SOLID Principle

# SOLID Full Form

* **S**stands for **Single Responsibility Principle**
* **O**stands for **Open Closed Principle**
* **L**stands for **Liskov Subsitution Princple**
* **I** stands for **Interface Segregation Principle**
* **D** stands for **Dependency Inversion Principle**

# Single Responsibility Principle

* 
* <https://paigeshin1991.medium.com/single-responsibility-principle-what-working-as-a-software-engineer-you-dont-know-it-e74883f92944>
* **Every Software Component should have only one responsibility.**
  + “Software component” could refer to a **class** or a **function** or a **method** or **even a module.**
  + “a class should have one responsibility.”.
  + “a function should tackle only one task” .
* **Always Aim for High Cohesion**
  + Cohesion is the degree to which the various parts of a software component are related
* **Avoid Tight Coupling**
  + Coupling is defined as the level of inter dependency between various software components
* **Software component needs only one reason to change**

# Open Closed Principle: Make your code cost-free and flexible



* [**https://paigeshin1991.medium.com/open-closed-principle-make-your-code-cost-free-and-flexible-49547c9a801a**](https://paigeshin1991.medium.com/open-closed-principle-make-your-code-cost-free-and-flexible-49547c9a801a)
* **Software components should be closed for modification, but open for extension**
* **Introduce interface**. And make other ‘additional classes’ conform to that interface. discountRate () method takes now ‘interface’ instead of a specific class. This is called ‘**abstraction’**. By doing this, not only can you add other types of customers **(open for extension**), you are not forced to modify directly your defined code (**closed for modification**).

## Benefits of this approach

* Ease of adding new features.
* Leads to minimal cost of developing and testing software.
* Open Closed Principle often requires decoupling, which, in turn, automatically follows the Single Responsibility Principle.

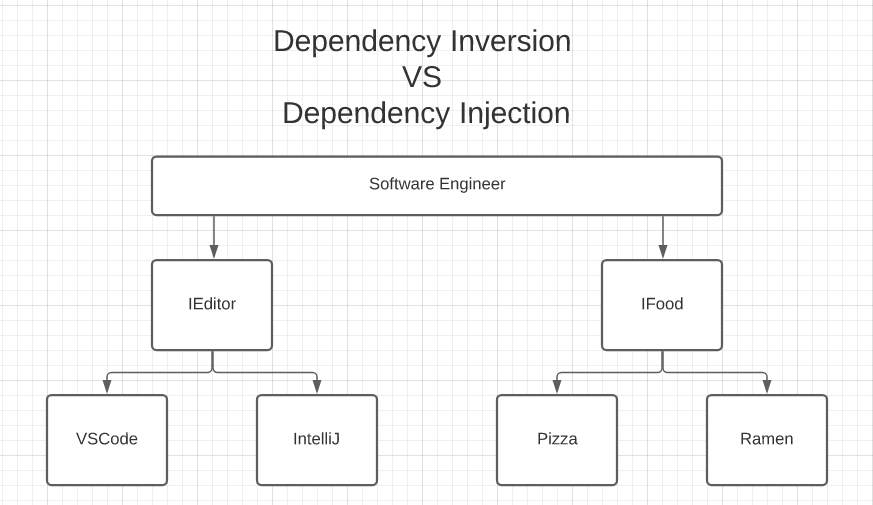
# Liskov Substitution Principle

* s
* <https://paigeshin1991.medium.com/liskov-substitution-principle-top-developers-technique-which-improves-2-5x-d8af89e07afb>
* Top Developer’s technique which improves 2.5x the quality of your code
* Objects should be replaceable with their subtypes without affecting the correctness of the program

# Interface Segregation Principle

* 
* <https://paigeshin1991.medium.com/interface-segregation-principle-your-only-way-to-be-a-super-competent-developer-698a7976e7f2>
* Your only way to be a super competent developer
* All enterprise level developers actively use interfaces
* No client should be forced to depend on methods it does not use
* The point is **“YOU SHOULD ONLY DEFINE METHODS THAT ARE GOIND TO BE USED”.**
* **ISP starts from splitting big FAT interfaces into smaller interfaces.**
* **Techniques to identify violations**
  + Find if your Interfaces have methods with low cohesion
  + Check Empty Method Implementations

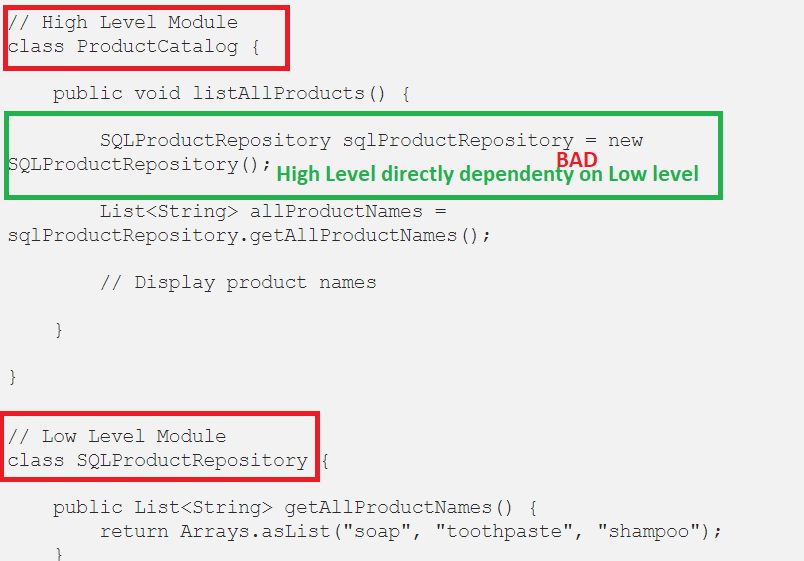
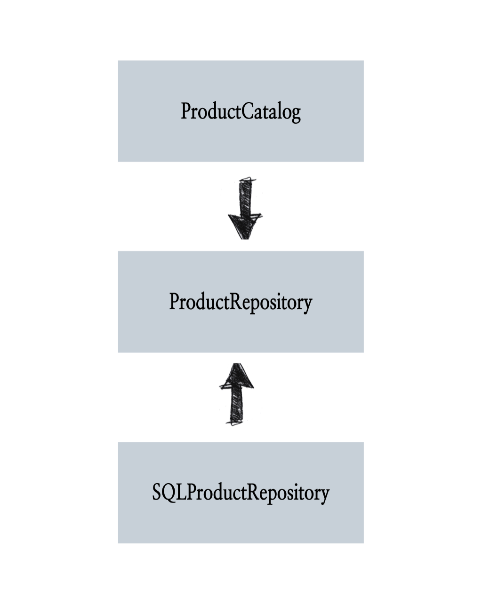
# Dependency Inversion vs. Dependency Injection

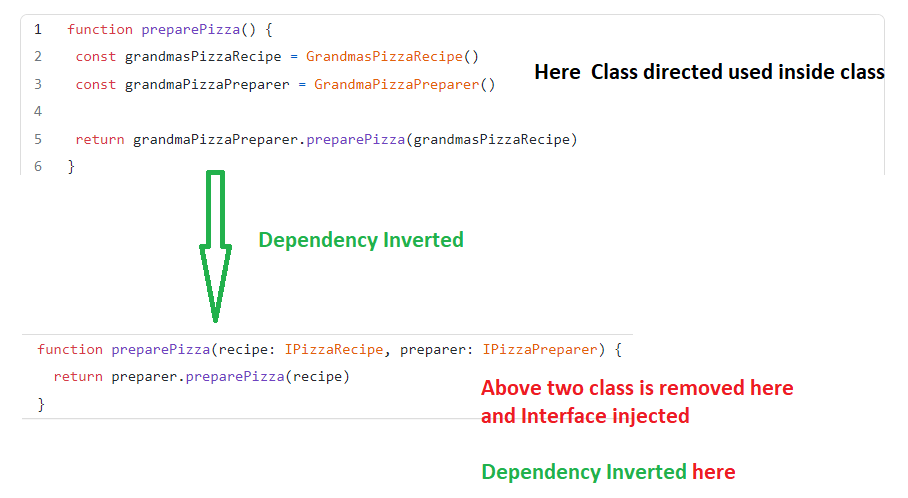




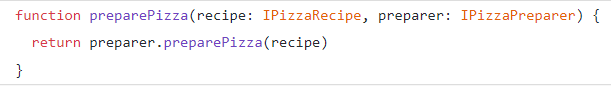
## **Dependency inversion**

* + It is a **design principle**
  + In simple terms, we should **not depend on low-level implementations**, but rather **rely on high-level abstractions**
  + Allows us to be agnostic towards the implementation details
  + Help to decouple from the low-level implementation details
  + High-level modules (Class want to use dependency) should not depend on low-level modules (Class which is used as dependency). **Both should depend on abstractions (Interface)**
  + Abstractions should not depend on details. Details should depend on abstractions



## **Dependency Injection**

* + It is a **design pattern**
  + allows us to **separate creation from use**
  + It allows us to “inject” the required objects at run-time, without worrying about constructing them ourselves
  + work hand in hand with the dependency inversion principle.
  + allows us to write more configurable and dynamic code
  + **Improves our ability to test our code**, since it allows us to easily mock injected classes, so we can focus on testing our core logic.
  + dependency injection enables us to use dependency inversion and rely on high-level abstractions (Interface)
  + 

## Interface -C#

* Help to implement dependency Injection
  + Help to write loosely coupled code
* Help to write testable code as it helps to Mock it
* A contract to a class which implement this interface – all method must be implemented
* Help to achieve multiple inheritance