Solid Principles C#

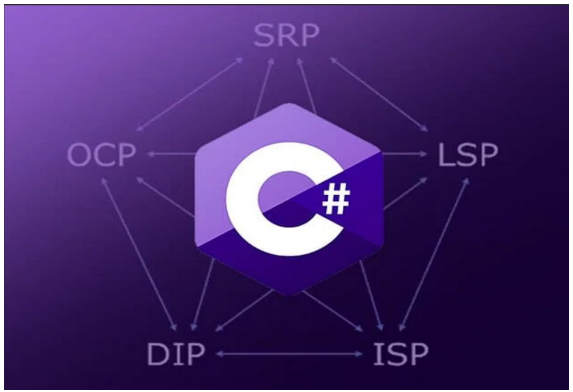
Mark Shinozaki, Darrel Nitereka, Yuuki Matsunari

What are Solid Principles?

- SOLID Principles which were introduced by Robert C.Martin also referred to as Uncle Bob, in his paper in the year 2000 *Design Principles and Design Patterns*.

- SOLID Principles are a set of five design principles in object-oriented programming that help developers create software that is easy to maintain and adapt to changing requirements, fostering scalability and flexibility in the development process.

- The SOLID Principles consist of:
  - The Single-Responsibility Principle
  - The Open-Closed Principle
  - The Liskov Substitution Principle
  - The Interface Segregation Principle
  - The Dependency Inversion Principle



Why are Solid Principles Important?

- **Enhanced Maintainability:** SOLID principles lead to a codebase that is easier to maintain, update, and scale.
- **Reduced Complexity:** They help break down complex processes into simpler, more manageable segments of code.
- **Increased Flexibility:** Adhering to SOLID principles allows systems to evolve with minimal impact on existing functionality.
- **Improved Testability:** Code that follows SOLID is generally easier to test due to reduced interdependencies.
- **Higher Quality Code**: SOLID principles promote the creation of high-quality, robust software that can handle changes and growth efficiently.
- **Facilitates Team Collaboration:** Clear, well-defined code structure aids in understanding among team members, making collaborative work more efficient.

Overview of Presentation

In this presentation we will be going over each of the five SOLID principles that make up the object-oriented principles for good design in software development. We will demonstrate this through:

- Visual examples
- Coding examples
- Real World examples
- Q/A’s

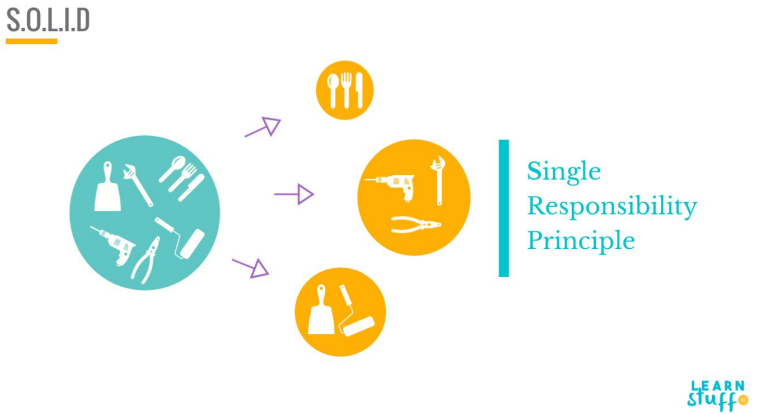
Single-Responsibility Principle

Demo Topic

- Darrel

What is Single-Responsibility Principle?

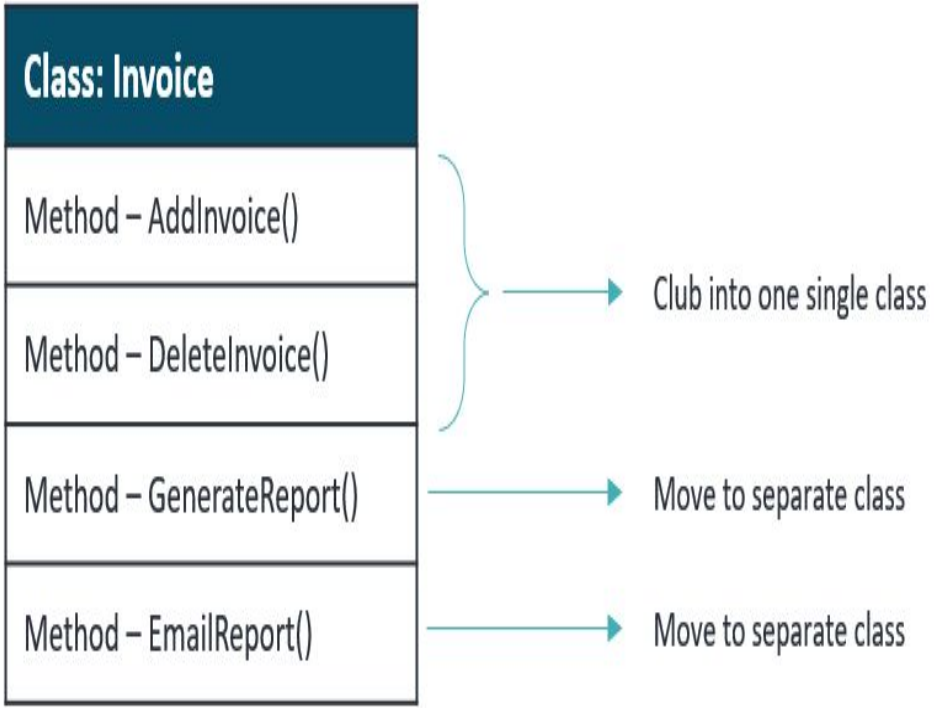
- Single Responsibility Principle(SRP) is a software design that states that “A class should only have one and only one reason to change”, meaning it should have only one job or responsibility.
- This principle aims to reduce complexity by dividing the software into small, manageable parts.
- With SRP, classes are clearer and modifications simpler, as changes affect fewer parts of the system.
- This principle enhances code cohesion and streamlines both development and debugging.
- It minimizes risk when fixing bugs, as changes are contained, and improves testability due to the targeted nature of each class's functionality.



The Single-Responsibility Principle is represented by this “Invoice” class visual:

- AddInvoice() and DeleteInvoice(): These methods relate directly to invoice management — the core responsibility of the "Invoice" class. The visual suggests these methods should be "clubbed into one single class," indicating they are appropriately placed as they serve the single responsibility of managing invoices.

- GenerateReport() and EmailReport(): These methods extend beyond the core function of invoice management. The visual indicates they should be "moved to separate classes," aligning with SRP to ensure the "Invoice" class doesn't handle multiple, unrelated responsibilities like report generation and email communications.



Before applying SRP

```
namespace ConsoleApp1;

public class InvoiceProcessor
{
    public void AddInvoice()
    {
        // Logic to add invoice
    }

    public void DeleteInvoice()
    {
        // Logic to delete invoice
    }

    public void GenerateInvoiceReport()
    {
        // Logic to generate an invoice report
    }

    public void EmailInvoiceReport()
    {
        // Logic to email the invoice report
    }
}
```

- **Conflated Responsibilities:** The InvoiceProcessor class handles multiple responsibilities, managing invoices, generating reports, and sending emails. This goes against SRP as the class has more than one reason to change.
- **Complexity:** A single class with multiple responsibilities can become overly complex and hard to understand, making maintenance and updates more difficult.
- **Tight Coupling:** The methods for different tasks are tightly coupled within one class, which can lead to a ripple effect of bugs and changes impacting unrelated functionalities.
- **Testing Challenges:** Writing unit tests for such a class is challenging because tests for one method may inadvertently affect others.
- **Obstacles to Scalability:** Scaling or modifying one aspect of the system (such as changing the email service) would require changes to the class that also affect invoice management.

After applying SRP

```
namespace ConsoleApp1;

1 usage
public class Invoice
{
    // Invoice properties like Id, Amount, Date, etc.

    public void Add()
    {
        // Logic to add invoice
    }

    public void Delete()
    {
        // Logic to delete invoice
    }
}
```

```
namespace ConsoleApp1;

public class InvoiceReportGenerator
{
    public string Generate(Invoice invoice)
    {
        // Logic to generate an invoice report
        return "Report content";
    }

}
```

```
namespace ConsoleApp1;

public class EmailService
{
    public void SendEmail(string emailAddress, string content)
    {
        // Logic to send email
    }
}
```

- **Focused Classes:** Each class has a clear, singular focus. Invoice for managing invoices, InvoiceReportGenerator for report generation, and EmailService for email communication.
- **Reduced Complexity:** With SRP, each class is simpler and only encapsulates logic for its single responsibility, making the code easier to understand and maintain.
- **Loose Coupling:** Classes are loosely coupled, meaning changes in one class have minimal impact on others, which enhances the modularity of the code.
- **Easier Testing:** Each class can be tested independently, leading to more straightforward and reliable unit tests.
- **Improved Scalability:** Modifications or expansions within one domain (such as adding new features to report generation) can be done without affecting the invoice management or email sending functionalities.
- **Adherence to SRP:** The code structure now properly follows the Single Responsibility Principle, which is one of the fundamental principles of object-oriented design for creating robust and maintainable software.

Open-Closed Principle

Demo Topic

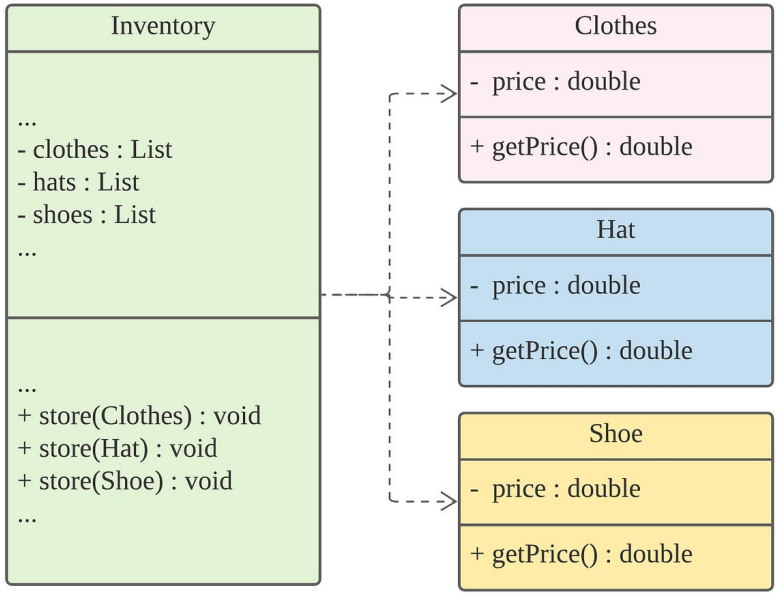
- Darrel

What is Open-Closed Principle?

- Open-Closed Principle(OCP) states that “a software module should be open for extension but closed to modification.”
- OCP dictates that code should easily accommodate new features without altering established logic.
- Open for Extension means it enables the addition of new behaviors (like through new subclasses) without altering existing classes.
- This approach helps maintain code integrity while ensuring adaptability for future developments, reducing the risk of regressions or bugs.

Example of Open-Closed Principle

- **Inventory Class Role:** Serves as a central hub for storing various item types like clothes, hats, and shoes.
- **Storage Methods:** Includes store() methods for each product type, indicating versatility in handling diverse items.
- **Product Class Design:** Each product class (Clothes, Hat, Shoe) contains a price and getPrice() method, suggesting a standardized approach across product types.
- **OCP Compliance:** Inventory management is robust to expansion; new item types can be added without altering the existing Inventory structure, aligning with OCP principles.



Code Example:

Abstract Base Class: By defining an abstract Product class with an abstract method GetPrice(), we establish a contract. Any subclass extending Product must provide its own implementation of GetPrice() without changing the method's signature.

Subclass Extension: New product types (Clothes, Hat, Shoe) extend the Product class without modifying it. Each subclass provides its own implementation of GetPrice(), adhering to the OCP's "open for extension" part.

```
public abstract class Product
{
        protected double price;

        public abstract double
                GetPrice();
}
```

```
public class Hat : Product
{
    public Hat(double price)
    {
        this.price = price;
    }

    public override double
GetPrice()
    {
        return price;
    }
}
```

```
public class Clothes : Product
{
    public Clothes(double price)
    {
        this.price = price;
    }

    public override double
GetPrice()
    {
        return price;
    }
}
```

```
public class Shoe : Product
{
    public Shoe(double price)
    {
        this.price = price;
    }

    public override double
GetPrice()
    {
        return price;
    }
}
```

Continued:

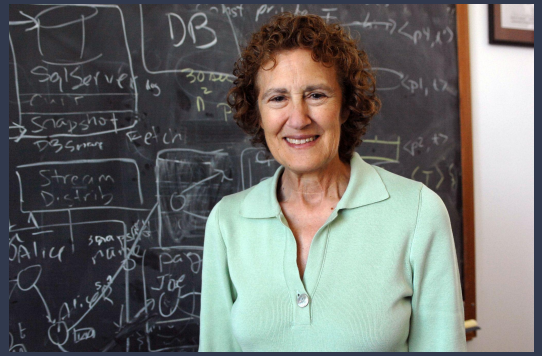
- **Closed for Modification:** The Inventory class processes items of type Product. It is designed to store any Product subclass without needing to know the specific details of each product. When new product types are added to the system, there is no need to change the Inventory class's code, fulfilling the "closed for modification" part of the principle.
- **Polymorphism:** The Inventory class's method StoreProduct(Product product) takes a Product type, enabling it to store any kind of product. This use of polymorphism means the Inventory class remains unchanged when new types of products are introduced.
- **Ease of Expansion:** If you wanted to add a new product category (like Accessory), you would simply create a new subclass of Product. The Inventory class would not require any changes to accommodate this new subclass, therefore leaving the system easy to expand.

```
public class Inventory
{
    private List<Product>
products = new
List<Product>();

    public void
StoreProduct(Product product)
    {
        products.Add(product);
    }

    public double
CalculateTotalInventoryValue()
    {
        double total = 0;
        foreach (var product in
products)
        {
            total +=
product.GetPrice();
        }
        return total;
    }
}
```

```
public class Accessory :
Product
{
    public Accessory(double
price)
        {
        this.price = price;
}

    public override double
GetPrice()
{
        return price;
 }
}
```

Liskov Substitution Principle

- Mark



Professor Barbara Liskov, Computer Scientist, MIT (Coveney)

- The Liskov Substitution Principle, introduced by Barbara Liskov in a 1987 paper titled, “Data Abstraction and Hierarchy,” is one of the five SOLID principles.

- LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program

- This principle ensures that a derived class can be substituted for its base class without altering the correctness of the program.

(deVilla)

What does, “objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program” mean exactly?

- ‘Shape’ is the base class with a virtual method ‘Draw’.
- ‘Circle’ and ‘Square’ are derived classes that override the ‘Draw’ method
- In the ‘Main’ method, we create objects of ‘Circle’ and ‘Square’ and assign them to variables of type ‘Shape’.
- When calling ‘Draw’ on these objects, the overridden method in the respective derived classes is invoked, demonstrating that we can use objects of derived classes wherever a base class object is expected.
- This example adheres to LSP, as substituting objects of the derived classes ‘Circle’ and ‘Square’ for objects of the base class ‘Shape’ does not affect the correctness of the program.

```
using System;

// Base class
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}

// Derived class 1
class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

// Derived class 2
class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a square");
    }
}

class Program
{
    static void Main()
    {
        // Using objects of the derived classes where a base class object is expected

        Shape shape1 = new Circle();
        shape1.Draw();  // Output: Drawing a circle

        Shape shape2 = new Square();
        shape2.Draw();  // Output: Drawing a square

        // This is an example of Liskov Substitution - using derived classes interchangeably with the base class.
    }
}
```

Good vs Bad Example of Liskov Substitution Principle

Importance of LSP

In terms of this example, by organizing the class hierarchy in this way, it ensures that any class within the ‘FlyingBirds’ hierarchy like ‘Duck’ can be used interchangeably with its base class ‘Bird’. Meanwhile, a class like ‘Ostrich’ can also be a bird but is not forced to inherit behaviors it doesn't exhibit (like flying).

In General terms, LSP is crucial in OOP design, it ensures that derived classes can be seamlessly substituted for their base classes, promoting flexibility, extensibility, and robustness without introducing unexpected behaviors.

(NotMyselfNotMyself 29.3k1717 gold badges5656 silver badges7474 bronze badges)

Bad example

```
public class Bird
{
    public void Fly()
    {
        Console.WriteLine("Flying");
    }
}

public class Duck : Bird { }

public class Ostrich : Bird { }
```

- ‘Duck’ & ‘Ostrich’ inherit from ‘Bird’, but an ‘Ostrich’ cannot fly.
- This violates the LSP because the derived class ‘Ostrich’ cannot be substituted for its base class ‘Bird’ without affecting the correctness of the program.

Good example

```
public class Bird { }

public class FlyingBirds : Bird
{
    public void Fly()
    {
        Console.WriteLine("Flying");
    }
}

public class Duck : FlyingBirds { }

public class Ostrich : Bird { }
```

- ‘FlyingBirds’ class is introduced to represent birds that can fly. Both ‘Duck’ and ‘Ostrich’ inherit from ‘Bird’, but only ‘Duck’ inherits from ‘FlyingBirds’ to indicate that it can fly.
- This adheres to LSP because now, only birds that can fly are part of ‘FlyingBirds’ hierarchy’

Interface Segregation Principle

- Mark

(deVilla)

What the is Interface Segregation Principle?

- **“Clients should not be forced to depend on methods they don’t use.”**
- **Code is more maintainable, reducing the coupling between classes and ensuring that interfaces are tailored to specific requirements**
- **Similar to the Single Responsibility Principle, the goal of the interface segregation principle is to reduce the side effects and frequency of required changes by splitting the software into multiple parts.**
- **You prevent bloated interfaces that define methods for multiple responsibilities.**

ISP Breakdown

```
Public interface IWorker
{
        void Work();
        void Eat();
        void Sleep();
}
```

```
Public class HumanWorker: IWorker
{
        void Work();
        void Eat();
        void Sleep();
}

Public class RobotWorker: IWorker
{
        void Work();
        void Eat();
        void Sleep();
}
```

```
Public interface IWorkable
{
        void Work();
}
Public interface IEatable
{
        void Eat();
}
Public interface ISleepable
{
        void Sleep();
}
```

```
Public class HumanWorker : IWorkable,
IEatable, ISleepable
{
        public void Work();
        public void Eat();
        Public void Sleep();

}

Public class RobotWorker : IWorkable
{
        Public void Work()
}
```

In this example, the IWorker interface violates the ISP, as it forces the RobotWorker class to implement the Eat/Sleep methods, which are irrelevant for robots. To adhere to the ISP, we can break down the IWorker interface into smaller interfaces.

(ByteHide)

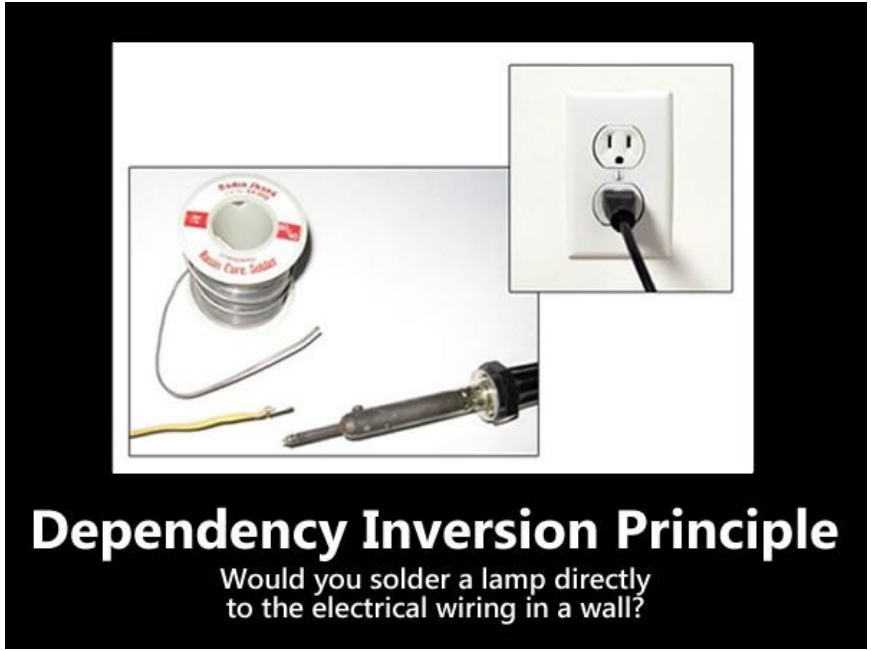
Importance of ISP

Interface Segregation Principle is a critical concept in OOP and promotes maintainable code. The main goal of ISP is to ensure that clients are not forced to implement interfaces they do not use. By breaking down interfaces into smaller, more specific interfaces, clients can implement only the methods that relevant to their needs, reducing coupling and promoting a more flexible architecture.

Dependency Inversion Principle

- Yuuki

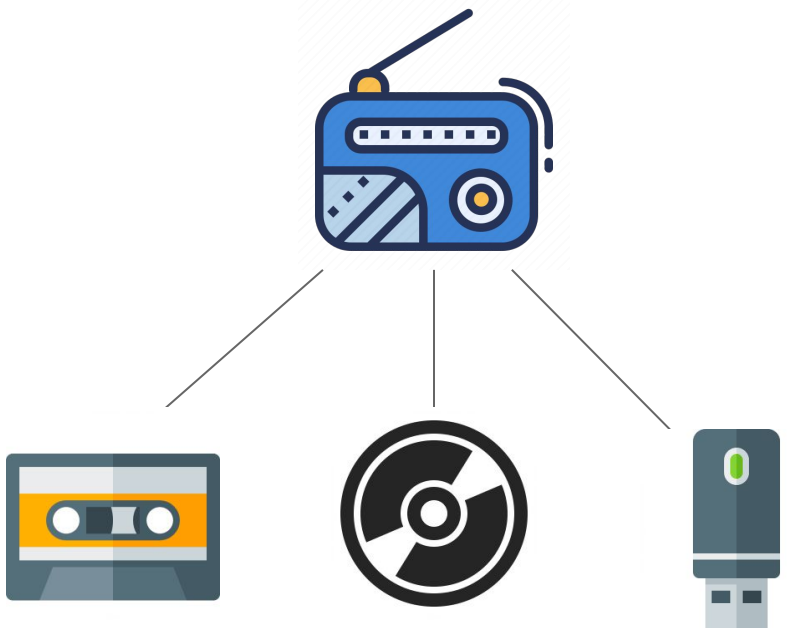
What is Dependency Inversion Principle?



- Dependency Inversion Principle is the principle:
  - **High-level modules should not depend on low-level modules**
    - BUT, both should depend on shared abstractions.
  - **Abstractions should not depend on details**
    - Instead, details should depend on abstractions.

For example...

- Assume we make a “radio recorder function” in C#



- We want to make functions
  - Recording on a tape
  - Recording on a CD
  - Recording on a USB

We made the code like..

C# RadioRecorder.cs

```
class RadioRecorder
{
  public void Record()
    {
        if (ConnectsTape())
        {
            Tape tape = new Tape();
            tape.Record();
            return;
        }

        if (ConnectsCD())
        {
            CD cd = new CD();
            cd.Record();
            return;
        }

        USB usb = new USB();
        us.Record();
    }
}
```

C# Tape.cs

```
public class Tape
{
    public void Record()
    {
        // Recordig on Tape
    }
}
```

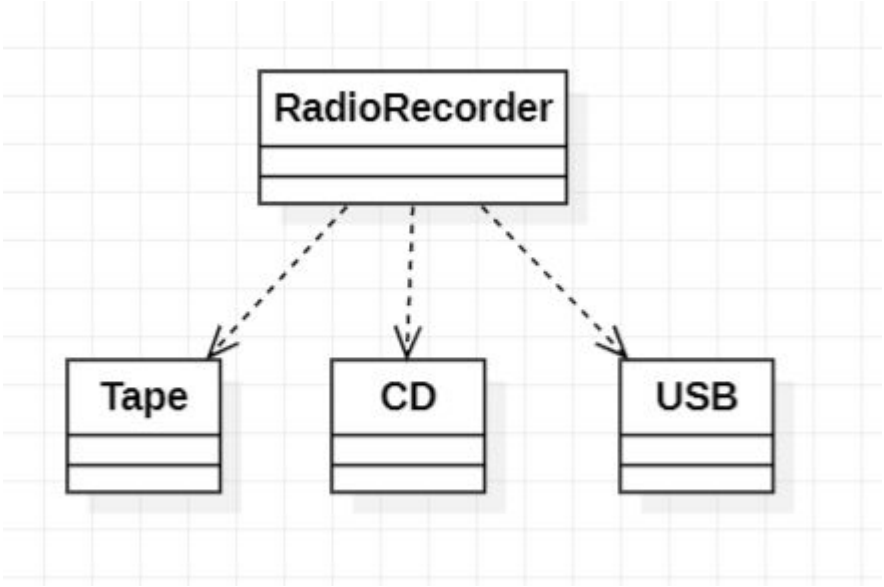
C# CD.cs

```
public class CD
{
    public void Record()
    {
        // Recordig on CD
    }
}
```

C# USB.cs

```
public class USB
{
    public void Record()
    {
        // Recordig on USB
    }
}
```

If we explain the code in UML...



- The code has a problem!
- Why?
  - *A high-level module*(RadioRecorder) is depends on *lower-level modules*(Tape, CD, USB).

C# RadioRecorder.cs

```
class RadioRecorder
{
  public void Record()
    {
        if (ConnectsTape())
        {
            Tape tape = new Tape();
            tape.Record();
            return;
        }

        if (ConnectsCD())
        {
            CD cd = new CD();
            cd.Record();
            return;
        }

        USB usb = new USB();
        us.Record();
    }
}
```

Make a command by checking “what will we record” (concrete).

↓

It means depending on low-level modules!

C# Tape.cs

```
public class Tape
{
    public void Record()
    {
        // Recordig on Tape
    }
}
```

C# CD.cs

```
public class CD
{
    public void Record()
    {
        // Recordig on CD
    }
}
```

C# USB.cs

```
public class USB
{
    public void Record()
    {
        // Recordig on USB
    }
}
```

The solution with Dependency Inversion Principle

- **To solve this...**
  - We have to change the code to satisfy “details should depend on abstractions”
- **What should we do?**
  - One solution: Use an Interface and factory class to make an abstraction

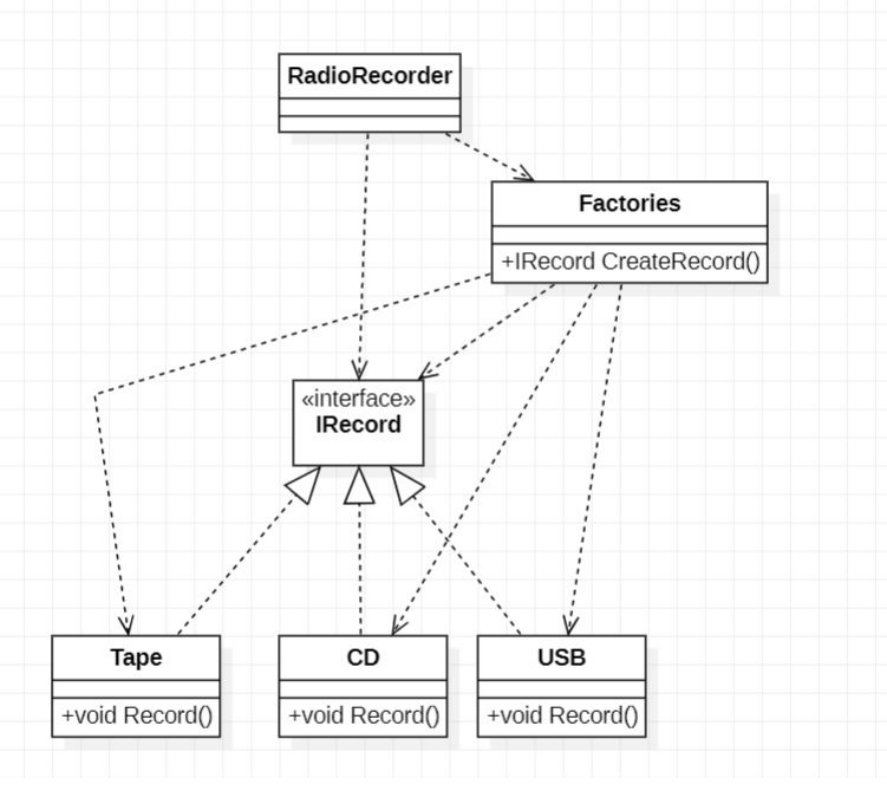
C# IRecord.cs

```
public interface IRecord
{
    enum Recorder
    {
        Tape,
        CD,
        USB
    }

    public void Record();
}
```

C# Factories.cs

```
public class Factories
{
    public IRecord CreateRecord()
    {
        // Get what recorder is connected
        var recorder = GetRecorder();

        if (recorder == IRecord.Recorder.Tape)
        {
            return new Tape();
        }

        if (recorder == IRecord.Recorder.CD)
        {
            return new CD();
        }

        return new USB();
    }
}
```

In UML...



- Now we have an interface as an abstraction
  - The high-module (RadioRecorder) no longer depends on low modules!

Merits of Dependency Inversion Principle

- **Reduced Coupling**
  - DIP promotes loose coupling between modules in a software system. High-level modules do not directly depend on the implementation details of low-level modules. This reduces the interdependencies between components, making the system less brittle and easier to modify.
  -
- **Ease of Maintenance**
  - When high-level modules depend on abstractions rather than concrete implementations, changes to the low-level modules have minimal impact on the high-level modules. This isolation makes maintenance and updates more straightforward and less prone to introducing unintended side effects.

- **Flexibility and Extensibility**
  - DIP allows you to introduce new implementations of low-level modules without affecting the high-level modules. This flexibility is particularly useful when you need to adapt the software to new requirements or integrate with external systems.

Conclusion

- ➔ Solid principles are important for promoting maintainability, scalability, and flexibility in software development.
- ➔ …. makes code cleaner, more modular and scalable
- ➔ …. Promote effective collaboration within development teams by providing a common set of guidelines and best practices
- ➔ …. Reduce dependencies so that engineers change one area of software without impacting others
- ➔ Specific concepts help team members to have a substantive conversation on issues throughout the development process

DEMO

Single Responsibility Principle & Open-Closed Principle

Anyone like Soccer?

Follow This Link:

https://github.com/MarkShinozaki/321SolidPrinciples

REVIEW SOCCER.CS

Questions ?

Thank You !!

Citations

Slide 6 (picture) - Ungureanu, Vlad. “Single Responsibility Principle.” *Medium*, Medium, 12 Nov. 2020, medium.com/@learnstuff.io/single-responsibility-principle-ad3ae3e264bb.

Slide 7-9 - “Single Responsibility in Solid Design Principle.” *GeeksforGeeks*, GeeksforGeeks, 31 Oct. 2023, www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/.

Slide 11 (picture) - DotNetCurry.com. “Open-Closed Principle (Software Gardening: Seeds).” *DNC Magazine*, www.dotnetcurry.com/software-gardening/1176/solid-open-closed-principle. Accessed 25 Nov. 2023.

Slide 12 - Bang, Jun. “What Is Open-Closed Principle?” *Medium*, Medium, 6 Nov. 2020, tan-jun-bang.medium.com/what-is-open-closed-principle-15b7b85dadf7.

Slide 13 (picture) - Coveney, Donna. *Q&A with Institute Professor Barbara Liskov*. 2009. *News.Mit.Edu*, Massachusetts , https://news.mit.edu/2009/turing-liskov-qaa-0310.

Slide 14, 20, 24 (picture) - deVilla, Joey. *Liskov Substitution principles*. *Global Nerdy*, Https://Www.Globalnerdy.Com/2009/07/15/the-Solid-Principles-Explained-with-Motivational-Posters/, July 15, 2009.

Slide 15 (code example) - NotMyselfNotMyself 29.3k1717 gold badges5656 silver badges7474 bronze badges. “What Is an Example of the Liskov Substitution Principle?” *Stack Overflow*, 1 Nov. 1954, stackoverflow.com/questions/56860/what-is-an-example-of-the-liskov-substitution-principle.

Slide 21 Code example - ByteHide. “▹learn Interface Segregation Principle in C# (+ Examples).” *ByteHide Blog*, 17 Aug. 2023, www.bytehide.com/blog/interface-segregation-principle-in-csharp-solid-principles.

Slide 25-31 (code example) - @k2491p, https://qiita.com/k2491p/items/686ee5dd72b4baf9a81a

Slide 32- Malik, S. K. (2023, November 12). *Dependency inversion principle in software design*. LinkedIn. https://www.linkedin.com/pulse/dependency-inversion-principle-software-design-sanjoy-kumar-malik