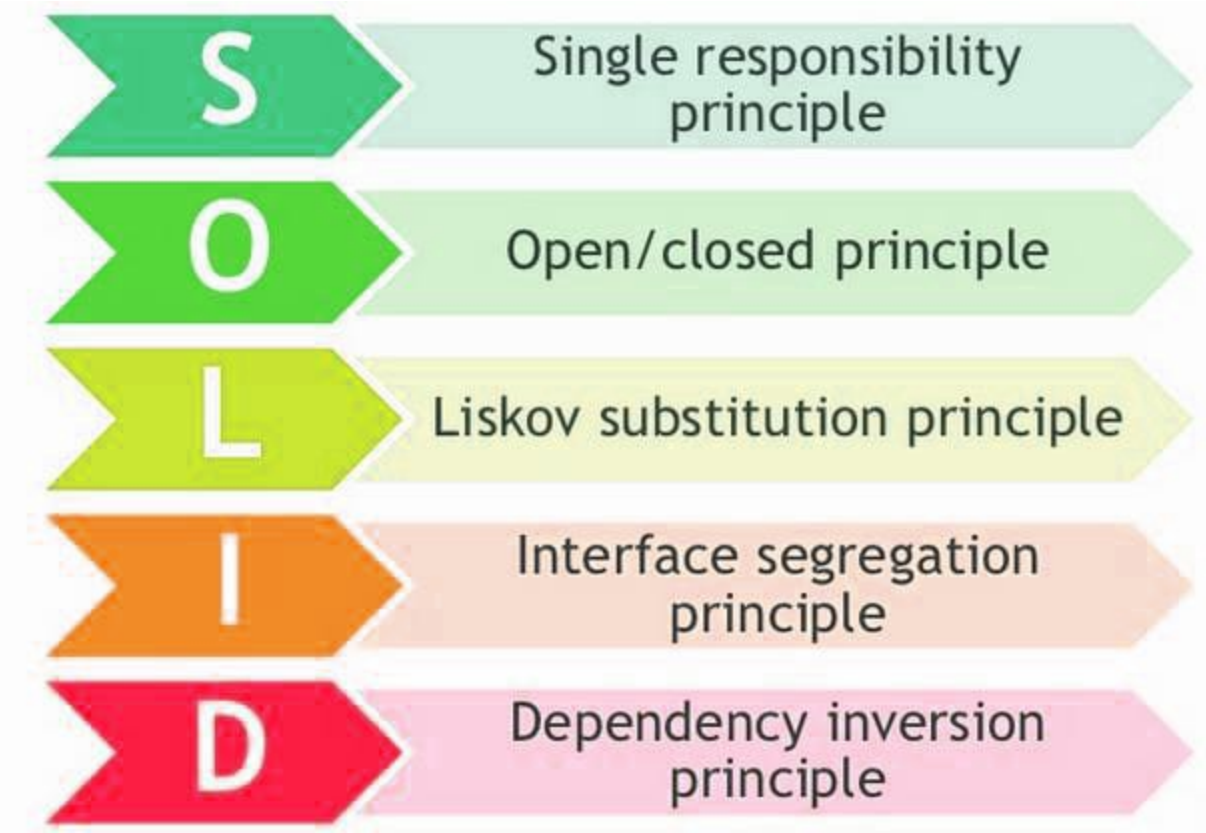
A Solid Guide to S.O.L.I.D Principles. Foundation for modern software architecture .

Mukul Attavania Follow

Jan 22 · 17 min read



Why you should use SOLID design principles

As software developers, the natural tendency is to start developing applications based on your own hands-on experience and knowledge right away. However, overtime issues in the application arise, adaptations to changes, and new features happen. Since then, you gradually realize that you have put too much effort into one thing: modifying the application. Even when implementing a simple task, it also requires understanding the whole system. You can't blame them for changes or new features since they are inevitable parts of software development. So, what is the main problem here?

The obvious answer could be derived from the application's design. Keeping the system design as clean and scalable as possible is one of the critical things that any professional developer should dedicate their time to. And that's where SOLID design principles come into play. It helps developers eliminate design smells and build the best designs for a set of features.

Although the SOLID design principles were first introduced by the famous Computer Scientist Robert C. Martin (a.k.a. Uncle Bob) in his paper in 2000, its acronym was introduced later by Michael Feathers. Uncle Bob is also the author of best-selling books Clean Code, Clean Architecture, Agile Software Development: Principles, Patterns, and Practices.

Why do we need SOLID Design Principles?

As a developer, we start developing applications using our experience and knowledge. But over time, the applications might arise bugs. We need to alter the application design for every change request or for a new feature request. After some time we might need to put in a lot of effort, even for simple tasks, it might require the full working knowledge of the entire system. But we can't blame change or new feature requests as they are part of the software development. We can't stop them and refuse them either. So who is the culprit here? Obviously, it is the design of the application.

Advantages of SOLID design principles in C#

The SOLID are long-standing principles used to manage most of the software design problems you encounter in your daily programming process.

Whether you are designing or developing the application, you can leverage the following advantages of SOLID principles to write code in the right way.

- **Maintainability**: So far, maintenance is vital for any organization to keep high quality as a standard in developing software. As the business has been growing and the market requires more changes, the software design should be adapted to future modifications at ease.
- **Testability**: When you design and develop a large-scale application, it's essential to build one that facilitates testing each functionality early and easily.
- **Flexibility and scalability**: Flexibility and scalability are the foremost crucial parts of enterprise applications. As a result, the system design should be adaptable to any

later update and extensible for adding new features smoothly and efficiently.

- **Parallel development**: Parallel development is one of the most key factors for saving time and costs in software development. It could be challenging for all team members to work on the same module at the same time. That's why the software needs to be broken down into various modules which allow different teams to work independently and simultaneously.

What's a STUPID codebase?

> A STUPID codebase is that codebase which has flaws or faults, which affect the maintainability, readability or efficiency.
>
> **Anti-Pattern Code == STUPID Code**

What causes STUPID codebase?

Why be STUPID, when you can be SOLID

- ***Singleton***: Violation of Singleton basically decreases the flexibility and reusability of the existing code, which deals with the object creation mechanism. It is an anti-pattern, where we define a class and its object in the same script/file and export the object for reusability. This is pattern is not wrong, but using it everywhere inappropriately is an symptom sick codebase.

```
/**
*
*  Creating class Singleton, which is an Anti Pattern
*  definition.
```

```
*
*  WHY?
*  Let us see.
*/
class Singleton {
  private static instance: Singleton;
  private _value: number;  /**
  * To avoid creating objects directly using 'new'
  * operator
  *
  * Therefore, the constructor is accessible to class
  * methods only
  */
  private constructor() { }   /**
  * Defining a Static function, so to directly
  *  make it accessible without creating an Object
  */
  static makeInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
      Singleton.instance._value = 0;
    }
    return Singleton.instance;
  }  getValue (): number {
    return this._value;
  }  setValue(score) {
    this._value = score;
  }
  incrementValueByOne(): number {
    return this._value += 1;
  }
}
/**
*  Since the Singleton class's constructor is private, we
*  need to create an instance using the static method
*  makeInstance()
*
*  Let us see what anomalies does that cause.
*
*  Creating an instance using 'new' throw an Error
*  Constructor of class 'Singleton' is private and
*  only accessible within the class declaration
*  const myInstance = new Singleton();
*/const myInstance1 = Singleton.makeInstance();
const myInstance2 =
Singleton.makeInstance();console.log(myInstance1.getValue()); //
OUTPUT: 0
console.log(myInstance2.getValue()); // OUTPUT: 0
myInstance1.incrementValueByOne(); // value = 1
myInstance2.incrementValueByOne(); // value =
2console.log(myInstance1.getValue()); // OUTPUT: 2
console.log(myInstance2.getValue()); // OUTPUT: 2/**
* This is the issue Singleton Anti-Pattern
* causing Issue with Singleton Pattern
*/
```

- ***Tight-Coupling***: Excessive coupling/dependency between classes or different separate functionality is a code smell, we need to be very careful about while we are developing or programming. We can figure tight-coupling when a method accesses the data of another object more than its own data or some sort of functional chaining scenarios.

```
/**
* A simple example for Tight-Coupling
*/class Car {  move() {
    console.log("Car is Moving");
  }}class Lorry {   move(){
      console.log("Lorry is Moving");
   }}class Traveller1 {  Car CarObj = new Car();  travellerStatus(){
     CarObj.move();
  }    }class Traveller2 {  Lorry LorryObj = new Lorry();
travellerStatus(){
     CarObj.move();
  }    }
```

- ***Untestabiility***: Unit Testing is a very important part of software development where you cross-check and test if the component you built is functioning exactly the way expected. It is always advised to ship a product only after writing test cases.
  Shipping an untested code/product is very much similar to deploying an application whose behavior you are not sure about.
  Apart from Unit testing, we have other tests like Integration testing, E2E testing, and so on, which are done based on their use cases and necessity.

- ***Premature Optimizations***: Avoid refactoring code if it doesn't improve the readability or performance of the system for no reason.
  Premature optimization can also be defined as trying to optimize the code, expecting it to improvise the performance or readability without having much data assuring it, and purely weighing upon intuitions.

- ***In-descriptive Naming***: Descriptive Naming and Naming Conventions are two important criteria. Most of the time, naming becomes the most painful issue.
  After some time when you or another developer visits the codebase, you would be asking the question 'What does this variable do?'. We fail to decide what would be the best descriptive name that can be given to a variable, class, class object/instance, or function. It is very important to give a descriptive name, for better readability and understandability.

```
/**
* Example for adding two numbers: Avoid this
*/
function a(a1,a2) { // It is less descriptive in nature
  return a1 + a2;
}console.log(a(1,2)); // It is less descriptive in nature
/**
* Example for adding two numbers: Better Approach
*/
function sum(num1,num2) {  // sum() is descriptive
  return num1 + num2;
}console.log(sum(1,2));
// Statement is descriptive in nature
```

- ***Duplication***: Sometimes, duplication of code is resultant of copy and paste. Violation of the DRY principle causes code-duplication. Always advised not to replicate the code across the codebase, as in long run causes huge technical debt. Duplication makes code maintenance tedious on a larger scale and longer run.

These flaws were often overlooked knowingly or unknowingly, for which SOLID principles served as the best cure.

So, you wondering now what SOLID principles hold and how does it solve the issues caused due to STUPID postulates. These are programming standards that all developers must understand very well, to create a product/system with good architecture.

SOLID principles can be considered as remedies to the problems caused due to any of the STUPID flaws in your codebase.

What do you mean by SOLID Design Principles in C#?

The **SOLID Design Principles in C#** are the design principles that basically used to manage most of the software design problems that generally we encountered in our day-to-day programming. These design principles are provided with some mechanism that will make the software designs more understandable, flexible, and maintainable.

What is the main reason behind most of the unsuccessful applications?

The following are the main reasons for most of the software failures.

1. Putting more functionalities on classes. (In the simple word a lot of functionalities we are putting into the class even though they are not related to the class.)

2. Implementing Tight coupling between the classes. If the classes are dependent on each other, then a change in one class will affect the other classes also.

How to overcome the unsuccessful application Development problem?

1. We need to use the correct architecture (i.e. MVC, Layered, 3-tier, MVP, and so on) as per the project requirements.
2. As a developer, we need to follow the Design Principles (i.e. SOLID Principles).
3. Again we need to choose the correct Design Patterns as per the project requirements.

Introduction to SOLID Design principles in C#

The SOLID Design principles are basically used to manage most of the software design problems that generally as a developer we face in our day-to-day programming. SOLID principles represent five design principles that basically used to make the software designs more understandable, flexible, and maintainable.

SOLID Acronym

1. **S** stands for the **Single Responsibility Principle** which is also known as SRP.
2. **O** stands for the **Open-Closed Principle** which is also known as OSP.
3. **L** stands for the **Liskov Substitution Principle** which is also known as LSP.
4. **I** stand for the **Interface Segregation Principle** which is also known as ISP.
5. **D** stands for **Dependency Inversion Principle** which is also known as DIP.

What are the Advantages of using SOLID Design Principles in C#?

We will get the following advantages of using SOLID Design Principles in C#.

1. Achieve the reduction in complexity of the code
2. Increase readability, extensibility, and maintenance
3. Reduce error and implement Reusability
4. Achieve Better testability
5. Reduce tight coupling

Single Responsibility

> Each class should should have RESPONSIBILITY over a single part of the functionality provided by the program.

What does this mean practically though? As a beginner programmer, this isn't very helpful. Let's expand on the concept.

Examples of single responsibilities :

- Validating inputs.
- Performing business logic.
- Saving and retrieving information to/from a database.
- Formatting a document.
- Performing calculations for the document.

So if you see a class that is validating inputs, logging events, reading and writing information to the database, and performing business logic, you have a class with A LOT of responsibilities; violating the Single Responsibility Principle.

How can you spot a class that may be violating the Single Responsibility Principle?

The class may have:

- Tight coupling
- Low cohesion
- No separation of concerns.

Tight coupling

Changing one class results in having to change a lot of other classes to get the program working again. Sound familiar?

Low Cohesion

The class contains fields and methods/functions that are unrelated to each other in any meaningful way.

A good way to spot this is if methods in a class don't reuse the same fields. Each method is using different fields from the class.

No Separation Of Concerns

Should my class that deals with validating an input be performing business logic and saving the data to the database? Not likely. Separate the program out into sections that deal with each concern.

A classic real-world example of something having too many responsibilities is the multi-function knives. They try to do too much and end up doing nothing well.

Implementation

Let's take a scenario of Garage service station functionality. It has 3 main functions; open gate, close gate, and performing service. The below example violates the SRP principle. The code below violates the SRP principle as it mixes open gate and close gate responsibilities with the main function of servicing of the vehicle.

```
public class GarageStation
{
 public void DoOpenGate()
 {
 //Open the gate functinality
 }

 public void PerformService(Vehicle vehicle)
 {
 //Check if garage is opened
 //finish the vehicle service
 }

 public void DoCloseGate()
 {
 //Close the gate functinality
 }
}
```

We can correctly apply SRP by refactoring of above code by introducing an interface. A new interface called IGarageUtility is created and gate-related methods are moved to a different class called GarageStationUtility.

```
public class GarageStation
{
 IGarageUtility _garageUtil;

 public GarageStation(IGarageUtility garageUtil)
 {
 this._garageUtil = garageUtil;
 }
 public void OpenForService()
 {
 _garageUtil.OpenGate();
 }
 public void DoService()
 {
 //Check if service station is opened and then
 //finish the vehicle service
 }
 public void CloseGarage()
 {
 _garageUtil.CloseGate();
 }
}
 public class GarageStationUtility : IGarageUtility
{
 public void OpenGate()
 {
 //Open the Garage for service
 }

 public void CloseGate()
 {
 //Close the Garage functionlity
 }
}

public interface IGarageUtility
{
 void OpenGate();
 void CloseGate();
}
```

Goal

This principle aims to separate behaviors so that if bugs arise as a result of your change, it won't affect other unrelated behaviors.

. . .

Open-Closed Principle (OCP)

> Software entities (classes, methods, modules) should be open for extension but closed for modification

What does this mean in a practical sense?

You should be able to change the behavior of a method without changing its source code.

For simple methods, adding/changing the logic in the method is perfectly reasonable. If you have to revisit this method 3+ times (not a hard number) due to requirements changing, you should start to think about the Open/Closed Principle.

Closing code to modification, why would you want to do this?

Code that we don't alter is less likely to create bugs due to unforeseen side effects.

This principle suggests that the class should be easily extended but there is no need to change its core implementations.

The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation.

i.e. New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

Implementation

Let's take an example of bank accounts like regular savings, salary savings, corporate, etc. for different customers. As for each customer type, there are different rules and different interest rates. The code below violates the OCP principle if the bank introduces a new Account type. Said code modifies this method for adding a new account type.

```
public class Account
{
 public decimal Interest { get; set; }
 public decimal Balance { get; set; }
 // members and function declaration
 public decimal CalcInterest(string accType)
 { if (accType == "Regular") // savings
 {
```

```
 Interest = (Balance * 4) / 100;
 if (Balance < 1000) Interest -= (Balance * 2) / 100;
 if (Balance < 50000) Interest += (Balance * 4) / 100;
 }
 else if (accType == "Salary") // salary savings
 {
 Interest = (Balance * 5) / 100;
 }
 else if (accType == "Corporate") // Corporate
 {
 Interest = (Balance * 3) / 100;
 }
 return Interest;
 }
}
```

We can apply OCP by using the interface, abstract class, abstract methods, and virtual methods when you want to extend functionality. Here I have used interface for example only but you can go as per your requirement.

```
interface IAccount
{
 // members and function declaration, properties
 decimal Balance { get; set; }
 decimal CalcInterest();
}

//regular savings account
public class RegularSavingAccount : IAccount
{
 public decimal Balance { get; set; } = 0;
 public decimal CalcInterest()
 {
 decimal Interest = (Balance * 4) / 100;
 if (Balance < 1000) Interest -= (Balance * 2) / 100;
 if (Balance < 50000) Interest += (Balance * 4) / 100;

 return Interest;
 }
}

//Salary savings account
public class SalarySavingAccount : IAccount
{
 public decimal Balance { get; set; } = 0;
 public decimal CalcInterest()
 {
 decimal Interest = (Balance * 5) / 100;
 return Interest;
 }
}
```

```
//Corporate Account
public class CorporateAccount : IAccount
{
 public decimal Balance { get; set; } = 0;
 public decimal CalcInterest()
 {
 decimal Interest = (Balance * 3) / 100;
 return Interest;
 }
}
```

In the above code three new classes are created; regular saving account, SalarySavingAccount, and CorporateAccount, by extending them from IAccount.

This solves the problem of modification of class and by extending the interface, we can extend functionality.

The above code is implementing both OCP and SRP principles, as each class has single is doing a single task and we are not modifying the class and only doing an extension.

Goal

This principle aims to extend a Class's behavior without changing the existing behavior of that Class. This is to avoid causing bugs wherever the Class is being used.

. . .

Liskov Substitution

> If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

LSP states that the child class should be perfectly substitutable for their parent class. If class C is derived from P then C should be substitutable for P.

We can check using LSP whether inheritance is applied correctly or not in our code.

LSP is a fundamental principle of SOLID Principles and states that if a program or module is using base class then derived class should be able to extend their base class without changing their original implementation.

Implementation

Let's consider the code below where LSP is violated. We cannot simply substitute a Triangle, which results in the printing shape of a triangle, with a Circle.

```
namespace Demo
{
 public class Program
 {
 static void Main(string[] args)
 {
 Triangle triangle = new Circle();
 Console.WriteLine(triangle.GetColor());
 }
 }

 public class Triangle
 {
 public virtual string GetShape()
 {
 return "Triangle";
 }
 }

 public class Circle: Triangle
 {
 public override string GetShape()
 {
 return "Circle";
 }
 }
}
```

To correct the above implementation, we need to refactor this code by introducing an interface with a method called GetShape.

```
namespace Demo
{
 class Program
 {
 static void Main(string[] args)
 {
 Shape shape = new Circle();
 Console.WriteLine(shape.GetShape());
 shape = new Triangle ();
 Console.WriteLine(shape.GetShape());
 }
 }
```

```
  public abstract class Shape
  {
  public abstract string GetShape();
  }

  public class Triangle: Shape
  {
  public override string GetShape()
  {
  return "Triangle";
  }
  }

  public class Circle: Triangle
  {
  public override string GetShape()
  {
  return "Circle";
  }
  }
 }
```

Goal

This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

. . .

Interface Segregation

> Clients should not be forced to depend on methods that they do not use.

The **interface segregation principle** (ISP) requires that classes only be able to perform behaviors that are useful to achieve its end functionality. In other words, classes do not include behaviors they do not use.

This relates to our first SOLID principle in that together these two principles strip a class of all variables, methods, or behaviors that do not directly contribute to their role. Methods must contribute to the end goal in their entirety.

The advantage of ISP is that it splits large methods into smaller, more specific methods. This makes the program easier to debug for three reasons:

1. There is less code carried between classes. Less code means fewer bugs.

2. A single method is responsible for a smaller variety of behaviors. If there is a problem with a behavior, you only need to look over the smaller methods.

3. If a general method with multiple behaviors is passed to a class that doesn't support all behaviors (such as calling for a property that the class doesn't have), there will be a bug if the class tries to use the unsupported behavior.

Implementation

In the below code, ISP is broken as the process method is not required by OfflineOrder class but is forced to implement.

```
public interface IOrder
 {
 void AddToCart();
 void CCProcess();
 }

 public class OnlineOrder : IOrder
 {
 public void AddToCart()
 {
 //Do Add to Cart
 }

 public void CCProcess()
 {
 //process through credit card
 }
 }

 public class OfflineOrder : IOrder
 {
 public void AddToCart()
 {
 //Do Add to Cart
 }

 public void CCProcess()
 {
 //Not required for Cash/ offline Order
 throw new NotImplementedException();
 }
 }
```

We can resolve this violation by dividing IOrder Interface.

```
public interface IOrder
 {
 void AddToCart();
 }

 public interface IOnlineOrder
 {
 void CCProcess();
 }

 public class OnlineOrder : IOrder, IOnlineOrder
 {
 public void AddToCart()
 {
 //Do Add to Cart
 }

 public void CCProcess()
 {
 //process through credit card
 }
 }

 public class OfflineOrder : IOrder
 {
 public void AddToCart()
 {
 //Do Add to Cart
 }
 }
```

Goal

This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.

. . .

Dependency Inversion

> - High-level modules should not depend on low-level modules. Both should depend on the abstraction.
>
> - Abstractions should not depend on details. Details should depend on abstractions.

The **dependency inversion principle** (DIP) has two parts:

1. High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)
2. Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions.

The first part of this principle **reverses traditional OOP software design**. Without DIP, programmers often construct programs to have high-level (less detail, more abstract) components explicitly connected with low-level (specific) components to complete tasks.

DIP decouples high and low-level components and instead connects both to abstractions. High and low-level components can still benefit from each other, but a change in one should not directly break the other.

The advantage of this part of DIP is that decoupled programs require less work to change. Webs of dependencies across your program mean that a single change can affect many separate parts.

> If you minimize dependencies, changes will be more localized and require less work to find all affected components.

The second part can be thought of as "the abstraction is not affected if the details are changed". The abstraction is the user-facing part of the program.

The details are the specific behind-the-scenes implementations that cause program behavior visible to the user. In a DIP program, we could fully overhaul the behind-the-scenes implementation of how the program achieves its behavior without the user's knowledge.

> This process is known as **refactoring**.

This means you won't have to hard-code the interface to work solely with the current details (implementation). This keeps our code loosely coupled and allows us the flexibility to refactor our implementations later.

Implementation

In below code, we have implemented DIP using IoC using injection constructor. There are different ways to implement Dependency injection. Here, I have use injection thru constructor but you inject the dependency into class's constructor (Constructor

Injection), set property (Setter Injection), method (Method Injection), events, index properties, fields and basically any members of the class which are public.

```
public interface IAutomobile
{
 void Ignition();
 void Stop();
}public class Jeep : IAutomobile
{
 #region IAutomobile Members
 public void Ignition()
 {
 Console.WriteLine("Jeep start");
 }

 public void Stop()
 {
 Console.WriteLine("Jeep stopped.");
 }
 #endregion
}

public class SUV : IAutomobile
{
 #region IAutomobile Members
 public void Ignition()
 {
 Console.WriteLine("SUV start");
 }

 public void Stop()
 {
 Console.WriteLine("SUV stopped.");
 }
 #endregion
}
public class AutomobileController
{
 IAutomobile m_Automobile;

 public AutomobileController(IAutomobile automobile)
 {
 this.m_Automobile = automobile;
 }

 public void Ignition()
 {
 m_Automobile.Ignition();
 }

 public void Stop()
 {
 m_Automobile.Stop();
```

```
 }
}

class Program
{
 static void Main(string[] args)
 {
 IAutomobile automobile = new Jeep();
 //IAutomobile automobile = new SUV();
 AutomobileController automobileController = new
AutomobileController(automobile);
 automobile.Ignition();
 automobile.Stop();

 Console.Read();
 }
}
```

In the above code, IAutomobile interface is in an abstraction layer and AutomobileController as the higher-level module. Here, we have integrated all in a single code but in real-world, each abstraction layer is a separate class with additional functionality. Here products are completely decoupled from the consumer using IAutomobile interface. The object is injected into the constructor of the AutomobileController class in reference to the interface IAutomobile. The constructor where the object gets injected is called injection constructor.

DI is a software design pattern that allows us to develop loosely coupled code. Using DI, we can reduce tight coupling between software components. DI also allows us to better accomplish future changes and other difficulties in our software. The purpose of DI is to make code sustainable.

Goal

This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.

. . .

Conclusion

In this article, we've taken a **deep dive into the SOLID principles of object-oriented design.**

We **started with a quick bit of SOLID history and the reasons these principles exist.**

Letter by letter, we've **broken down the meaning of each principle with a quick code example that violates it. We then saw how to fix our code** and make it adhere to the SOLID principles.

Join FAUN: **Website** |**Podcast** |**Twitter** |**Facebook** |**Instagram** |**Facebook Group** |**Linkedin Group** | **Slack** |**Cloud Native News** |**More**.

If this post was helpful, please click the clap button below a few times to show your support for the author

Coding | Programming | Programming Languages | Development | Developer