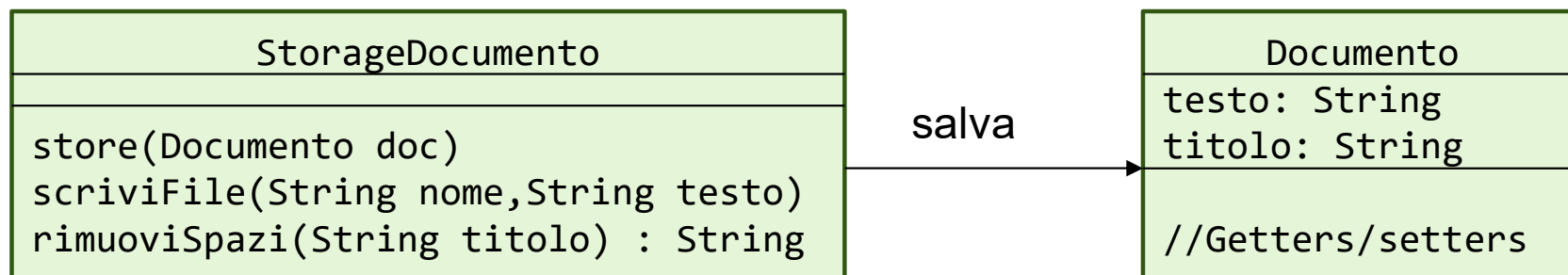
- Supponiamo di avere una gerarchia di oggetti Documento così descritta:



# Esempio Liskov

(2/3)

- Supponiamo inoltre di avere una classe `StorageDocumento` che prevede un metodo `store` per memorizzare documenti, che utilizza i metodi `scriviFile` e `rimuoviSpazi` che si avvalgono del `testo` e del `titolo` del documento.



# Esempio Liskov

(3/3)

- Nel main creo un Romanzo ed un SMS e provo a salvarli con la classe StorageDocumento.

```
public class Main {  
    public static void main(String[] args) {  
        Romanzo rom = new Romanzo();  
        rom.setTitolo("Divina Commedia");  
        rom.setAutore("Dante Alighieri");  
        rom.setTesto("Nel mezzo del cammin di nostra vita...");  
  
        SMS sms = new SMS();  
        sms.setDestinatario("346331122");  
        sms.setTesto("ciao ciao");  
  
        StorageDocumento storage = new StorageDocumento();  
        storage.store(rom);  
        storage.store(sms);  
    }  
}
```

SMS non imposta il titolo, sebbene lo eredita da Documento.

Il metodo store va in eccezione

```
scrittura nel file: DivinaCommedia.txt  
Divina Commedia  
Nel mezzo del cammin di nostra vita...  
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.replaceAll(String, String)" because "titolo" is null  
    at liskov.StorageDocumento.rimuoviSpazi(StorageDocumento.java:19)  
    at liskov.StorageDocumento.store(StorageDocumento.java:7)  
    at liskov.Main.main(Main.java:17)
```

# Soluzione errata

- La classica soluzione di aggiungere un if per gestire i sottotipi di Documento (dove serve) è sbagliata perché viola l'Open-Closed principle

```
public class StorageDocumento {  
  
    public void store(Documento doc) {  
        String testo = doc.getTitolo() + "\n" + doc.getTesto();  
        if(doc instanceof SMS)  
            scriviFile(rimuoviSpazi(((SMS)doc).getDestinatario()) + ".txt", testo);  
        else  
            scriviFile(rimuoviSpazi(doc.getTitolo()) + ".txt", testo);  
    }  
  
    private void scriviFile(String nomeFile, String testo) {  
        // codice che scrive sul file il testo specificato  
        System.out.println("scrittura nel file: " + nomeFile);  
        System.out.println(testo);  
    }  
  
    private String rimuoviSpazi(String titolo) {  
        // supponiamo di codice che rimuove gli spazi dal titolo  
        String newtitolo = titolo.replaceAll(" ", "");  
        return newtitolo;  
    }  
}
```

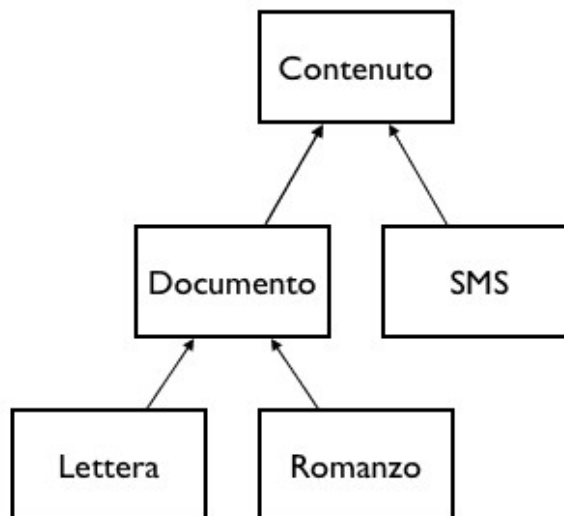
errato

# Soluzione Liskov n.1

- **Usare l'Astrazione.**
- Eliminare gli aspetti che non sono realmente comuni a tutte le sottoclassi
- Questa soluzione comporta però la necessità di reimplementare l'aspetto rimosso all'interno delle classi figlie che ne beneficiavano realmente, rischiando di incorrere in pericolose duplicazioni di codice.

# Soluzione Liskov n.2

- ❑ **Creare un nuovo livello.**
- ❑ Prevedere una classe Contenuto, ancora più generica, che verrà poi estesa sia da Documento che da SMS
- ❑ Impostare il comportamento di Contenuto in base a quello di SMS e poi prevedere l'overriding nella classe Documento



# Soluzione Liskov n.2

- Nell'esempio Documento dovrebbe sovrascrivere il metodo getDestinatario restituendo il titolo del documento

```
public class Contenuto {  
  
    private String testo;  
    private String destinatario;  
    public String getTesto() {  
        return testo;  
    }  
    public void setTesto(String testo) {  
        this.testo = testo;  
    }  
    public String getDestinatario() {  
        return destinatario;  
    }  
    public void setDestinatario(String destinatario) {  
        this.destinatario = destinatario;  
    }  
}
```

```
public class Documento extends Contenuto{  
  
    private String titolo;  
  
    public String getTitolo() {  
        return titolo;  
    }  
  
    public void setTitolo(String titolo) {  
        this.titolo = titolo;  
    }  
  
    @Override  
    public String getDestinatario() {  
        return titolo;  
    }  
}
```

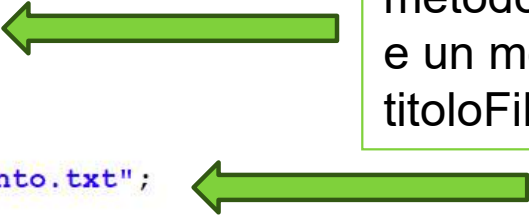


# Soluzione Liskov n.3

- Implementare il comportamento di base nella classe padre e poi fare overriding nelle derivate

```
public class Documento {  
  
    private String titolo;  
    private String testo;  
  
    public String stampa() {  
        return titolo + " " + testo;  
    }  
  
    public String titoloFile() {  
        return rimuoviSpazi(titolo) + ".documento.txt";  
    }  
  
    public String rimuoviSpazi(String titolo) {  
        // supponiamo di codice che rimuove gli spazi dal titolo  
        String newtitolo = titolo.replaceAll(" ", "");  
        return newtitolo;  
    }  
  
    // getters/setters
```

Documento definisce un metodo che imposta il testo e un metodo che imposta il titoloFile (senza spazi)



# Soluzione Liskov n.3

- Quindi SMS opera con l'overriding e StorageDocumento avrà una versione semplificata del metodo store:

```
public class SMS extends Documento {  
    private String destinatario;  
  
    public String stampa() {  
        return destinatario + " " + getTesto();  
    }  
  
    public String titoloFile() {  
        return rimuoviSpazi(destinatario) + ".sms.txt";  
    }  
  
    public String getDestinatario() {  
        return destinatario;  
    }  
  
    public void setDestinatario(String destinatario) {  
        this.destinatario = destinatario;  
    }  
}
```

```
public class StorageDocumento {  
    public void store(Documento doc) {  
        // la chiamata usa titoloFile di Documento, che avrà overriding in SMS  
        scriviFile(doc.titoloFile(), doc.getTesto());  
    }  
  
    private void scriviFile(String nomeFile, String testo) {  
        // codice che scrive sul file il testo specificato  
        System.out.println("scrittura nel file: " + nomeFile);  
        System.out.println(testo);  
    }  
}
```

# Interface Segregation Principle

- Questo principio afferma che avere molte interfacce specifiche è preferibile ad averne poche e generiche.
- In altre parole, le interfacce che prevedono molti comportamenti sono quindi da evitare perchè scomode da mantenere ed evolvere.
- Le interfacce (dove possibile) dovrebbero essere semplici e specifiche rispetto alle loro finalità.
  - ▣ Il principio dovrebbe essere che ogni interfaccia è pensata per uno specifico Client
- Le conseguenze immediate sono che le classi potranno implementare anche molte interfacce ma NON saranno costrette ad implementare metodi che non usano.

# Esempio ISP

## □ Contesto iniziale non ottimizzato con I

```
public interface ILaundry {  
  
    public void washClothes();  
    public void dryClothes();  
}
```

```
public class LavaAsciuga implements ILaundry {  
  
    @Override  
    public void washClothes() {  
  
        System.out.println("fase 1: lavaggio");  
    }  
  
    @Override  
    public void dryClothes() {  
        System.out.println("fase 2: asciugatura");  
    }  
}
```

Vorremo evitare di sollevare eccezione  
quando il metodo che bisogna  
sovrascrivere non è pertinente con la classe

```
public class Lavatrice implements ILaundry{  
  
    @Override  
    public void washClothes() {  
        System.out.println("eseguo lavaggio...");  
    }  
  
    @Override  
    public void dryClothes() {  
        throw new UnsupportedOperationException("questo modello di lavatrice non asciuga");  
    }  
}
```

# Soluzione ISP

- Invece di avere una sola interfaccia, ne prevedo 2

```
public interface IWashMachine{  
    public void washClothes();  
}
```

```
public interface IDryMachine {  
    public void dryClothes();  
}
```

```
public class LavaAsciuga implements IWashMachine, IDryMachine {  
    @Override  
    public void washClothes() {  
        System.out.println("fase 1: lavaggio");  
    }  
    @Override  
    public void dryClothes() {  
        System.out.println("fase 2: asciugatura");  
    }  
}
```

- LavaAsciuga implementa 2 interfacce,  
mentre Lavatrice sola una!

```
public class Lavatrice implements IWashMachine{  
    @Override  
    public void washClothes() {  
        System.out.println("eseguo lavaggio...");  
    }  
}
```

# Dependency Inversion principle

- Questo principio afferma che i moduli high level NON devono dipendere dai moduli low level, entrambi dovrebbero dipendere dalle astrazioni.
- Le astrazioni NON dovrebbero dipendere dalle implementazioni
- Invece le implementazioni dovrebbero dipendere dalle astrazioni
  
- In altre parole, tutti i moduli (high o low) del programma non devono dipendere da oggetti specifici ma sempre dalle astrazioni di essi (dalle interfacce che essi implementano).
  
- In questo modo si realizzano architetture più flessibili e scalabili

# Esempio DIP

- Supponiamo di avere la classica stratificazione



```
public class DataPersistenceClass {  
    public void lowLevelFunction() {  
        System.out.println(">> store data");  
    }  
    public void otherLowLevelFunction() {  
        System.out.println(">> load data");  
    }  
}  
  
public class BusinessClass {  
    private DataPersistenceClass persistence;  
    public BusinessClass() {  
        this.persistence = new DataPersistenceClass();  
    }  
    public void service() {  
        System.out.println("starting business service...");  
        this.persistence.lowLevelFunction();  
        this.persistence.otherLowLevelFunction();  
        System.out.println("end service");  
    }  
}
```

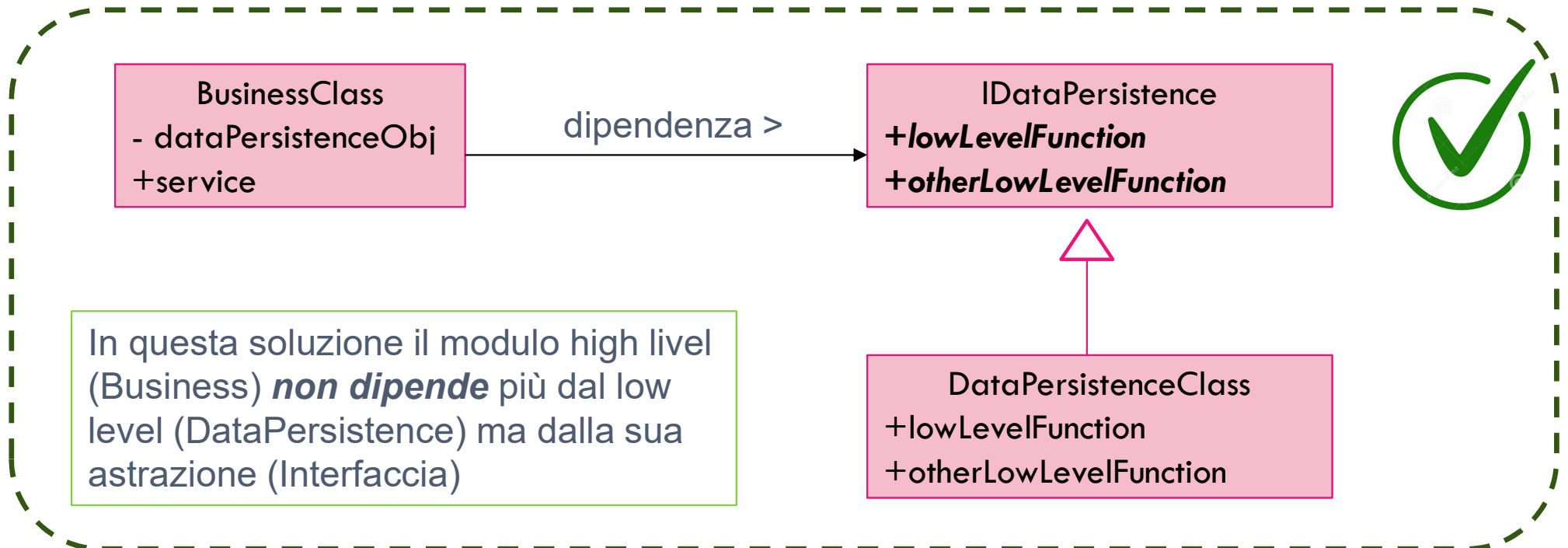
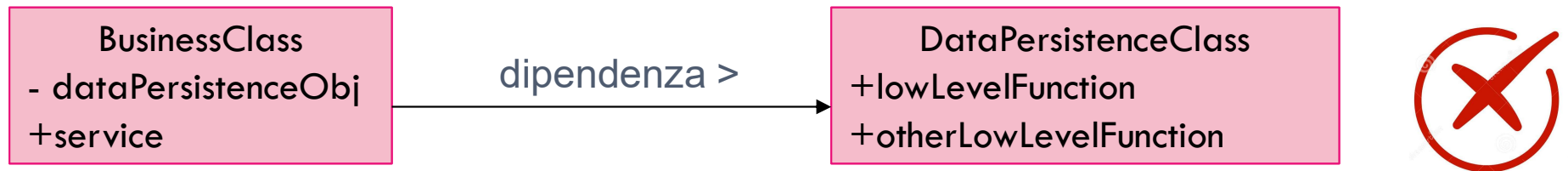
La classe di business dipende dalla classe del Persistence layer  
Similmente il presentation layer dipenderà dalla classe di business



Un cambiamento alla classe di persistenza, porterebbe modifiche alla classe di business perché il coupling è molto alto

# Soluzione DIP

- La soluzione DIP consiste nel creare interfacce per ogni coppia di elementi da disaccoppiare e invertire così le dipendenze





# Soluzione DPI (2)

- Crea l'interfaccia dell'oggetto di persistenza e la classe concreta, le modifiche ricadono sul costruttore della classe Business

```
public class BusinessClass {  
  
    private DataPersistenceClass persistence;  
  
    public BusinessClass() {  
        this.persistence = new DataPersistenceClass();  
    }  
}
```

NO DIP

```
public class BusinessClass {  
  
    private IDataPersistence persistence;  
  
    public BusinessClass(IDataPersistence persistence) {  
        this.persistence = persistence;  
    }  
}
```

soluzione  
DIP

- Avendo a disposizione un framework che risolve le dipendenze, si potrebbe configurare quale specifico DataPersistence vogliamo abbinare alla classe Business e il framework realizza la **Dependency Injection**