

# SOLID Principle With C#

## اصول سالیڈ با سی شارپ

🧠 In the era of scalable software systems, project success hinges not only on development velocity but also on the quality and forward-looking nature of its design. SOLID principles serve as a scientific roadmap, empowering developers to clearly separate concerns and minimize coupling. Whether you're a beginner or an experienced engineer, practical mastery of these principles especially through step by step C# examples propels your professional growth and overall project reliability when facing the challenges of complex systems. Through bilingual (Persian + English) content and a real-world Order Management System project encompassing all five SOLID principles, this article offers a reliable, in-depth resource for mastering clean and flexible software architecture.

🧠 در عصر توسعه نرم افزارهای مقیاس پذیر، آنچه موفقیت پروژه ها را تضمین می کند نه فقط سرعت تولید بلکه کیفیت و آینده نگرانی طراحی آن است. اصول SOLID، همچون نقشه راه توسعه نرم افزار، چارچوبی علمی برای جداسازی مسئولیت ها و کاهش وابستگی ها ارائه می دهد. چه توسعه دهنده مبتدی باشید و چه حرفه ای، فراگیری عملی این اصول به ویژه با مثال های گام به گام زبان C# شما را در مسیر رشد حرفه ای و مصمم تر شدن در مواجهه با مشکلات سیستم های بزرگ یاری خواهند داد. این مقاله با ترکیب محتوای آموزشی فارسی و انگلیسی و ارائه یک پروژه واقعی مدیریت سفارشات که هر پنج اصل SOLID را پوشش می دهد، مرجعی مطمئن برای یادگیری عمیق و عملی معماری نرم افزار پاک و منعطف خواهد بود.

## فهرست

.....	Single Responsibility Principle
.....	Open-Closed Principle
.....	Liskov Substitution Principle
.....	Interface Segregation Principle
.....	Dependency Inversion Principle
.....	Final Project Order Manage System (OMS)

تاریخ انتشار : 1404/07/11

نویسنده : رضا فردوس آراء

وبسایت : [rezaferdosara.ir](http://rezaferdosara.ir)

دسته بندی: طراحی نرم افزار، اصول SOLID، معماری شی گرا، آموزش C#

سطح : مبتدی تا متوسط

Published on: 11/07/1404

Author: Reza FerdosAra

Website: [rezaferdosara.ir](http://rezaferdosara.ir)

Category: Software Design, SOLID Principles, Object-Oriented Architecture, C# Education

Level: Beginner to Intermediate

## ◆ Principle: SRP – Single Responsibility Principle (SOLID)

◆ اصل: SRP – اصل مسئولیت واحد

### What is SRP?

This principle states that each class should have only one responsibility, which naturally means there should be only one reason to change the class. When this principle is not followed, a class will contain a large volume of code, and if any changes are needed in the system, that class will require modification. Applying changes to this class will result in rerunning the tests. On the other hand, by adhering to this principle, a large problem is effectively divided into multiple smaller problems, each implemented within its own class. Therefore, changes in the system will affect only one of these smaller classes, and only the tests related to that small class need to be rerun. The Single Responsibility Principle (SRP) is very similar to another object-oriented principle called Separation of Concerns (SoC).

### اصل SRP چیست؟

این اصل بیان دارد که هر کلاسی فقط و فقط بایستی یک وظیفه داشته باشد که به طبع آن صرفاً یک دلیل برای تغییر کلاس وجود خواهد داشت. زمانیکه این اصل رعایت نشود، یک کلاس حاوی حجم زیادی کد خواهد بود که در صورت نیاز به اعمال تغییر در سامانه، این کلاس نیاز به تغییر خواهد داشت. اعمال تغییر در این کلاس، منجر به اجرای مجدد تست ها خواهد شد. در سوی مقابل با رعایت این اصل، در واقع یک مشکل بزرگ به چندین مشکل ریزتر تقسیم می شود که هر مشکل در قالب یک کلاس پیاده می شود. فاذا اعمال تغییر در سیستم منجر به تغییر در یکی از این کلاس های کوچک خواهد شد و صرفاً نیاز خواهد بود تست های مرتبط با این کلاس کوچک مجدداً اجرا شوند. اصل SRP بسیار شبیه اصل دیگری در شی گرای به نام SoC (Separation Of Concern) نیز می باشد.

## Scenario Overview

We are building a simple customer registration system. When a new customer is added, we want to log the operation. The mistake happens when logging logic is placed directly inside the business class.

 توضیح سناریو

در حال ساخت یک سیستم ساده ثبت مشتری هستیم. وقتی مشتری جدیدی اضافه می‌شود، می‌خواهیم عملیات را لاگ کنیم. اشتباه زمانی رخ می‌دهد که منطق لاگ‌نویسی مستقیماً داخل کلاس تجاری قرار می‌گیرد.

### ✗ Before Applying SRP (Wrong Design)

```
public class WrongSRP
{
    public void AddCustomer()
    {
        // Add a New Customer Into Database
        System.IO.File.WriteAllText("C://log.txt", "Customer Added Successful");
    }
}
```

### Problems:

- The class WrongSRP is responsible for both customer registration and logging.
- If logging changes (e.g., switch to database or cloud), this class must change.
- Violates SRP by mixing unrelated responsibilities.

- کلاس WrongSRP هم مسئول ثبت مشتری است و هم لاگ نویسی.
- اگر نحوه لاگ نویسی تغییر کند (مثلاً ذخیره در دیتابیس یا کلود)، این کلاس باید تغییر کند.
- اصل SRP را با ترکیب مسئولیت‌های نامرتبط نقض می‌کند.

### ✅ After Applying SRP (Correct Design)

```
public class CorrectSRP
{
    public void AddCustomer()
    {
        // Add a New Customer Into Database

        // ☒ Delegate logging to a separate class
        Log.Write("Customer Added Successful");
    }
}

public static class Log
{
    public static void Write(string msg)
    {
        // Write Log Into Database
    }
}
```

### 🔍 Improvements:

- CorrectSRP only handles customer registration.
- Logging is delegated to the Log class.
- Each class has a single, focused responsibility.

- کلاس CorrectSRP فقط مسئول ثبت مشتری است.
- لاگ نویسی به کلاس جداگانه‌ای به نام Log واگذار شده.
- هر کلاس فقط یک مسئولیت مشخص و متمرکز دارد.

## 📊 Comparison Table

Aspect	❌ WrongSRP	✅ CorrectSRP
Responsibility	Registration + Logging	Only Registration
Maintainability	Low – tightly coupled logic	High – loosely coupled
Testability	Hard to isolate	Easy to test independently
Extensibility	Changes affect multiple areas	Changes are localized

## 📌 Key Takeaways

- ✅ Keep each class focused on a single task.
- ✅ Delegate secondary responsibilities (like logging, emailing, etc.) to separate classes.
- ✅ SRP leads to cleaner, more maintainable, and testable code.

## 📌 نکات کلیدی

- ✅ هر کلاس را فقط برای یک وظیفه خاص طراحی کنید.
- ✅ مسئولیت‌های جانبی مثل لاگ نویسی را به کلاس‌های جداگانه واگذار کنید.
- ✅ رعایت SRP باعث کدی تمیزتر، قابل نگهداری‌تر و تست‌پذیرتر می‌شود.

## ◆ Principle: OCP – Open/Closed Principle (SOLID)

◆ اصل – OCP: اصل باز/بسته بودن

### What is OCP?

This principle states that a class should be open for extension but closed for modification. In other words, once a class has been implemented and other parts of the system start using it, that class should no longer be changed. It is clear that changing this class could cause problems for those parts of the system that depend on it. If new features need to be added to the class, they should be added by extending the class rather than modifying it.

### اصل OCP چیست؟

این اصل بیان می دارد که یک کلاس بایستی برای گسترش دادن باز و برای اعمال تغییر بسته باشد. به عبارت دیگر زمانیکه یک کلاس پیاده سازی می شود و بخش های دیگر سامانه از این کلاس شروع به استفاده می کنند ، دیگر نباید این کلاس تغییر کند. واضح است که اعمال تغییر در این کلاس می تواند بخش هایی از سامانه را که از آن کلاس می کنند را با مشکل روبرو سازد. چنانچه نیاز باشد تا قابلیت های جدیدی به کلاس افزوده شوند. این قابلیت ها از طریق بسط و گسترش دادن کلاس بایستی به آن افزوده شود.

### Scenario Overview

We are building a discount system for different product types. Each discount type (Fall, Seasonal, Event, etc.) has its own percentage. The wrong approach uses conditional logic inside a single class, making it hard to extend and maintain.

در حال ساخت یک سیستم تخفیف برای محصولات مختلف هستیم. هر نوع تخفیف (پاییزه، فصلی، مناسبتی و ...) درصد خاص خود را دارد. روش اشتباه از شرطهای متعدد در یک کلاس استفاده می‌کند که توسعه و نگهداری را سخت می‌کند.

## ✗ Before Applying OCP (Wrong Design)

```
public class WrongOCP
{
    public string ProductName { get; set; }
    public decimal ProductPrice { get; set; }
    public string DiscountType { get; set; }

    public decimal Discount()
    {
        if (DiscountType == "Fall")
            return (ProductPrice / 100) * 10;
        if (DiscountType == "Seasonal")
            return (ProductPrice / 100) * 25;
        if (DiscountType == "Event")
            return (ProductPrice / 100) * 30;
        // And more...

        return 0;
    }
}
```

## Problems:

- Every time a new discount type is added, this class must be modified.
- Violates OCP by mixing logic and data.
- Hard to test and maintain.



- هر بار که نوع تخفیف جدیدی اضافه شود، باید این کلاس را تغییر دهیم.
- اصل OCP را با ترکیب منطق و داده نقض می‌کند.
- تست و نگهداری آن دشوار است.

### ✅ After Applying OCP (Correct Design)

```
public class DiscountBaseClass
{
    public string ProductName { get; set; }
    public decimal ProductPrice { get; set; }

    public virtual decimal Discount()
    {
        return 0;
    }
}

public class FallDiscount : DiscountBaseClass
{
    public override decimal Discount()
    {
        return (ProductPrice / 100) * 10;
    }
}

public class SeasonalDiscount : DiscountBaseClass
{
    public override decimal Discount()
    {
        return (ProductPrice / 100) * 25;
    }
}

public class EventDiscount : DiscountBaseClass
{
    public override decimal Discount()
    {
        return (ProductPrice / 100) * 30;
    }
}
```





## Improvements:

- Each discount type is a separate class.
- No need to modify existing code when adding new types.
- Follows OCP by allowing extension via inheritance.




 بهبودها:

- هر نوع تخفیف در کلاس جداگانه‌ای پیاده‌سازی شده است.
- برای اضافه کردن نوع جدید نیازی به تغییر کد قبلی نیست.
- اصل OCP با قابلیت توسعه از طریق ارث‌بری رعایت شده است.

## Comparison Table

Aspect	 WrongOCP	 CorrectOCP
Extensibility	Low – requires modification	High – add new classes only
Maintainability	Hard – logic mixed	Easy – logic separated
Testability	Complex condition branches	Simple, isolated classes
OCP Compliance	 Violated	 Fully respected

## Key Takeaways

-  Avoid hard-coded conditionals for behavior changes.
-  Use inheritance or polymorphism to extend functionality.
-  OCP leads to scalable, maintainable, and flexible code.

## 🔴 نکات کلیدی

- ✓ از شرطهای سخت کد شده برای تغییر رفتار اجتناب کنید.
- ✓ از ارث‌بری یا پلی‌مورفیسم برای توسعه عملکرد استفاده کنید.
- ✓ رعایت OCP منجر به کدی مقیاس‌پذیر، قابل نگهداری و انعطاف‌پذیر می‌شود.

### ◆ Principle: LSP – Liskov Substitution Principle (SOLID)

◆ اصل – LSP: اصل جایگزینی لیسکوف

#### 🧠 What is LSP?

The Liskov Substitution Principle emphasizes that subclasses should be substitutable for their base classes without altering the correctness of the program. In other words, objects of a derived class must be able to replace objects of the base class seamlessly. This guarantees that derived classes honor the contracts and behaviors expected by users of the base class, ensuring consistent and reliable behavior across the hierarchy. Violation of this principle often leads to unexpected bugs and fragility in the software design.

#### 🧠 اصل LSP چیست؟

اصل جایگزینی لیسکوف بیان می‌کند که هر زیرکلاس باید بتواند جایگزین کلاس والد خود شود بدون اینکه رفتار سیستم دچار مشکل شود. این بدین معنی است که رفتار کلاس فرزند باید با انتظارات و قراردادهای کلاس والد مطابقت داشته باشد و نباید باعث رفتار غیرمنتظره شود. اگر این اصل رعایت نشود، باعث بروز خطاها و مشکلات در طراحی سلسله‌مراتبی کلاس‌ها خواهد شد و قابلیت اطمینان سیستم کاهش می‌یابد.

## Scenario Overview

We are building a payment system that supports multiple payment methods: Credit Card, PayPal, Crypto, and Cash. The wrong design hardcodes one payment method, making it impossible to substitute other types without modifying the class.

### توضیح سناریو

در حال ساخت یک سیستم پرداخت هستیم که از روش‌های مختلفی مثل کارت اعتباری، پی‌پال، رمزارز و نقدی پشتیبانی می‌کند. طراحی اشتباه فقط یک روش پرداخت را به صورت سخت‌کد شده استفاده می‌کند و امکان جایگزینی روش‌های دیگر بدون تغییر کلاس وجود ندارد.

## Before Applying LSP (Wrong Design)

```
public class PaymentWrong
{
    public void Pay()
    {
        Console.WriteLine("Paid By Credit Card");
    }
}
```

## Problems:

- Only supports one payment method.
- Cannot substitute other payment types without modifying the class.
- Violates LSP by preventing polymorphic substitution.

- فقط از یک روش پرداخت پشتیبانی می‌کند.
- برای اضافه کردن روش‌های دیگر باید کلاس را تغییر دهیم.
- اصل LSP را با جلوگیری از جایگزینی پلی‌مورفیک نقض می‌کند.

### ✅ After Applying LSP (Correct Design)

```
public interface Payment
{
    void Pay();
}

public class CreditPayment : Payment
{
    public void Pay()
    {
        Console.WriteLine("Paid By Credit Card");
    }
}

public class PayPalPayment : Payment
{
    public void Pay()
    {
        Console.WriteLine("Paid By PayPal");
    }
}

public class CryptoPayment : Payment
{
    public void Pay()
    {
        Console.WriteLine("Paid By Cryptocurrency");
    }
}
```

```
public class CashPayment : Payment
{
    public void Pay()
    {
        Console.WriteLine("Paid By Cash");
    }
}
```

### Improvements:

- Each payment method is a separate class implementing the same interface.
- We can substitute any payment type without changing the client code.
- Fully respects LSP and supports polymorphism.

بهبودها: 

- هر روش پرداخت در کلاس جداگانه‌ای پیاده‌سازی شده است.
- می‌توان هر نوع پرداخت را بدون تغییر در کد اصلی جایگزین کرد.
- اصل LSP به‌طور کامل رعایت شده و پلی‌مورفیسم پشتیبانی می‌شود.

## Comparison Table

Aspect	❌ PaymentWrong	✅ Payment Interface Design
Substitutability	❌ Not possible	✅ Fully supported
Extensibility	❌ Hard-coded logic	✅ Easy to extend
Maintainability	❌ Requires modification	✅ Add new types independently
LSP Compliance	❌ Violated	✅ Fully respected

## Key Takeaways

- ✅ Use interfaces or abstract classes to support polymorphism.
- ✅ Design classes so that they can be substituted without breaking functionality.
- ✅ LSP leads to flexible, extensible, and robust systems.

## نکات کلیدی

- ✅ از Interface یا کلاس‌های پایه برای پشتیبانی از پلی مورفیسم استفاده کنید.
- ✅ کلاس‌ها را طوری طراحی کنید که قابل جایگزینی باشند بدون اینکه عملکرد برنامه دچار مشکل شود.
- ✅ رعایت LSP منجر به سیستم‌هایی انعطاف‌پذیر، قابل توسعه و مقاوم می‌شود.

## ◆ Principle: ISP – Interface Segregation Principle (SOLID)

◆ اصل – ISP: اصل تفکیک رابطها

### What is ISP?

The Interface Segregation Principle (ISP) states that clients (classes or modules that depend on an interface) should not be forced to depend on methods they do not use. In other words, a class should not have to implement methods it does not need or use, and interfaces should be designed to be smaller and more specific to the needs of individual clients. This reduces unnecessary dependencies and results in more flexible and maintainable software systems.

### اصل ISP چیست؟

اصل جداسازی اینترفیس می‌گوید که مشتریان (کلاس‌ها یا ماژول‌هایی که از اینترفیس استفاده می‌کنند) نباید مجبور باشند به متدهایی که از آن‌ها استفاده نمی‌کنند، وابسته باشند. به عبارت دیگر، یک کلاس نباید مجبور باشد متدهایی را پیاده‌سازی کند که برای آن نیازی وجود ندارد و اینترفیس‌ها باید کوچک‌تر و تخصصی‌تر طراحی شوند تا دقیقاً نیازهای هر مشتری را پوشش دهند. این کار باعث می‌شود که وابستگی‌های غیرضروری کاهش یابد و سیستم نرم‌افزاری منعطف‌تر و قابل نگهداری‌تر شود.

### Scenario Overview

We are designing a printer system that supports different capabilities: Print, Scan, and Fax. Not all printers support all features. The wrong design forces every printer class to implement all methods—even if they don't apply.



در حال طراحی یک سیستم پرینتر هستیم که قابلیت‌هایی مثل چاپ، اسکن و فکس را پشتیبانی می‌کند. اما همه پرینترها این قابلیت‌ها را ندارند. طراحی اشتباه باعث می‌شود که همه کلاس‌ها مجبور باشند همه متدها را پیاده‌سازی کنند حتی اگر به آن‌ها نیازی نداشته باشند.

## ✗ Before Applying ISP (Wrong Design)

```
public interface Printer
{
    void Fax();
    void Scan();
    void Print();
}

public class Printer_HP : Printer
{
    public void Fax() => throw new NotImplementedException();
    public void Scan() => throw new NotImplementedException();
    public void Print() => Console.WriteLine("Print Successful");
}
```

### Problems:

- **Printer\_HP only supports printing, but is forced to implement Fax() and Scan().**
- **Leads to NotImplementedException, which breaks runtime behavior.**
- **Violates ISP by forcing unnecessary method implementations.**

- کلاس `Printer_HP` فقط قابلیت چاپ دارد، اما مجبور است متدهای `Fax()` و `Scan()` را نیز پیاده‌سازی کند.
- منجر به خطای `NotImplementedException` در زمان اجرا می‌شود.
- اصل ISP را با اجبار به پیاده‌سازی متدهای غیرضروری نقض می‌کند.

### ✓ After Applying ISP (Correct Design)

```
public interface IPrint { void Print(); }
public interface IScan { void Scan(); }
public interface IFax { void Fax(); }

public class HP_Printer : IPrint
{
    public void Print() => Console.WriteLine("Print Successful");
}

public class Samsung_Printer : IScan, IFax
{
    public void Scan() => Console.WriteLine("Scan Successful");
    public void Fax() => Console.WriteLine("Fax Successful");
}

public class Canon_Printer : IPrint, IScan, IFax
{
    public void Print() => Console.WriteLine("Print Successful");
    public void Scan() => Console.WriteLine("Scan Successful");
    public void Fax() => Console.WriteLine("Fax Successful");
}
```

### Improvements:

- Each printer class only implements the interfaces it needs.
- No unnecessary methods or exceptions.
- Fully respects ISP and improves modularity.

- هر کلاس فقط Interface هایی را پیاده‌سازی می‌کند که واقعاً به آن‌ها نیاز دارد.
- هیچ متد غیرضروری یا خطای اجرایی وجود ندارد.
- اصل ISP به‌طور کامل رعایت شده و ماژولار بودن سیستم افزایش یافته است.

## Comparison Table

Aspect	❌ Wrong Design	✅ Correct Design
Interface Size	Large, bloated	Small, focused
Method Usage	Forced unused methods	Only needed methods
Runtime Safety	Risk of exceptions	Safe and predictable
ISP Compliance	❌ Violated	✅ Fully respected

## Key Takeaways

- ✅ Break large interfaces into smaller, role-specific ones.
- ✅ Let classes implement only what they truly need.
- ✅ ISP leads to cleaner, safer, and more maintainable code.

## نکات کلیدی 📌

- ✅ Interface های بزرگ را به رابط‌های کوچکتر و تخصصی تقسیم کنید.
- ✅ اجازه دهید کلاس‌ها فقط متدهایی را پیاده‌سازی کنند که واقعاً به آن‌ها نیاز دارند.
- ✅ رعایت ISP منجر به کدی تمیزتر، ایمن‌تر و قابل نگهداری‌تر می‌شود.

## ◆ Principle: DIP – Dependency Inversion Principle (SOLID)

◆ اصل – DIP: اصل وارونگی وابستگی

### What is DIP?

The Dependency Inversion Principle requires that high-level modules should not depend on low-level modules; instead, both should depend on abstractions such as interfaces or abstract classes. Furthermore, abstractions should not depend on details; details should depend on abstractions. This principle reduces direct coupling between components and promotes flexible and reusable code. A common way to implement DIP is through dependency injection, where dependencies are provided externally (e.g., via constructors), making the codebase more maintainable and testable.

### اصل DIP چیست؟

اصل وارونگی وابستگی می‌گوید ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند، بلکه هر دو باید به انتزاع‌ها (مثل اینترفیس‌ها یا کلاس‌های انتزاعی) وابسته باشند. همچنین جزئیات نباید به انتزاع‌ها وابسته باشند بلکه بالعکس جزئیات باید به انتزاع‌ها وابسته باشند. این اصل باعث کاهش وابستگی‌های مستقیم و افزایش انعطاف‌پذیری کد می‌شود. معمولاً از تکنیک تزریق وابستگی برای پیاده‌سازی این اصل استفاده می‌شود که نگهداری و تست نرم‌افزار را آسان‌تر می‌کند.

### Scenario Overview

We are building an authentication system that sends notifications (OTP, login alerts). The wrong design tightly couples the Authenticate class to a specific notification type (EmailNotificationWrong), making it hard to extend or test.

در حال ساخت یک سیستم احراز هویت هستیم که نوتیفیکیشن‌هایی مثل OTP و هشدار ورود ارسال می‌کند. طراحی اشتباه باعث می‌شود کلاس Authenticate به یک نوع خاص از نوتیفیکیشن (EmailNotificationWrong) وابسته باشد، که توسعه و تست را دشوار می‌کند.

## ✗ Before Applying DIP (Wrong Design)

```
public class EmailNotificationWrong
{
    public void SendEmail()
    {
        Console.WriteLine("Email Send.");
    }
}

public class AuthenticateWrong
{
    private EmailNotificationWrong _notification;

    public AuthenticateWrong()
    {
        this._notification = new EmailNotificationWrong();
    }

    public void OTP()
    {
        Console.WriteLine("OTP Method Uses");
        _notification.SendEmail();
    }

    public void BaseLogin()
    {
        Console.WriteLine("Base Login Method Uses");
        _notification.SendEmail();
    }
}
```

## Problems:

- **AuthenticateWrong is tightly coupled to EmailNotificationWrong.**
- **Cannot switch to SMS or other notification types without modifying the class.**
- **Violates DIP by depending on concrete implementation instead of abstraction.**

## مشکلات:

- کلاس AuthenticateWrong به کلاس EmailNotificationWrong وابسته است.
- نمی‌توان نوع نوتیفیکیشن را بدون تغییر در کلاس اصلی عوض کرد.
- اصل DIP را با وابستگی به پیاده‌سازی خاص نقض می‌کند.

## After Applying DIP (Correct Design)

```
public interface INotification
{
    void Send();
}

public class EmailNotification : INotification
{
    public void Send()
    {
        Console.WriteLine("Email Send Successful");
    }
}
```

```

public class SMSNotification : INotification
{
    public void Send()
    {
        Console.WriteLine("SMS Send Successful");
    }
}

public class Authenticate
{
    private INotification _notification;

    public Authenticate(INotification notification)
    {
        this._notification = notification;
    }

    public void OTP()
    {
        Console.WriteLine("OTP Method Uses");
        _notification.Send();
    }

    public void BaseLogin()
    {
        Console.WriteLine("Base Login Method Uses");
        _notification.Send();
    }
}

```

## Improvements:

- **Authenticate depends on the INotification interface, not a concrete class.**
- **Easily switch between EmailNotification, SMSNotification, or others.**
- **Fully respects DIP and supports loose coupling.**

- کلاس Authenticate به Interface INotification وابسته است، نه به کلاس خاص.
- به راحتی می‌توان بین انواع نوتیفیکیشن‌ها سوییچ کرد.
- اصل DIP به طور کامل رعایت شده و وابستگی‌ها شل شده‌اند.

### 📊 Comparison Table

Aspect	❌ Wrong Design	✅ Correct Design
Coupling	Tightly coupled	Loosely coupled
Flexibility	Hard to switch notification	Easy to extend
Testability	Hard to mock	Easy to test
DIP Compliance	❌ Violated	✅ Fully respected

### 📌 Key Takeaways

- ✅ Depend on abstractions, not concrete implementations.
- ✅ Use constructor injection to provide flexibility.
- ✅ DIP leads to scalable, testable, and maintainable systems.

### 📌 نکات کلیدی

- ✅ به abstraction وابسته باشید، نه به پیاده‌سازی خاص.
- ✅ از تزریق وابستگی در constructor استفاده کنید تا انعطاف‌پذیری داشته باشید.
- ✅ رعایت DIP منجر به سیستم‌هایی مقیاس‌پذیر، تست‌پذیر و قابل نگهداری می‌شود.



## ◆ SOLID in Action: Order Management System

◆ پیاده‌سازی اصول SOLID در سیستم مدیریت سفارش

### Scenario Overview | توضیح سناریو

**We built a modular Order Management System that handles discounts, notifications, and invoice generation. Each component is designed to follow one of the SOLID principles, resulting in a clean, extensible, and testable architecture.**

ما یک سیستم مدیریت سفارش طراحی کردیم که تخفیف‌ها، نوتیفیکیشن‌ها و تولید فاکتور را مدیریت می‌کند. هر بخش از سیستم با رعایت یکی از اصول SOLID ساخته شده تا معماری‌ای تمیز، قابل توسعه و تست‌پذیر داشته باشیم.

## ✓ نحوه‌ی پیاده‌سازی هر اصل | How Each Principle Was Applied

Principle	Applied In	توضیح فارسی
SRP	<b>OrderProcessor only processes orders. Logging, discounting, and notifications are delegated to separate classes.</b>	کلاس <code>OrderProcessor</code> فقط مسئول پردازش سفارش است. سایر وظایف مثل تخفیف و نوتیفیکیشن به کلاس‌های جداگانه واگذار شده‌اند.
OCP	<b>IDiscountStrategy allows adding new discount types without modifying existing logic.</b>	با استفاده از <code>IDiscountStrategy</code> می‌توان انواع تخفیف جدید را بدون تغییر در منطق فعلی اضافه کرد.
LSP	<b>INotification implementations like Email, SMS can be substituted freely.</b>	کلاس‌های <code>Email</code> و <code>SMS</code> قابل جایگزینی هستند بدون اینکه منطق برنامه دچار مشکل شود.
ISP	<b>Invoice capabilities (Print, Email, SMS) are split into separate interfaces.</b>	قابلیت‌های فاکتور مثل چاپ، ایمیل و پیامک در <code>Interface</code> های جداگانه تفکیک شده‌اند.
DIP	<b>OrderProcessor depends on abstractions, not concrete classes.</b>	کلاس <code>OrderProcessor</code> به <code>abstraction</code> وابسته است، نه به پیاده‌سازی خاص. وابستگی‌ها از بیرون تزریق می‌شوند.

## 🧠 Architectural Benefits | مزایای معماری

- ☒ Each class has a single, clear responsibility
- ☒ Easy to extend with new discount or notification types
- ☒ Fully testable with mock interfaces
- ☒ Flexible invoice handling via composition
- ☒ Clean separation of concerns

- ☒ هر کلاس فقط یک مسئولیت مشخص دارد
- ☒ افزودن انواع جدید تخفیف یا نوتیفیکیشن بسیار ساده است
- ☒ تست پذیری کامل با استفاده از mock interface ها
- ☒ مدیریت فاکتور به صورت ترکیبی و انعطاف پذیر
- ☒ جداسازی کامل وظایف و منطقها

## 📌 Final Thoughts | جمع بندی نهایی

- ☒ This project proves that SOLID is not just theory—it's a practical mindset for building scalable and maintainable systems.
- ☒ By applying each principle in a real-world scenario, we created a clean and professional architecture ready for production and teaching.

- ☒ این پروژه نشان می دهد که SOLID فقط یک نظریه نیست—بلکه یک طرز فکر عملی برای ساخت سیستم های مقیاس پذیر و قابل نگهداری است.
- ☒ با پیاده سازی هر اصل در یک سناریوی واقعی، معماری ای تمیز و حرفه ای ساختیم که هم برای تولید آماده است و هم برای آموزش.



## Full Code: SOLID-Based Order Management System



Covers: SRP, OCP, LSP, ISP, DIP

```
// ☒ SRP & DIP – Order Processor only handles orchestration, not business logic
public class OrderProcessor
{
    private readonly IDiscountStrategy discount;
    private readonly INotification notification;
    private readonly InvoiceHandle invoice;

    public OrderProcessor(IDiscountStrategy discount, INotification notification,
InvoiceHandle invoice)
    {
        this.discount = discount;
        this.notification = notification;
        this.invoice = invoice;
    }

    public void Process(Order order)
    {
        discount.ApplyDiscount(order);
        notification.Send(order);
        invoice.Handle();
        Console.WriteLine("process Finished!");
    }
}

// ☒ Entity – Order Model
public class Order
{
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; } = DateTime.Now;
    public decimal TotalPrice { get; set; }
    public decimal Discount { get; set; }
    public decimal NetPrice { get; set; }
}

// ☒ OCP – Discount Strategy Interface & Implementations
public interface IDiscountStrategy
{
    void ApplyDiscount(Order order);
}
```

```
public class BlackFridayDiscount : IDiscountStrategy
{
    public void ApplyDiscount(Order order)
    {
        order.Discount = (order.TotalPrice / 100) * 35;
        order.NetPrice = order.TotalPrice - order.Discount;
    }
}
```

```
public class OutletDiscount : IDiscountStrategy
{
    public void ApplyDiscount(Order order)
    {
        order.Discount = (order.TotalPrice / 100) * 50;
        order.NetPrice = order.TotalPrice - order.Discount;
    }
}
```

// ☒ LSP – Notification Interface & Implementations

```
public interface INotification
{
    void Send(Order order);
}
```

```
public class Email : INotification
{
    public void Send(Order order)
    {
        Console.WriteLine($"Email Sent. For Order : {order.OrderId}");
    }
}
```

```
public class SMS : INotification
{
    public void Send(Order order)
    {
        Console.WriteLine($"SMS Sent. For Order : {order.OrderId}");
    }
}
```

// ☒ ISP – Segregated Interfaces for Invoice Capabilities

public interface IPrintable

```
{  
    void Print();  
}
```

public interface ISMSable

```
{  
    void SendSMS();  
}
```

public interface IEmailable

```
{  
    void SendEmail();  
}
```

// ☒ ISP – Invoice Implementations

public class InvoiceAll : IPrintable, ISMSable, IEmailable

```
{  
    public void Print() => Console.WriteLine("Invoice Printed");  
    public void SendEmail() => Console.WriteLine("Invoice sent via Email");  
    public void SendSMS() => Console.WriteLine("Invoice sent via SMS");  
}
```

public class InvoicePS : IPrintable, ISMSable

```
{  
    public void Print() => Console.WriteLine("Invoice Printed");  
    public void SendSMS() => Console.WriteLine("Invoice sent via SMS");  
}
```

public class InvoicePE : IPrintable, IEmailable

```
{  
    public void Print() => Console.WriteLine("Invoice Printed");  
    public void SendEmail() => Console.WriteLine("Invoice sent via Email");  
}
```

// ☒ ISP + DIP – Invoice Handler via Constructor Injection

```
public class InvoiceHandle
```

```
{  
    private readonly IPrintable? _printable;  
    private readonly ISMSable? _smsable;  
    private readonly IEmailable? _emailable;
```

```
    public InvoiceHandle(IPrintable? printable = null, ISMSable? smsable = null, IEmailable?  
emailable = null)
```

```
{  
    _printable = printable;  
    _smsable = smsable;  
    _emailable = emailable;  
}
```

```
public void Handle()  
{  
    _printable?.Print();  
    _smsable?.SendSMS();  
    _emailable?.SendEmail();  
}
```

```
}
```

// ☒ Execution – Putting It All Together

```
Order order = new Order
```

```
{  
    OrderId = 1,  
    TotalPrice = 1500  
};
```

```
InvoiceHandle invoice = new InvoiceHandle(new InvoicePE());
```

```
OrderProcessor op = new OrderProcessor(  
    discount: new BlackFridayDiscount(),  
    notification: new Email(),  
    invoice: invoice  
);
```

```
op.Process(order);
```



## Sample Output

Order Detail => ID : 1 | TotalPrice : 1500 | Discount : 0 | NetPrice : 0

---

Email Sent. For Order : 1

Invoice Printed

process Finished!

---

Order Detail => ID : 1 | TotalPrice : 1500 | Discount : 525 | NetPrice : 975