

S.O.L.I.D.



Solids



What is Solids ?

SOLID is an acronym for five fundamental design principles in object-oriented programming (OOP) that improve software maintainability and scalability. These principles were introduced by **Robert C. Martin (Uncle Bob)** and help developers write cleaner, more flexible, and robust code. 🚀



What is The five SOLID principles?

- 1 Single Responsibility Principle (SRP)
- 2 Open/Closed Principle (OCP)
- 3 LISKOV'S Substitution Principle (LSP)
- 4 Interface Segregation Principle (ISP)
- 5 Dependency Inversion Principle (DIP)



What problems might occur if we do not follow the SOLID principles in software development?



- 1 **Tightly Coupled Code** – Making changes in one part of the system may break other parts.
- 2 **Difficult Maintenance** – Code becomes harder to modify, fix, or extend.
- 3 **Low Reusability** – Components cannot be reused efficiently in different contexts.
- 4 **Violation of Open/Closed Principle** – Every new feature may require modifying existing code, increasing the risk of bugs.
- 5 **Hard-to-Test Code** – Unit testing becomes complex due to dependencies and tightly coupled components.
- 6 **Poor Scalability** – As the project grows, adding new features becomes more challenging and error-prone.
- 7 **Inflexible Design** – Lack of abstraction makes it difficult to introduce new requirements or changes.

1

Single Responsibility Principle (SRP) 📌

✍️ تخيل معايا ان في شخص بيشتغل في مكتب وبيعمل اكتر من وظيفة في نفس الوقت

1. بيكتب تقارير
2. بيخزن ملفات
3. بيرسل ايميلات للعملاء

دلوقتي لو الشخص دا عاوز يعدل في طريقة ارسال الايميلات هيضطر يروح يغير في كل 🤔
حاجة طبليه ؟

لأن بيقوم بكل المهام دي في نفس الوقت وفي نفس المكان

🤔 طب والحل اي يا دولي ؟

👤 الحل ان نعين موظف لكل مهمه:

واحد بيكتب تقارير فقط وواحد بيخزن الملفات فقط وواحد بيرسل ايميلات فقط

🤔 طب احنا كدا استفدنا اي ؟

👤 كدا لو عوزين نعدل في ارسال الايميلات المسؤول عن كدا هو شخص واحد:


وبالتالي هيعدل في مكان واحد ومش هياثر علي الباقي لأن





وبس كدا يا دولي تعالي بقا ناخذ الموضوع في البرمجة

Single Responsibility Principle (SRP) 📖











The **Single Responsibility Principle** says:

"A class should have only one reason to change." 

This means:

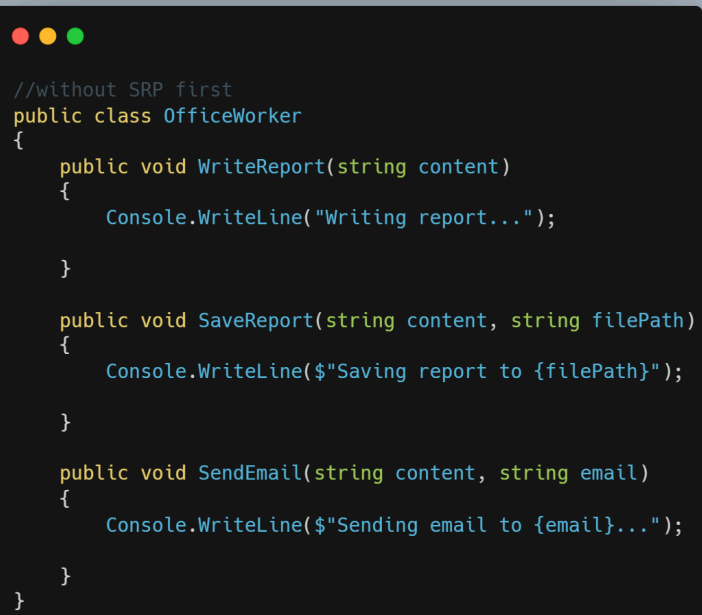
- **One responsibility:** Each class or component should only handle one job or task. 
- **Focus:** Keep things simple and focused on a single responsibility. 

Why is SRP Important?

1.  **Easier Maintenance:** With one responsibility per class, modifications are isolated and don't affect other parts of the code. 
2.  **Reusability:** A class focused on one task can be easily reused in other parts of the system. 
3.  **Reduced Risk:** If a class handles one responsibility, changes to that responsibility don't break unrelated code. 
4.  **Less Complexity:** Keeping classes simple and focused makes the system easier to understand and manage. 
5.  **Improved Testing:** It's easier to write tests for classes with one responsibility, leading to better coverage. 

How to Implement SRP in C#?

1. **without SRP first** 



```
//without SRP first
public class OfficeWorker
{
    public void WriteReport(string content)
    {
        Console.WriteLine("Writing report...");
    }

    public void SaveReport(string content, string filePath)
    {
        Console.WriteLine($"Saving report to {filePath}");
    }

    public void SendEmail(string content, string email)
    {
        Console.WriteLine($"Sending email to {email}...");
    }
}
```

2. Applying SRP (Better Approach) ✓

```

public class ReportWriter
{
    public string WriteReport(string content)
    {
        Console.WriteLine("Writing report...");

        return content;
    }
}

public class ReportSaver
{
    public void SaveReport(string content, string filePath)
    {
        Console.WriteLine($"Saving report to {filePath}");
    }
}

public class EmailSender
{
    public void SendEmail(string content, string email)
    {
        Console.WriteLine($"Sending email to {email}...");
    }
}

public class OfficeWorker
{
    private readonly ReportWriter _reportWriter;
    private readonly ReportSaver _reportSaver;
    private readonly EmailSender _emailSender;

    public OfficeWorker(ReportWriter reportWriter, ReportSaver reportSaver,
        EmailSender emailSender)
    {
        _reportWriter = reportWriter;
        _reportSaver = reportSaver;
        _emailSender = emailSender;
    }

    public void CompleteTask(string reportContent, string filePath,
        string email)
    {
        var report = _reportWriter.WriteReport(reportContent);
        _reportSaver.SaveReport(report, filePath);
        _emailSender.SendEmail(report, email);
    }
}

//using SRP
public class Program
{
    public static void Main()
    {
        var reportWriter = new ReportWriter();
        var reportSaver = new ReportSaver();
        var emailSender = new EmailSender();
        var officeWorker =
            new OfficeWorker(reportWriter, reportSaver, emailSender);

        officeWorker.CompleteTask("my report",
            "path/to/file", "emailaddress@gmail.com");
    }
}

```




2

Open/Closed Principle (OCP)

✍️ افترض معايا انك شغال علي موقع تجارة الكترونية وعندك طرق دفع واعوز تزود واحدة كمان فعشان تعمل كذا التعديل هياثر علي باقي الكود لان كلو معتمد علي بعضو

لو تفتكر حاجة في Operating Systems كان اسمها Tightly Couple

😬 طب والحل اي يا دولي ؟

الحل انك تقلل الاعتمادية بحيث كل جزء من الكود ميكنش معتمد علي الاخر: 


ودا هيفدنا لو هنعمل تعديل او اضافة هنروح نعدل في الجزء المطلوب او نضيف الوظيفة الجديدة بدون متأثر علي باقي الكود يعني هنعول من **Tightly Couple** to **loosely couple**

وبكدا يكون الكود بتاعك

مغلق للتعديل ومفتوح للإضافة

وبردو احنا هنا هنطبق مبدأ من مبادئ ال OOP وهو ال Abstraction

😬 طب هنطبق ال OCP ازاى؟

 هنستخدم اما ال Interface or abstract class

: وبردو هنا



وبس كذا يا دولي تعالي بقا ناخذ الموضوع في البرمجة

Open/Closed Principle (OCP)

The **Open/Closed Principle** says:

"Software should be open for extension but closed for modification." 🔒❌

This means:

- **Open for extension:** You can **add new features** or **extend functionality** easily ✨.
- **Closed for modification:** Don't change the existing code when adding new features 🔒.

Why is OCP Important?

1. 🚀 **Flexibility:** Add new features without breaking old ones.
2. 💪 **Stability:** Existing code stays the same, no bugs! 🐛
3. 🛠️ **Easier Maintenance:** Changing code doesn't affect other parts.
4. 🛠️ **Scalable:** Easily add or change functionality when the app grows
5. 🛡️ **Closed for modification:** Keep existing code safe.

How to Implement SRP in C#? 🖥️

1. Example of Violating OCP (Bad Approach)❌

```
public class PaymentProcessor
{
    public void ProcessPayment(string paymentMethod)
    {
        switch(paymentMethod)
        {
            case "CreditCard":
                Console.WriteLine("Processing credit card payment...");
                break;
            case "PayPal":
                Console.WriteLine("Processing PayPal payment...");
                break;
        }
    }
}
```

to add new payment

```
public class PaymentProcessor
{
    public void ProcessPayment(string paymentMethod)
    {
        switch(paymentMethod)
        {
            case "CreditCard":
                Console.WriteLine("Processing credit card payment...");
                break;
            case "PayPal":
                Console.WriteLine("Processing PayPal payment...");
                break;
            case "Bitcoin":
                Console.WriteLine("Processing Bitcoin payment...");
                break;
        }
    }
}
```

2. Applying OCP (Good Approach) ✓

```

//1. Define an interface for payment methods:

public interface IPaymentMethod
{
    void ProcessPayment();
}

//Create specific payment method classes that implement the interface:

public class CreditCardPayment : IPaymentMethod
{
    public void ProcessPayment()
    {
        Console.WriteLine("Processing credit card payment...");
    }
}

public class PayPalPayment : IPaymentMethod
{
    public void ProcessPayment()
    {
        Console.WriteLine("Processing PayPal payment...");
    }
}

//Now, if we need to add a new payment method ( Bitcoin),
//we can create a new class implementing IPaymentMethod without
//modifying the PaymentProcessor class.

public class BitcoinPayment : IPaymentMethod
{
    public void ProcessPayment()
    {
        Console.WriteLine("Processing Bitcoin payment...");
    }
}

//Modify the PaymentProcessor class to work with the interface:

public class PaymentProcessor
{
    public void ProcessPayment(IPaymentMethod paymentMethod)
    {
        paymentMethod.ProcessPayment();
    }
}

```

Let's pay



```
static void Main()
{

    PaymentProcessor processor = new PaymentProcessor();

    //if i want to pay by PayPal
    PayPalPayment paypal=new PayPalPayment();

    //let's pay
    processor.ProcessPayment(paypal);

    //output
    //Processing PayPal payment...

}
```

3

LISKOV'S Substitution Principle (LSP)

✍️ الفكرة بتعتو ببساطة ان كل واحد بيعرف يعمل حاجة ممكن غيرو ميقدر يعملها وفي نفس الوقت غيرو عندو ميزة مش موجودة عند الآخر

👤 تخيل معايا كدا عندك سيارة ودراجة فانت لما بتركب السيارة بتشغل المحرك الاول: طب هل نفس الموضوع ينفع تعملو الدراجة طبعا لا دي بتعمل حاجة والتانية تقدر تعمل حاجة مختلفة طب 😊 ازاى اطبق دا في الكود بتاعي بحيث اراعي الجزء دا

👤 عن طريق تطبيق مبدأ من مبادئ ال OOP وهو ال Polymorphism

وتستعين بال Inheritance

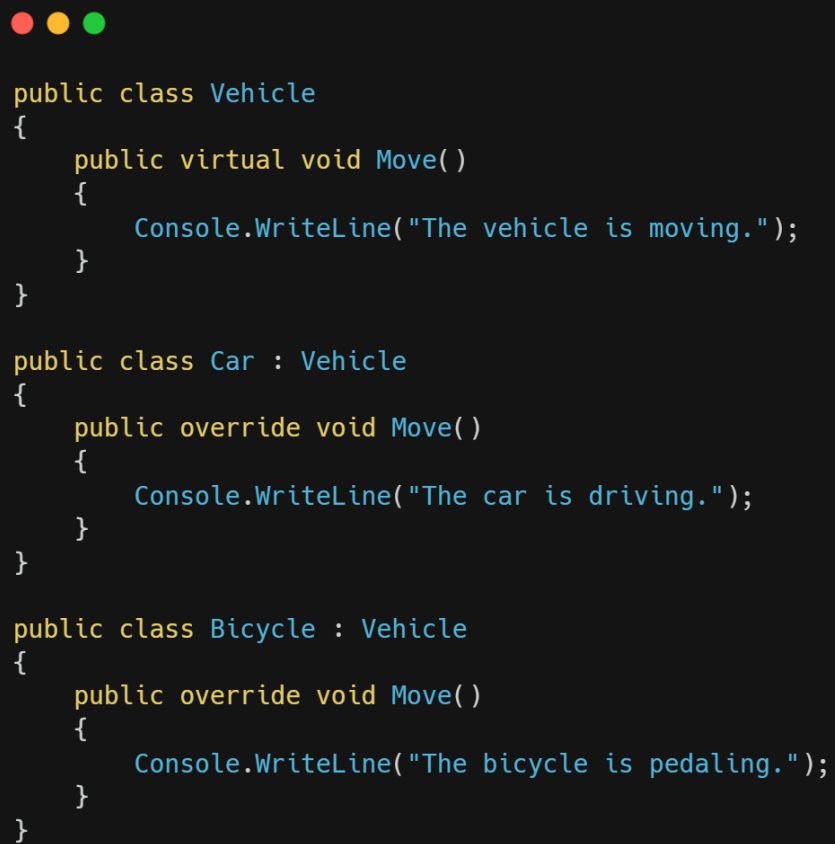
👤 يبقى كدا كل كلاس هيبقي مثلا بينفذ نفس الدالة ب Logic مختلف عن الآخر : بمعنى ان الكلاسات الفرعية لازم تقدر تحل محل الاب بدون ما يحصل مشاكل فالحل تحافظ علي تنسيق الكود بحيث ان الكود الي بيستخدمو الاب يقدر يشغل بالابناء طب تعالى ناخذ مثال عشان نفهم اكثر

1. Without LSP ❌ : here bad Logic

```
public class Vehicle
{
    public virtual void StartEngine()
    {
        Console.WriteLine("Starting the vehicle's engine.");
    }
}

public class Bicycle : Vehicle
{
    public override void StartEngine()
    {
        throw new InvalidOperationException("Bicycles don't have engines!");
    }
}
```

2. With LSP ✅

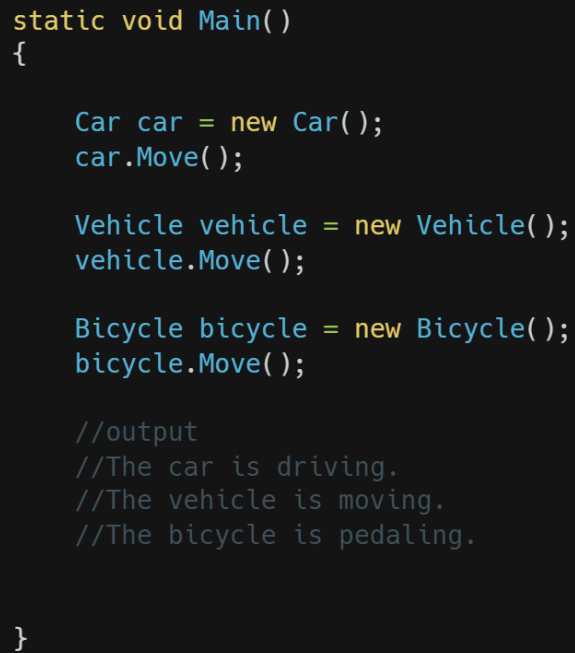


```
public class Vehicle
{
    public virtual void Move()
    {
        Console.WriteLine("The vehicle is moving.");
    }
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The car is driving.");
    }
}

public class Bicycle : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The bicycle is pedaling.");
    }
}
```


Try Code



```
static void Main()  
{  
  
    Car car = new Car();  
    car.Move();  
  
    Vehicle vehicle = new Vehicle();  
    vehicle.Move();  
  
    Bicycle bicycle = new Bicycle();  
    bicycle.Move();  
  
    //output  
    //The car is driving.  
    //The vehicle is moving.  
    //The bicycle is pedaling.  
  
}
```

4

Interface Segregation Principle (ISP)

 افترض معايا ان انت قاعد مع روبوت والروبوت قادر يمشي يبقي كدا قدر يعمل حاجة زي الانسان طب هل يعقل تبقي بتاكل وتعزم عليه 😊 وتطلب معاك

😂 ان لازم هياكل هل دا منطقي?

طبعا لا نفس الكلام فالكود لو مثلا عامل interface وفيه مجموعة دوال

وبدأت تورثو لمجموعة كلاسات لازم تراعي ان كل كلاس هيكون محتاج الدوال

مبيقاش في دالة معمولة في ال interface لمجرد ان كلاس واحد هiestخدمها

!! لدام كدا ليه مخلي الباقي يطبقها

في الحالة دي هنعمل اكثر من interface وكل واحد هيبقي في مجموعة الدوال

الي هيجتجها كلاس معين ونورث كل interface للكللاس الي محتاجو



وبس كدا يا دولي تعالي ناخذ الموضوع فالبرمجة

Interface Segregation Principle (ISP)





The **Interface Segregation Principle** says:

"Clients should not be forced to depend on interfaces they do not use."  

This means:

- **Split large interfaces into smaller, more specific ones:** Rather than having a big interface that forces a class to implement methods it doesn't need, break it down into smaller, more focused interfaces. 
- **Only implement what's needed:** A class should only implement the methods that are relevant to it. If it doesn't need a particular method, don't force it to implement it. 

Why is ISP Important?

1.  **Flexibility:** Smaller, focused interfaces allow for better customization and flexibility in how you build classes.
2.  **Maintainability:** Easier to maintain and extend your classes when they don't depend on unnecessary methods.
3.  **Cleaner Code:** By separating concerns, your code becomes more readable and easier to understand.
4.  **Loose Coupling:** It reduces dependencies between classes, making it easier to change one part of the code without affecting others.

Example

1. Bad Example (Violates ISP) ❌:

```
public interface IWorker
{
    void Work();
    void Eat();
}

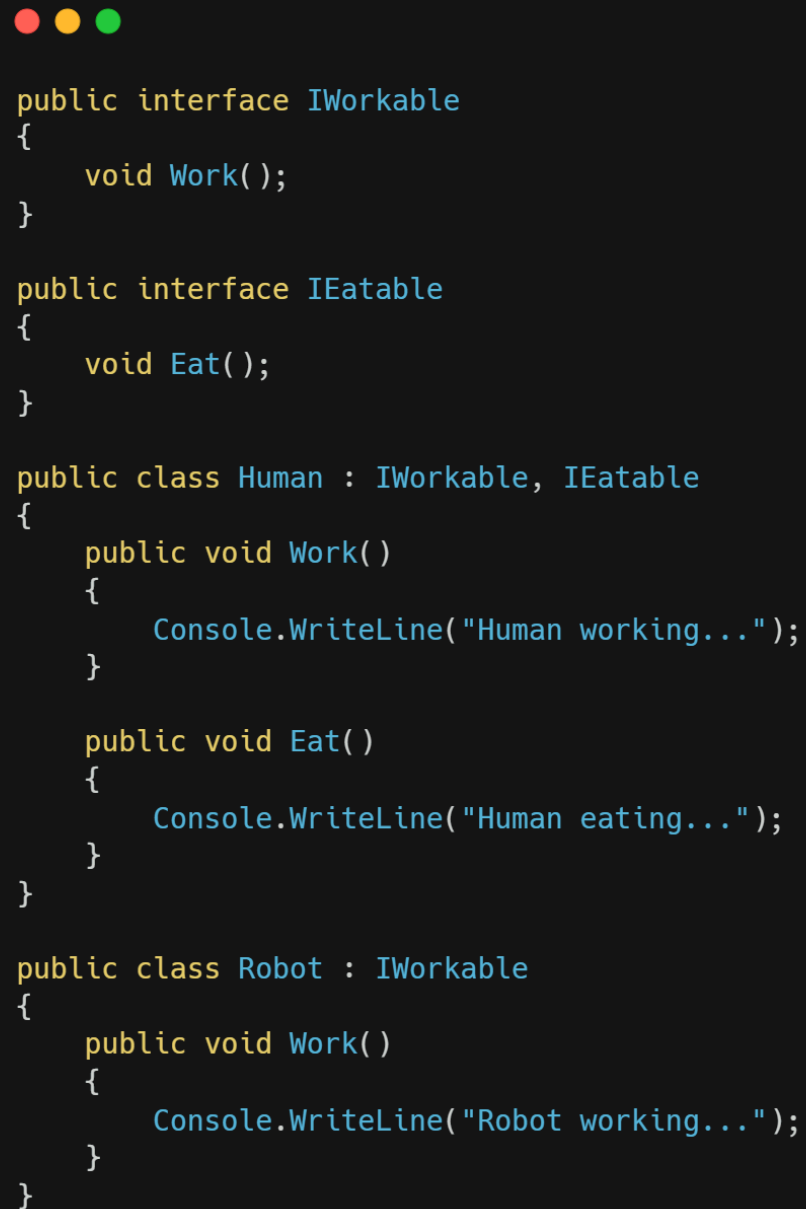
public class Human : IWorker
{
    public void Work()
    {
        Console.WriteLine("Human working...");
    }

    public void Eat()
    {
        Console.WriteLine("Human eating...");
    }
}

public class Robot : IWorker
{
    public void Work()
    {
        Console.WriteLine("Robot working...");
    }

    // The robot doesn't need to eat, but it still implements this method!
    public void Eat()
    {
        throw new NotImplementedException("Robots don't eat!");
    }
}
```

2. Good Example (Follows ISP) ✅:



```
public interface IWorkable
{
    void Work();
}

public interface IEatable
{
    void Eat();
}

public class Human : IWorkable, IEatable
{
    public void Work()
    {
        Console.WriteLine("Human working...");
    }

    public void Eat()
    {
        Console.WriteLine("Human eating...");
    }
}

public class Robot : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Robot working...");
    }
}
```


5

Dependency Inversion Principle (DIP)

تخيل إن عندك كافيه ☕ ييقدم مشروبات زي قهوة، شاي، عصير 🍹
 في الأول، الكاشير كان بيطلب دايمًا قهوة
 من الماكينة مباشرة، فبقى فيه
 اعتماد قوي بين الكاشير والماكينة
 الكاشير معتمد مباشرة على ماكينة القهوة، فلو غيرنا الماكينة أو حبيننا نضيف مشروبات
 تانية، هحتاج نعدل في كود الكاشير نفسه
 🤔 طب والحل إي ؟
 بدل ما الكاشير يكون معتمد مباشرة على ماكينة القهوة، هنستخدم واجهة تمثل أي
 مشروب، وبكده الكاشير يقدر يطلب أي مشروب بدون ما يعرف تفاصيله
 يبقى مش احسن حاجة ان كلاس يعتمد علي كلاس اخر الافضل يعتمد علي interface
 بحيث تقدر تغير التنفيذ بسهولة بدون ما تكسر الكود كلاً
 وبس كذا يا دولي تعالي بقي ناخذ الموضوع في البرمجة

Dependency Inversion Principle (DIP)

📌 The Principle States:

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

What Does This Mean? 🤔

- Code should depend on **Interfaces and Abstractions** instead of specific classes.
- The goal is to **reduce tight coupling**, making it easier to modify and extend without breaking other parts of the code.

⚠️ Problem Without DIP

Imagine your code is tightly coupled, meaning a small change in one class affects everything else.

Example:

1. ❌ **Bad Example (Violates DIP - Tightly Coupled Code)**

```

using System;

public class CoffeeMachine
{
    public void BrewCoffee()
    {
        Console.WriteLine("☕ Brewing coffee...");
    }
}

public class Cashier
{
    private CoffeeMachine _coffeeMachine = new CoffeeMachine();

    public void ServeCoffee()
    {
        _coffeeMachine.BrewCoffee();
        Console.WriteLine("✅ Coffee is ready!");
    }
}

class Program
{
    static void Main()
    {
        Cashier cashier = new Cashier();
        cashier.ServeCoffee();
    }
}

```

Problems Before Using DIP:

- ❌ **Tightly coupled:** `Cashier` is tied to `CoffeeMachine`.
- ❌ **Hard to extend:** Adding tea or juice requires modifying `Cashier`.
- ❌ **Difficult to test:** We cannot replace `CoffeeMachine` easily in unit tests.

2. ✅ Solution Using DIP (Decoupled & Flexible Code)

```

using System;

// Abstraction for any drink
public interface IDrink
{
    void Prepare();
}

// Concrete implementation for coffee
public class Coffee : IDrink
{
    public void Prepare()
    {
        Console.WriteLine("☕ Brewing coffee...");
    }
}

// Concrete implementation for tea
public class Tea : IDrink
{
    public void Prepare()
    {
        Console.WriteLine("🍵 Brewing tea...");
    }
}

// Cashier depends on abstraction (IDrink) instead of a specific drink
public class Cashier
{
    private readonly IDrink _drink;

    public Cashier(IDrink drink)
    {
        _drink = drink;
    }

    public void ServeDrink()
    {
        _drink.Prepare();
        Console.WriteLine("✅ Your drink is ready!");
    }
}

class Program
{
    static void Main()
    {
        // Now, we can serve any drink without modifying the Cashier class!
        Cashier cashier1 = new Cashier(new Coffee());
        cashier1.ServeDrink();

        Cashier cashier2 = new Cashier(new Tea());
        cashier2.ServeDrink();
    }
}

```

Benefits After Using DIP:

- ✅ **Loosely coupled:** `Cashier` does not depend on a specific drink class.
- ✅ **Easily extendable:** We can add `Juice` without modifying `Cashier`.

✅ **Better for testing:** We can mock `IDrink` for unit tests.

Now, our coffee shop is scalable and flexible! 🚀



Summary




المبدأ	معناه	لو ما طبقناهوش	الحل بالمبدأ
Single Responsibility Principle (SRP)	كل كلاس يكون ليه وظيفة واحدة بس ✂️	أي تعديل في جزء ممكن يبور حاجات ثانية، وصعوبة الصيانة 🛠️	نقسم الكود بحيث كل كلاس يكون ليه مسؤولية واحدة فقط ✅
Open/Closed Principle (OCP)	الكود يكون مفتوح للإضافة ومغلق للتعديل 🔒❌	كل مرة نضيف ميزة جديدة نضطر نعدل في الكود الأصلي، فيحصل مشاكل أو Bugs 🐛	زي الـ Abstraction نستخدم عشان نضيف Interfaces ميزات جديدة بدون ما نعدل في الكود القديم ✅
LISKOV'S Substitution Principle (LSP)	الكائنات اللي ترث من كلاس لازم تقدر تحل محله بدون مشاكل 🔄	الكود ممكن ينهار لو الكلاس الابن مش متوافق مع الأب ⚡	نخلي الوراثة منطقية ونراعي إن أي كلاس فرعي يقدر يتبدل مكان الأب بسهولة ✅
Interface Segregation Principle (ISP)	الكلاسات لازم تستخدم الواجهات اللي محتاجها بس 🎯	الكود بيبقى معقد وكلاسات كثير فيها دوال مش بتستخدمها 📄	صغيرة بدل Interfaces نعمل Interface ما نحط كل حاجة في واحدة واحدة ✅
Dependency Inversion Principle (DIP)	الكود يعتمد على Abstractions مش على تفاصيل محددة 🐛	أي تغيير في التفاصيل يخرب الكود كله 😞	نخلي الكود يعتمد على Interfaces بدل الاعتماد المباشر على كلاس معين ✅

بمعنى بسيط: لو طبقنا SOLIDS 💡 الكود بتاعنا هيبقى مرن، قابل للتطوير، سهل الصيانة

ومفيهوش Bugs كثير 🚀✅

With my best wishes

My personal accounts links

 LinkedIn	https://www.linkedin.com/in/ahmed-hany-899a9a321? utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=android_app
 WhatsApp	https://wa.me/gr/7KNUQ7Zl3KO2N1
 Facebook	https://www.facebook.com/share/1NFM1PfSjc/