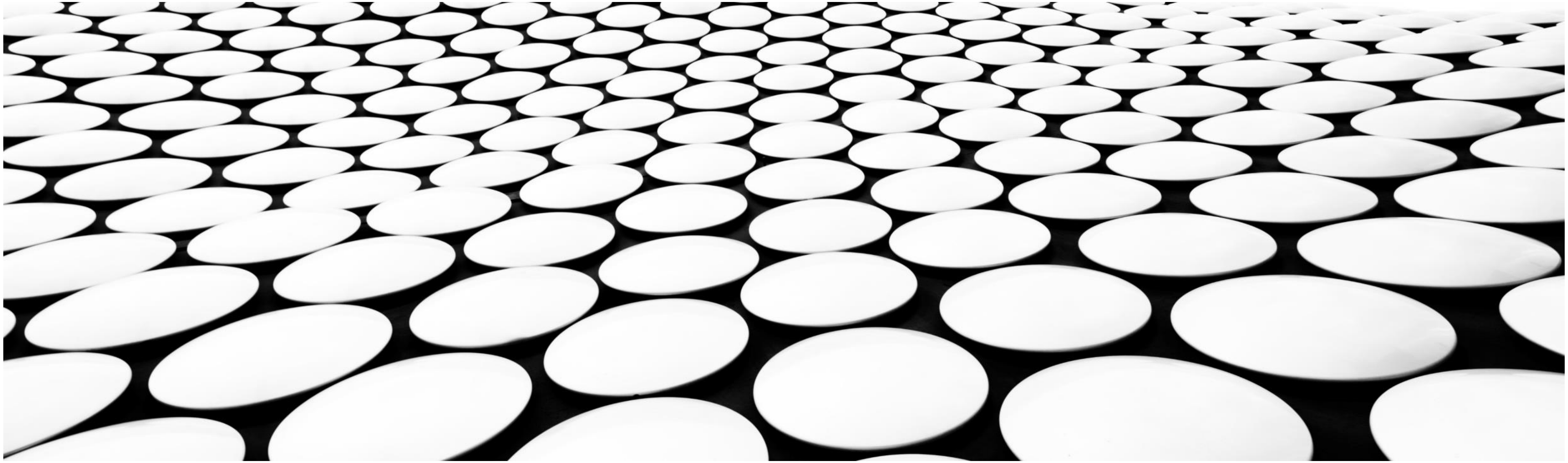
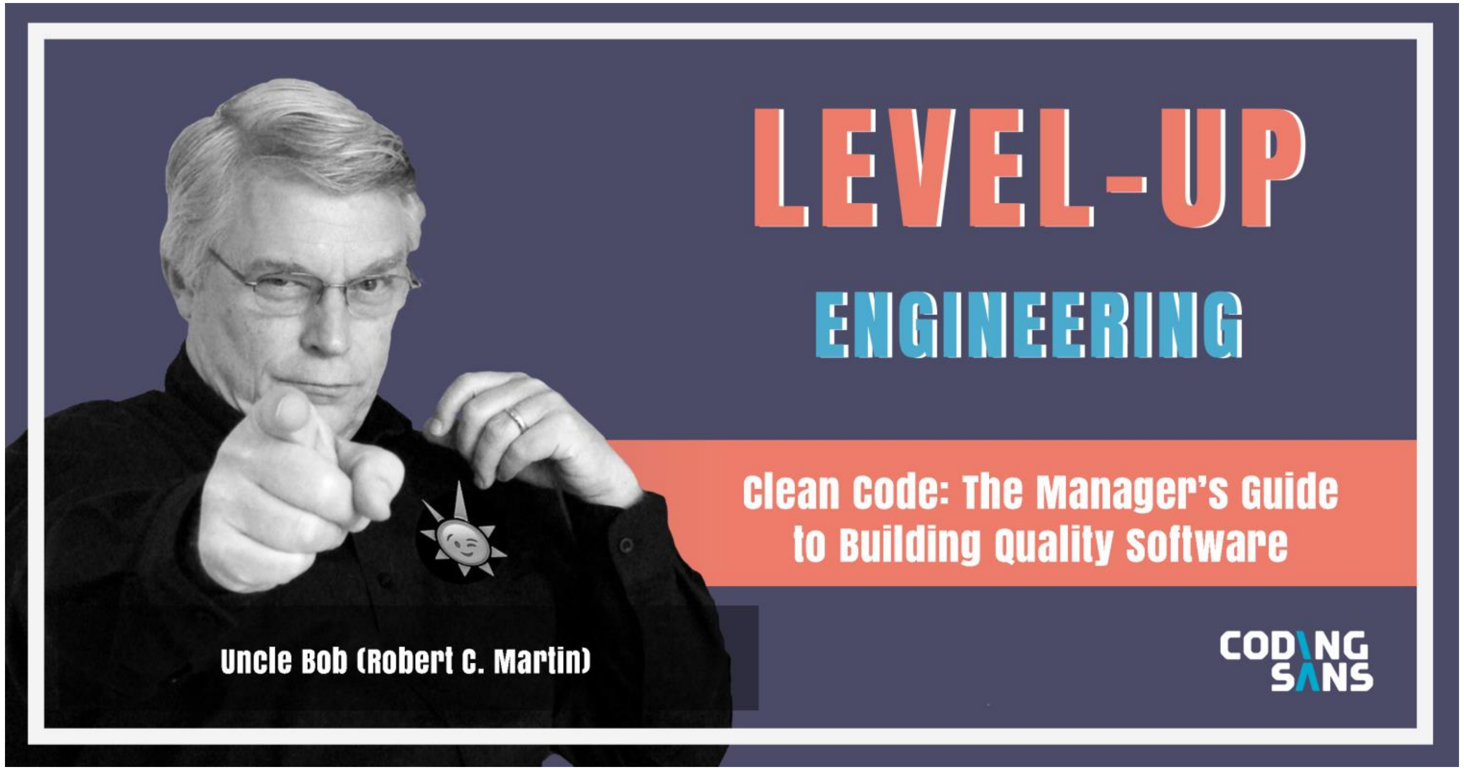
SOLID PRINCIPLES

OOP DESIGN



SOLID PRINCIPLES

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle


LEVEL-UP
ENGINEERING
Clean Code: The Manager's Guide to Building Quality Software
Uncle Bob (Robert C. Martin)
CODING SANS

https://cleancoders.com/

- ✓ Anlaşılır
- ✓ Tekrar Kullanılabilir
- ✓ Esnek

SINGLE RESPONSIBILITY PRINCIPLE

your code should have only one job to do

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

SINGLE RESPONSIBILITY PRINCIPLE

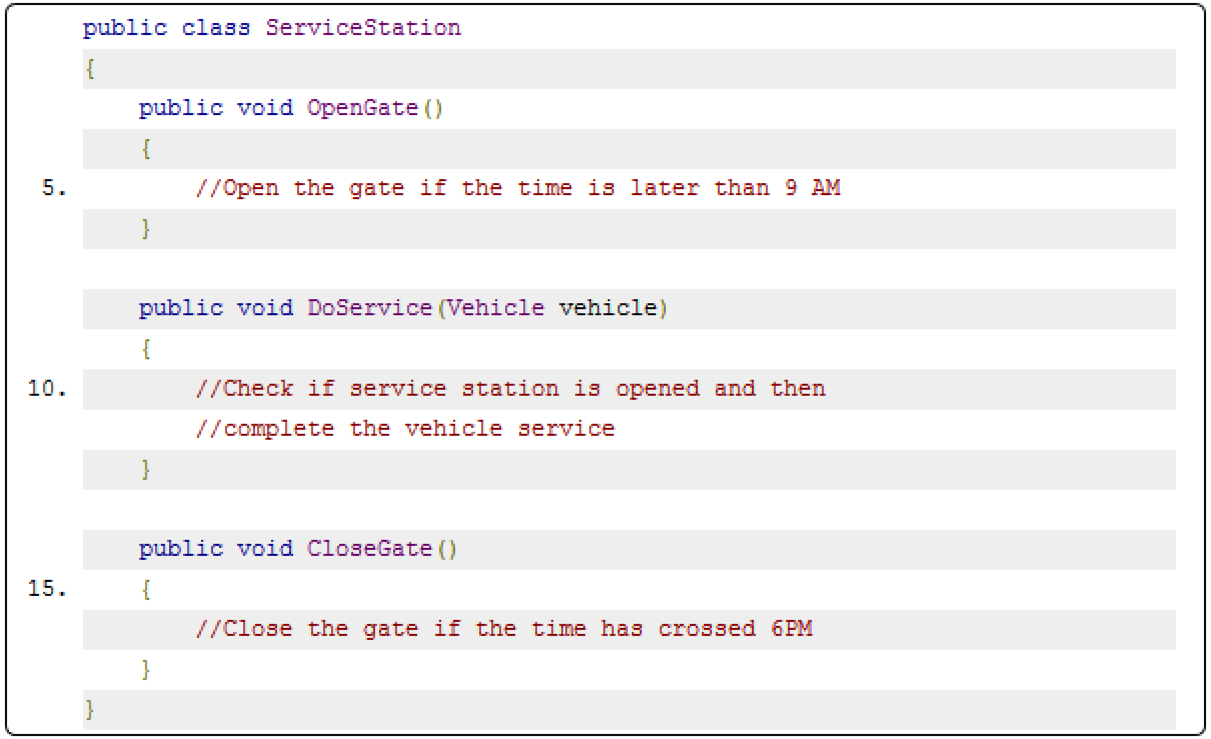
your code should have
only one job to do

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

SINGLE RESPONSIBILITY PRINCIPLE

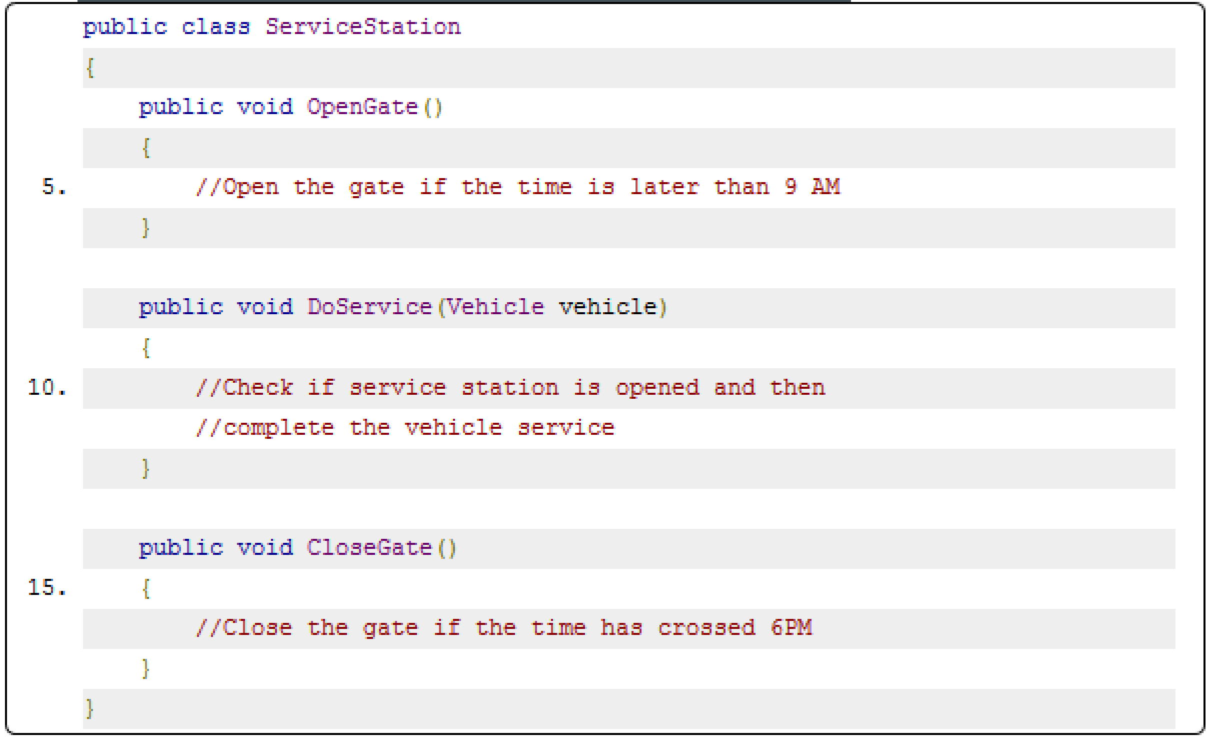
your code should have only one job to do



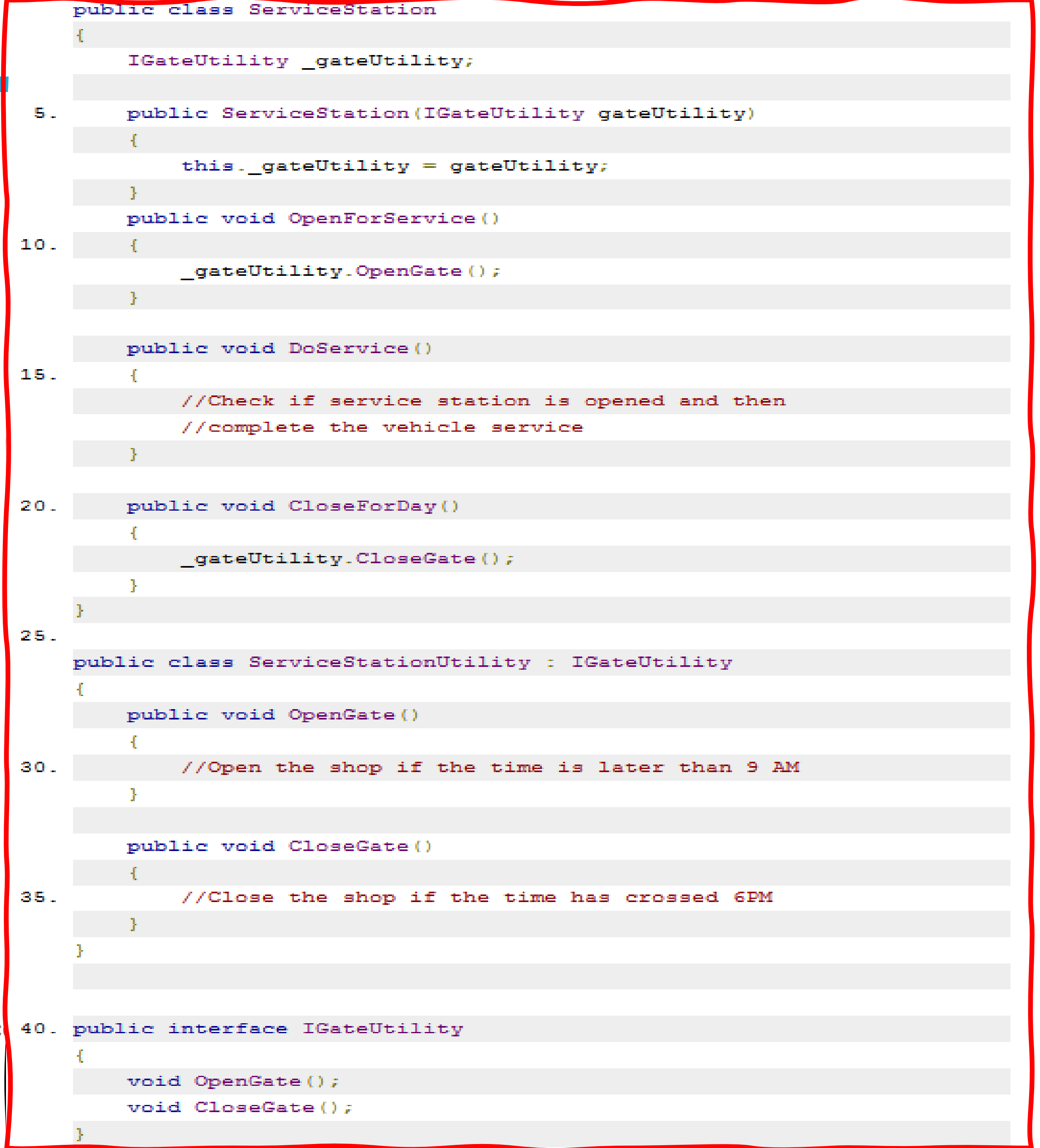
```
public class ServiceStation
{
    public void OpenGate()
    {
        //Open the gate if the time is later than 9 AM
    }

    public void DoService(Vehicle vehicle)
    {
        //Check if service station is opened and then
        //complete the vehicle service
    }

    public void CloseGate()
    {
        //Close the gate if the time has crossed 6PM
    }
}
```

https://www.codeguru.com/columns/experts/solid-principles-in-c-an-overview.htm

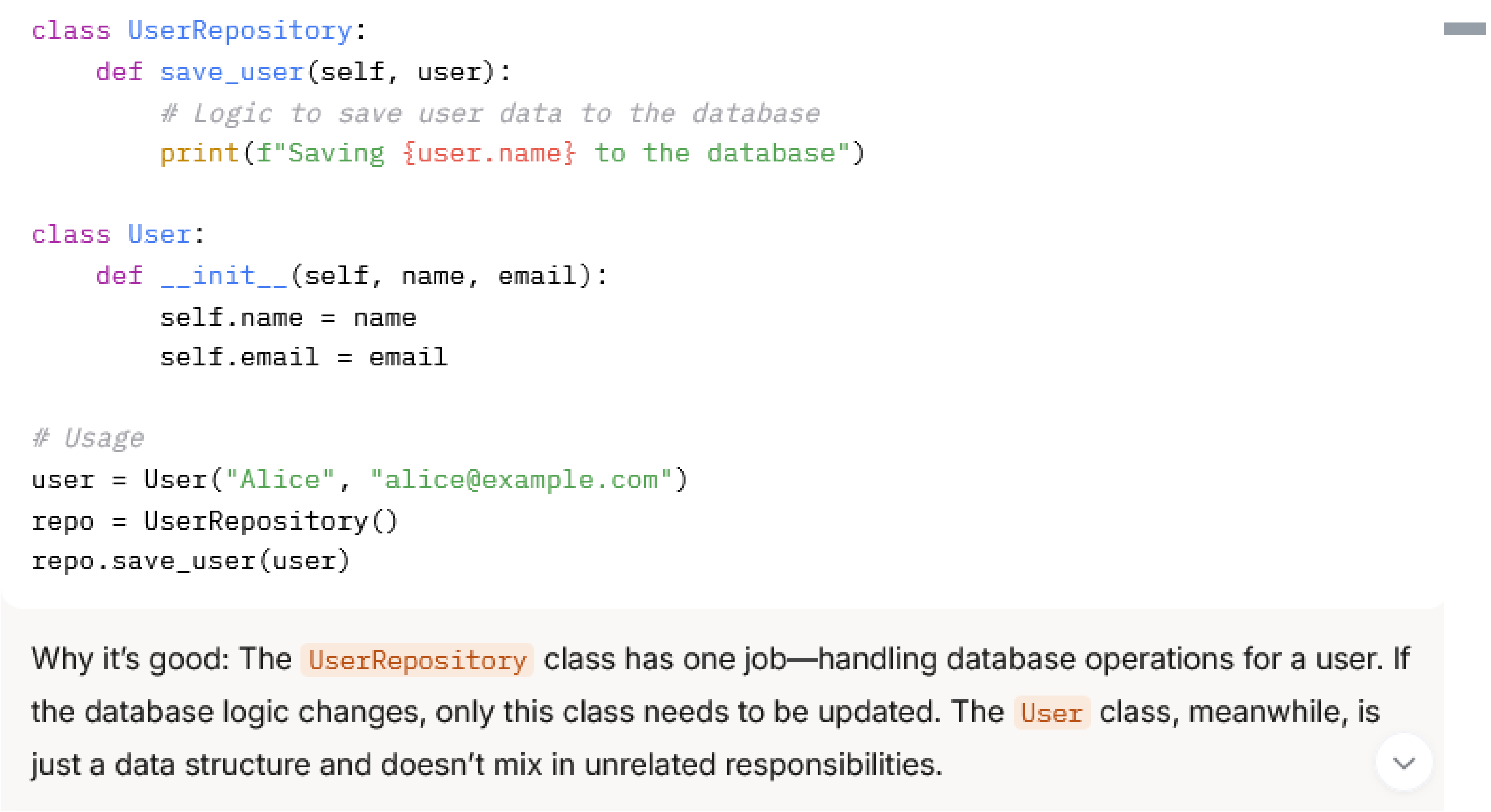
SINGLE RESPONSIBILITY PRINCIPLE



```
public class ServiceStation
{
    public void OpenGate()
    {
        //Open the gate if the time is later than 9 AM
    }

    public void DoService(Vehicle vehicle)
    {
        //Check if service station is opened and then
        //complete the vehicle service
    }

    public void CloseGate()
    {
        //Close the gate if the time has crossed 6PM
    }
}
```



```
public class ServiceStation
{
    IGateUtility _gateUtility;

    public ServiceStation(IGateUtility gateUtility)
    {
        this._gateUtility = gateUtility;
    }
    public void OpenForService()
    {
        _gateUtility.OpenGate();
    }

    public void DoService()
    {
        //Check if service station is opened and then
        //complete the vehicle service
    }

    public void CloseForDay()
    {
        _gateUtility.CloseGate();
    }
}

public class ServiceStationUtility : IGateUtility
{
    public void OpenGate()
    {
        //Open the shop if the time is later than 9 AM
    }

    public void CloseGate()
    {
        //Close the shop if the time has crossed 6PM
    }
}


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



```python
class UserRepository:
    def save_user(self, user):
        # Logic to save user data to the database
        print(f"Saving {user.name} to the database")

class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

# Usage
user = User("Alice", "alice@example.com")
repo = UserRepository()
repo.save_user(user)
```

Why it's good: The `UserRepository` class has one job—handling database operations for a user. If the database logic changes, only this class needs to be updated. The `User` class, meanwhile, is just a data structure and doesn't mix in unrelated responsibilities.

```python
class UserManager:
    def process_user(self, name, email):
        # Create user
        self.name = name
        self.email = email

        # Save to database
        print(f"Saving {self.name} to the database")

        # Send email notification
        print(f"Sending email to {self.email}")

# Usage
manager = UserManager()
manager.process_user("Bob", "bob@example.com")
```

Why it's bad: The `UserManager` class has multiple responsibilities: it handles user data, database saving, and email sending. If the database logic changes, or if the email service needs a new provider, this single class has to be modified for unrelated reasons. This makes it harder to maintain and more prone to bugs.

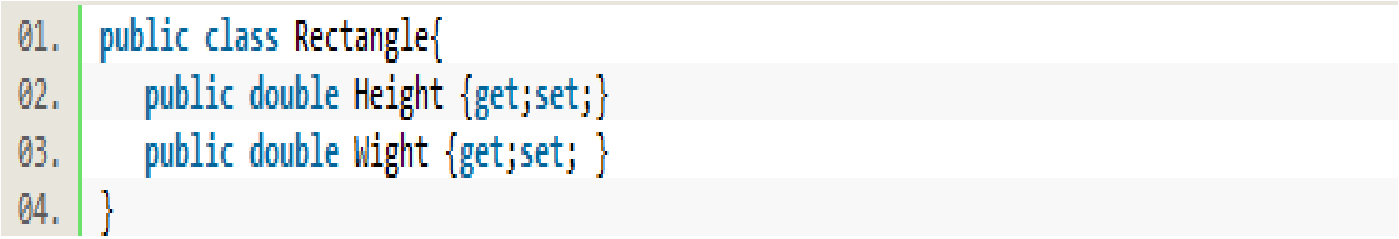
OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.



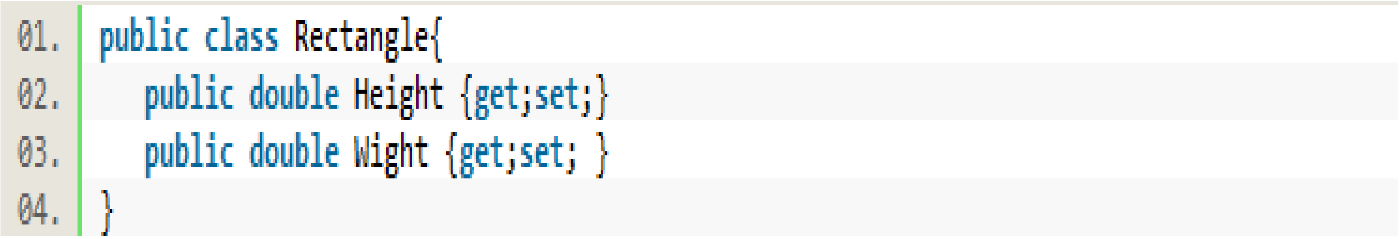
```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
```

- Alanını hesaplayalım

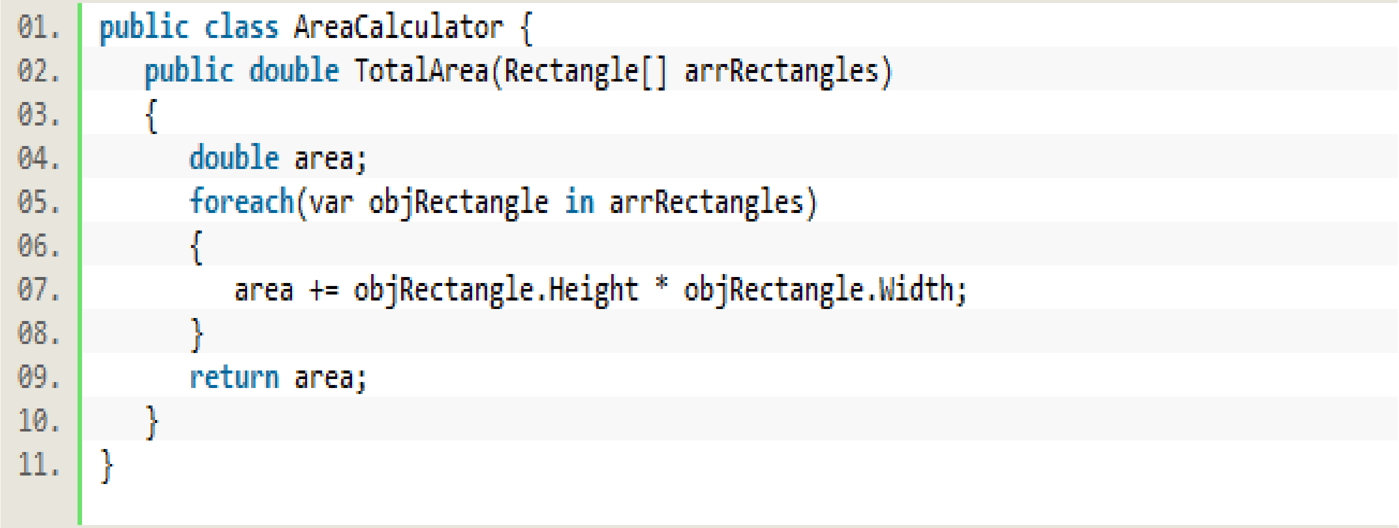
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.



```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}
```



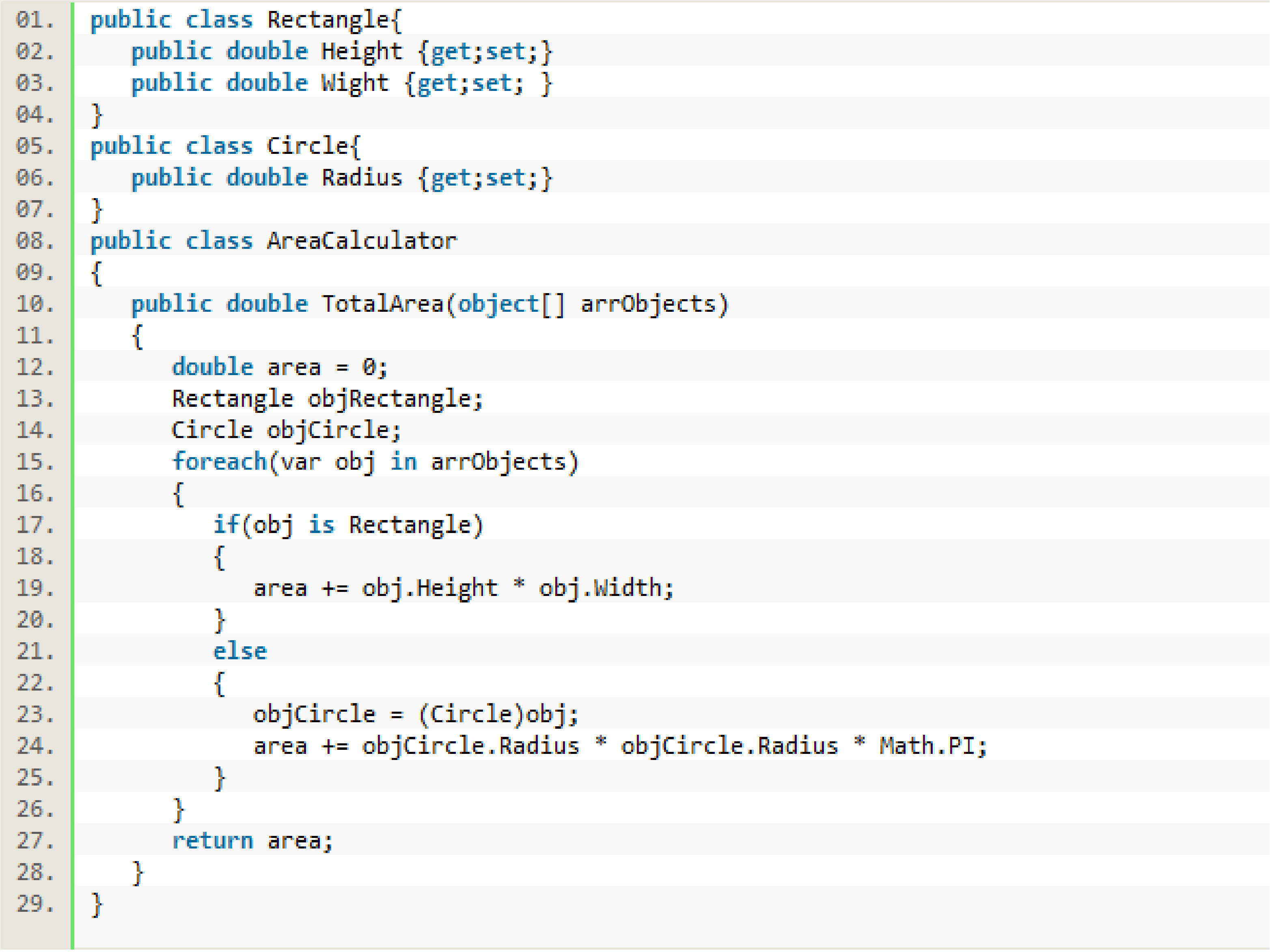
```
public class AreaCalculator {
   public double TotalArea(Rectangle[] arrRectangles)
   {
      double area;
      foreach(var objRectangle in arrRectangles)
      {
         area += objRectangle.Height * objRectangle.Width;
      }
      return area;
   }
}
```

- Daire ekleyelim

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.

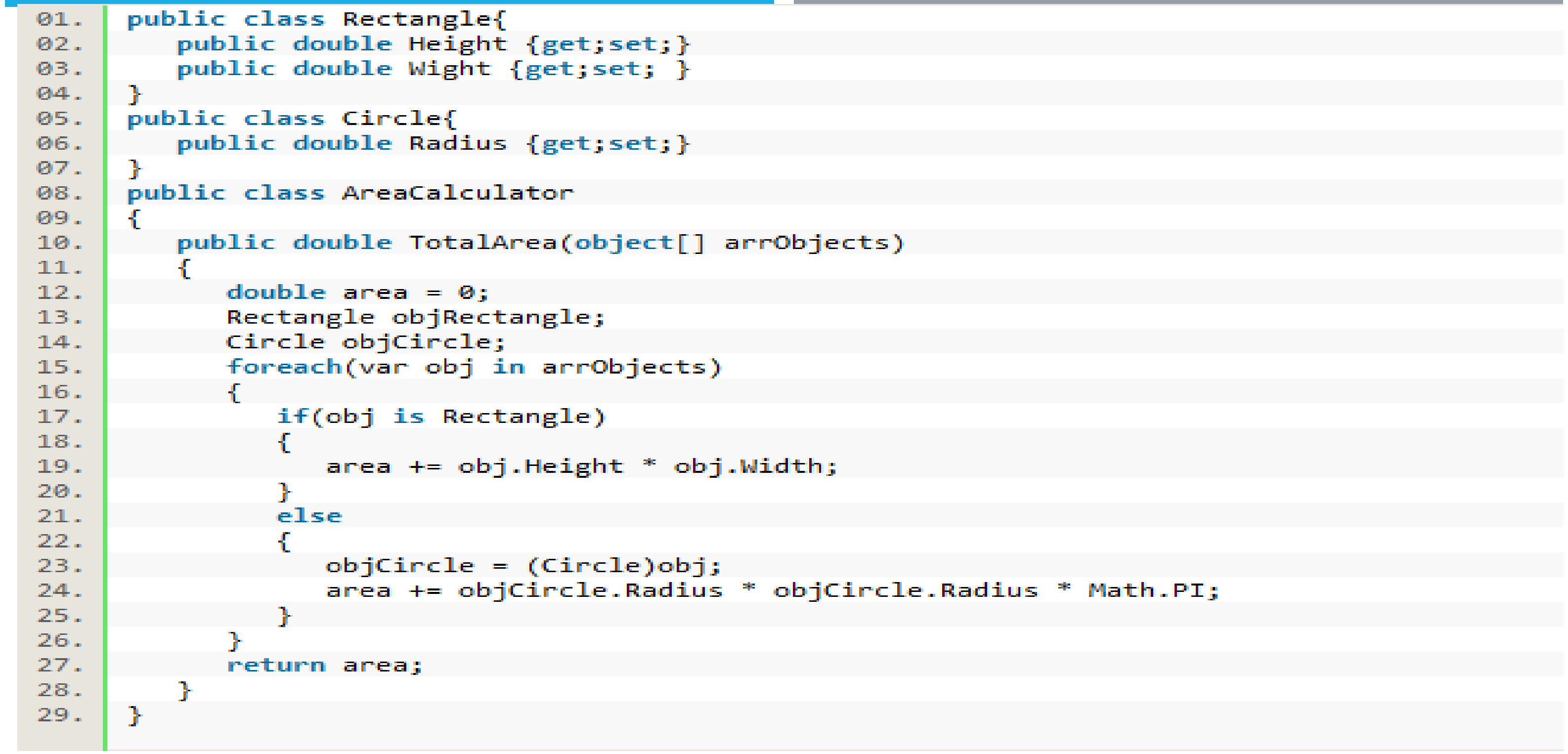


```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        Circle objCircle;
        foreach(var obj in arrObjects)
        {
            if(obj is Rectangle)
            {
                area += obj.Height * obj.Width;
            }
            else
            {
                objCircle = (Circle)obj;
                area += objCircle.Radius * objCircle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

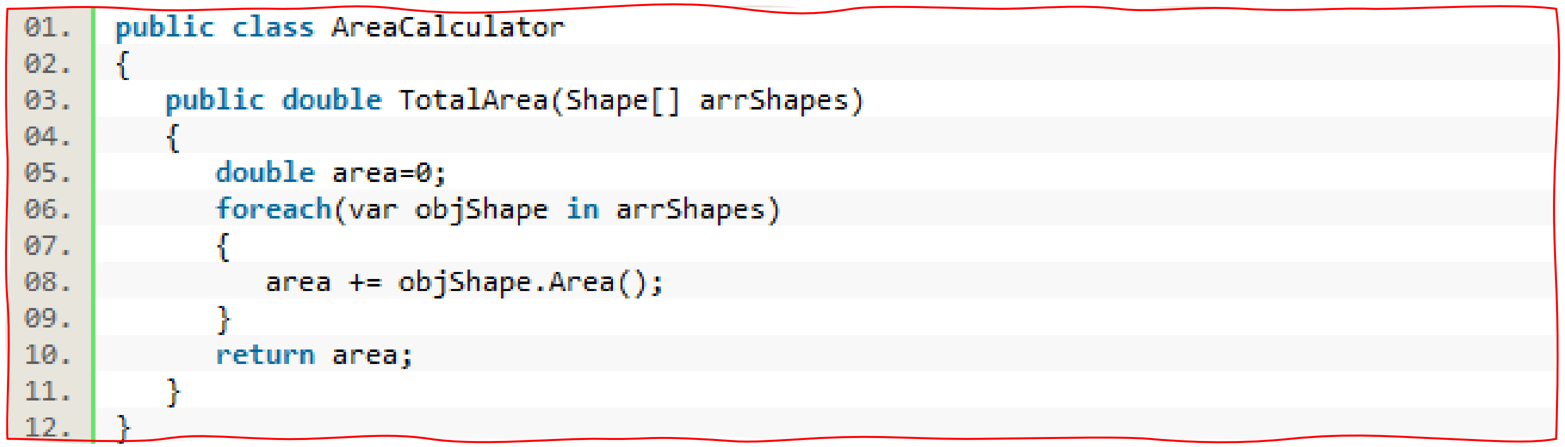
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.



```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        Circle objCircle;
        foreach(var obj in arrObjects)
        {
            if(obj is Rectangle)
            {
                area += obj.Height * obj.Width;
            }
            else
            {
                objCircle = (Circle)obj;
                area += objCircle.Radius * objCircle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

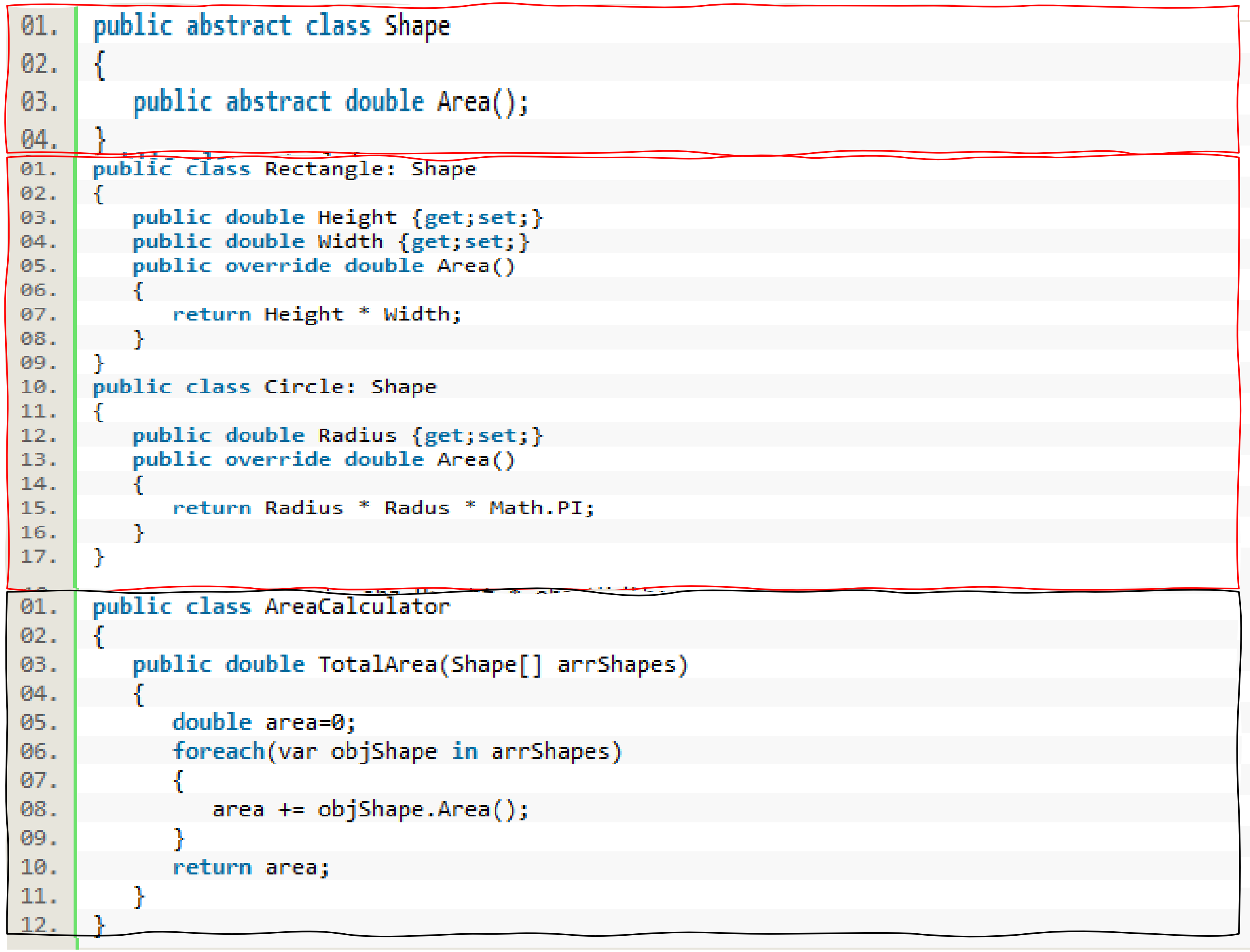


```
public class AreaCalculator
{
    public double TotalArea(Shape[] arrShapes)
    {
        double area=0;
        foreach(var objShape in arrShapes)
        {
            area += objShape.Area();
        }
        return area;
    }
}
```

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

OPEN/CLOSED PRINCIPLE

A software module/class is open for extension and closed for modification.

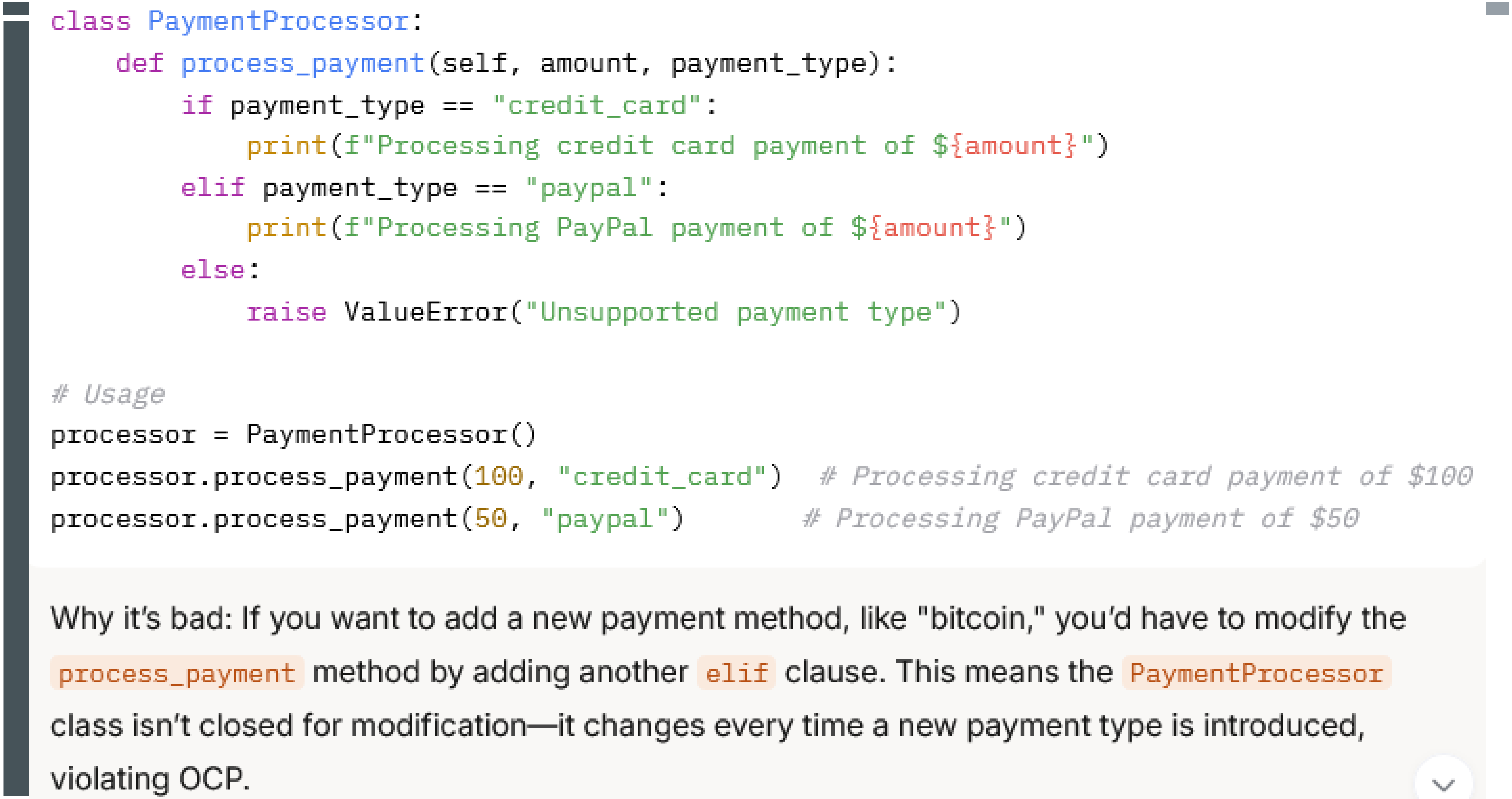


```
public abstract class Shape
{
    public abstract double Area();
}
```

```
public class Rectangle: Shape
{
    public double Height {get;set;}
    public double Width {get;set;}
    public override double Area()
    {
        return Height * Width;
    }
}
public class Circle: Shape
{
    public double Radius {get;set;}
    public override double Area()
    {
        return Radius * Radus * Math.PI;
    }
}
```

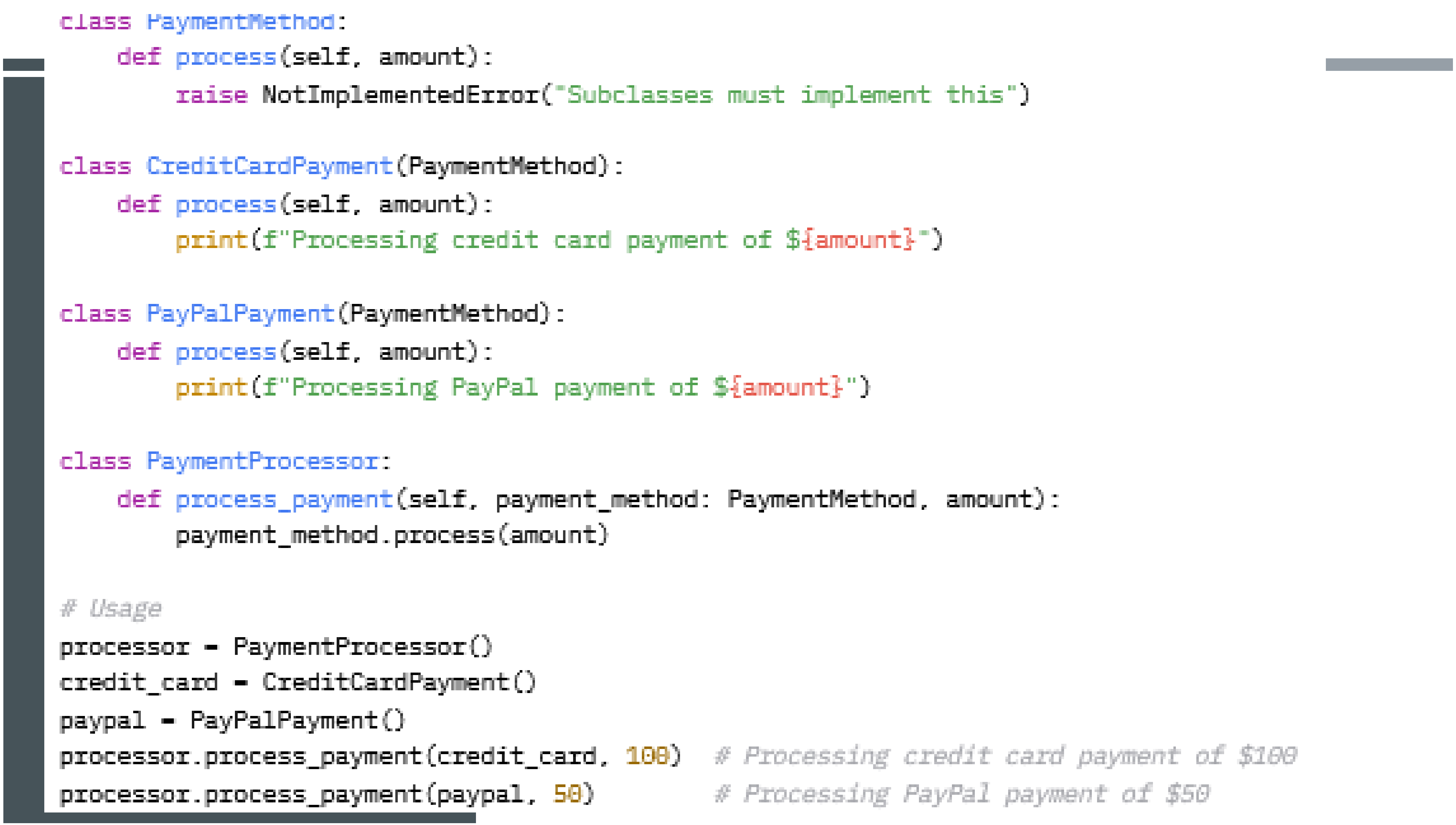
```
public class AreaCalculator
{
    public double TotalArea(Shape[] arrShapes)
    {
        double area=0;
        foreach(var objShape in arrShapes)
        {
            area += objShape.Area();
        }
        return area;
    }
}
```

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

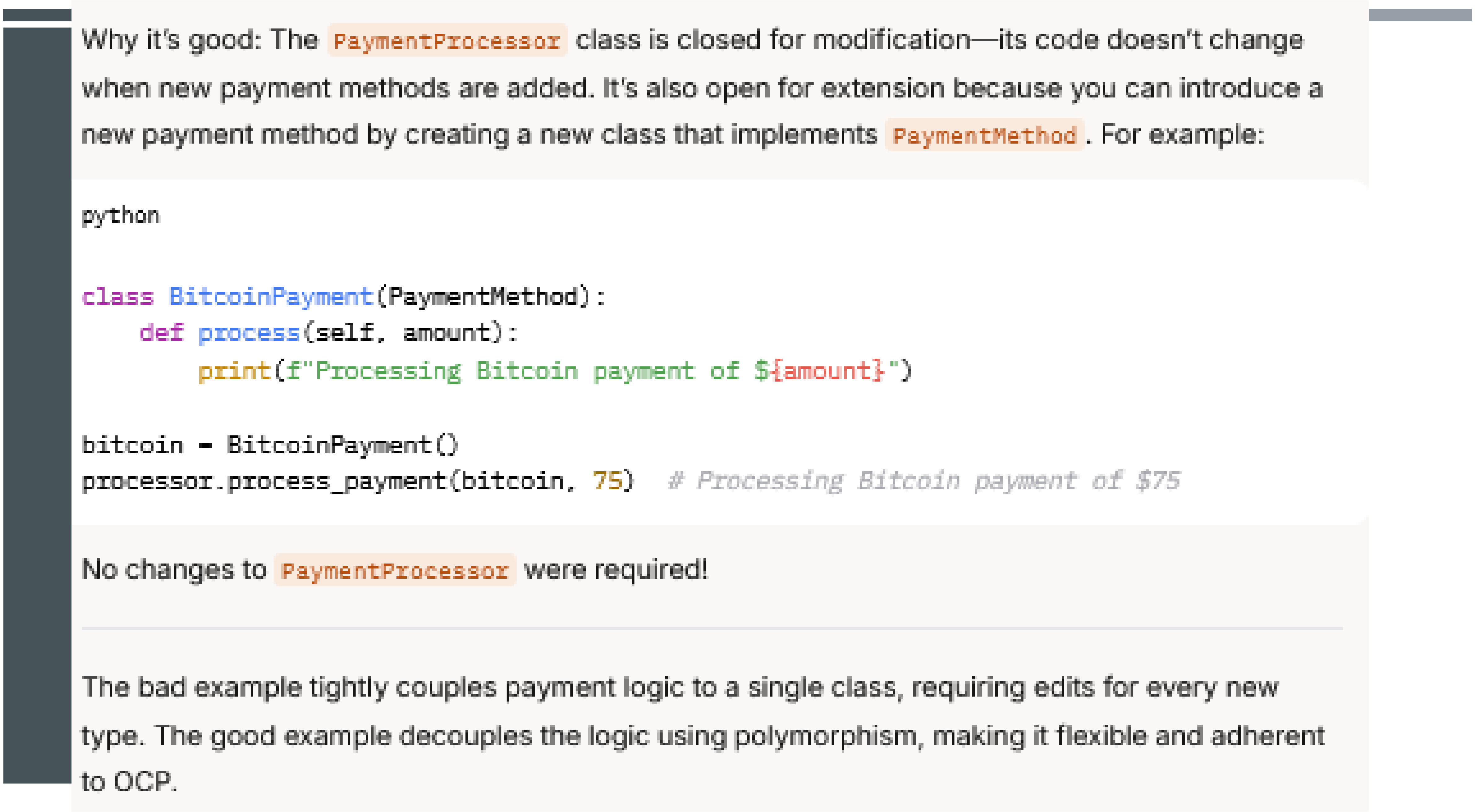


```python
class PaymentProcessor:
    def process_payment(self, amount, payment_type):
        if payment_type == "credit_card":
            print(f"Processing credit card payment of ${amount}")
        elif payment_type == "paypal":
            print(f"Processing PayPal payment of ${amount}")
        else:
            raise ValueError("Unsupported payment type")

# Usage
processor = PaymentProcessor()
processor.process_payment(100, "credit_card")  # Processing credit card payment of $100
processor.process_payment(50, "paypal")        # Processing PayPal payment of $50
```

Why it's bad: If you want to add a new payment method, like "bitcoin," you'd have to modify the `process_payment` method by adding another `elif` clause. This means the `PaymentProcessor` class isn't closed for modification—it changes every time a new payment type is introduced, violating OCP.



```python
class PaymentMethod:
    def process(self, amount):
        raise NotImplementedError("Subclasses must implement this")

class CreditCardPayment(PaymentMethod):
    def process(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(PaymentMethod):
    def process(self, amount):
        print(f"Processing PayPal payment of ${amount}")

class PaymentProcessor:
    def process_payment(self, payment_method: PaymentMethod, amount):
        payment_method.process(amount)

# Usage
processor = PaymentProcessor()
credit_card = CreditCardPayment()
paypal = PayPalPayment()
processor.process_payment(credit_card, 100)  # Processing credit card payment of $100
processor.process_payment(paypal, 50)        # Processing PayPal payment of $50
```



Why it's good: The `PaymentProcessor` class is closed for modification—its code doesn't change when new payment methods are added. It's also open for extension because you can introduce a new payment method by creating a new class that implements `PaymentMethod`. For example:

python

```python
class BitcoinPayment(PaymentMethod):
    def process(self, amount):
        print(f"Processing Bitcoin payment of ${amount}")

bitcoin = BitcoinPayment()
processor.process_payment(bitcoin, 75)  # Processing Bitcoin payment of $75
```

No changes to `PaymentProcessor` were required!

The bad example tightly couples payment logic to a single class, requiring edits for every new type. The good example decouples the logic using polymorphism, making it flexible and adherent to OCP.

LISKOV SUBSTITUTION PRINCIPLE

you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification.

It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.

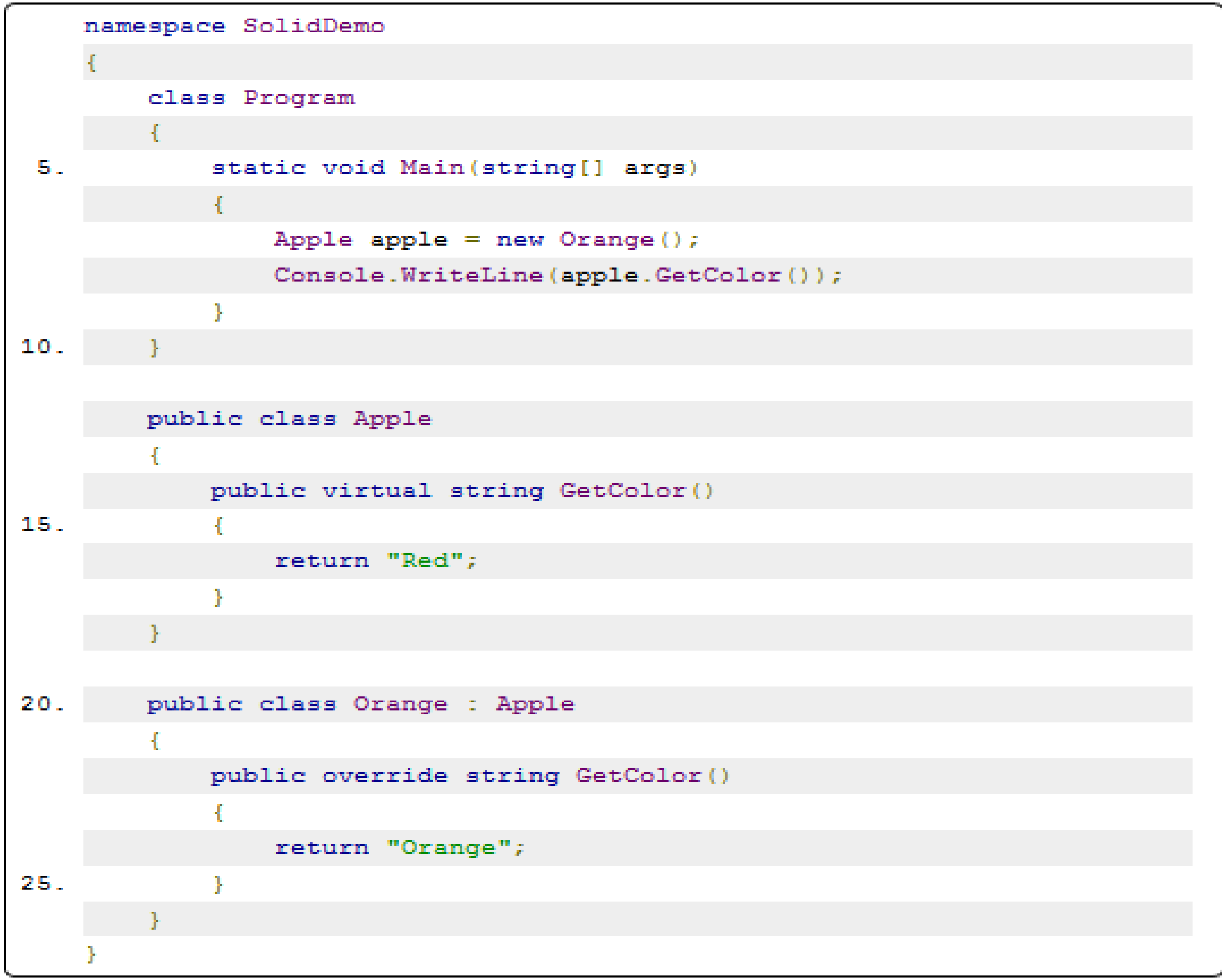
LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

LISKOV SUBSTITUTION PRINCIPLE

you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification.

It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.



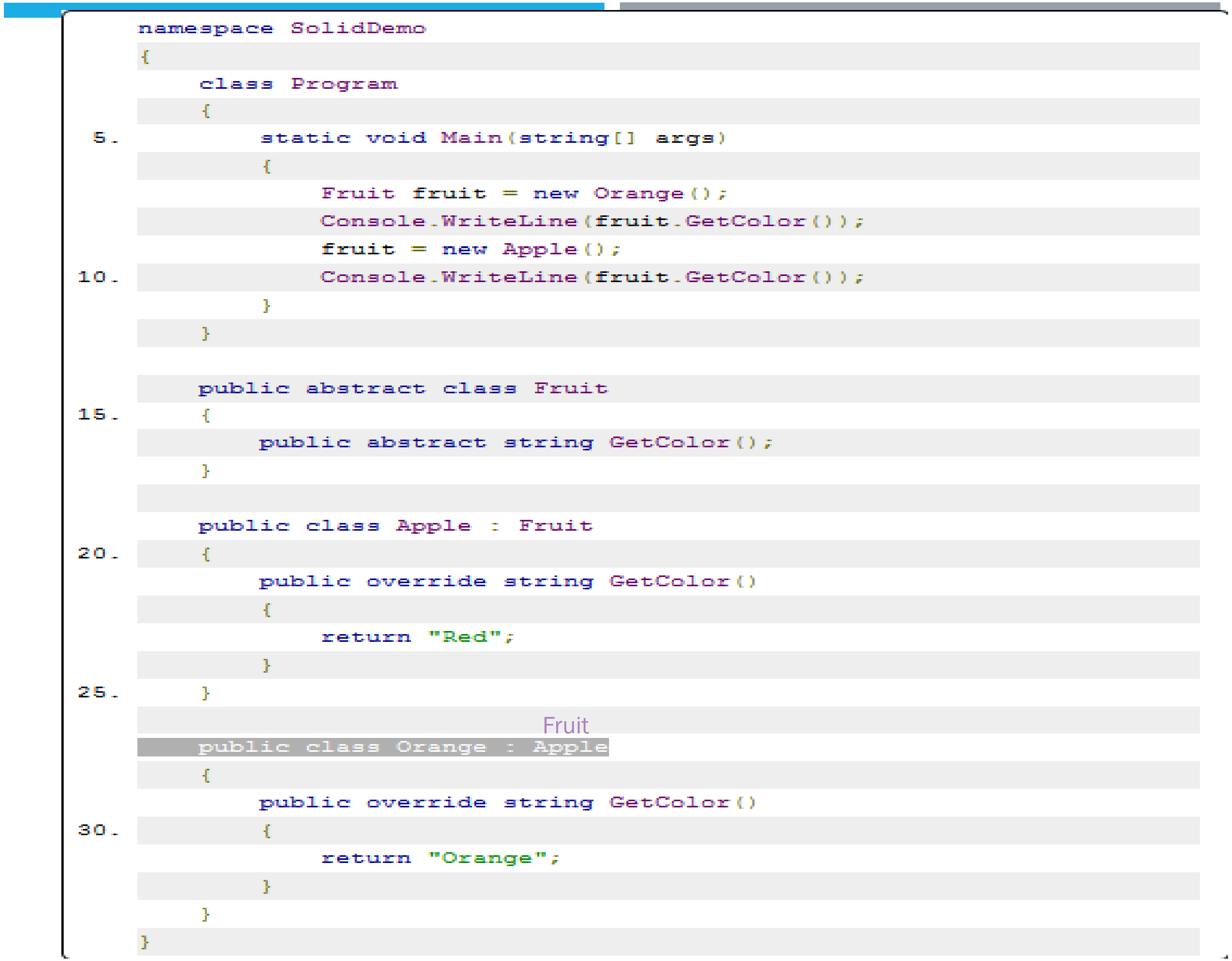
```
namespace SolidDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Apple apple = new Orange();
            Console.WriteLine(apple.GetColor());
        }
    }

    public class Apple
    {
        public virtual string GetColor()
        {
            return "Red";
        }
    }

    public class Orange : Apple
    {
        public override string GetColor()
        {
            return "Orange";
        }
    }
}
```

https://www.codeguru.com/columns/experts/solid-principles-in-c-an-overview.htm

LISKOV SUBSTITUTION PRINCIPLE

you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification.

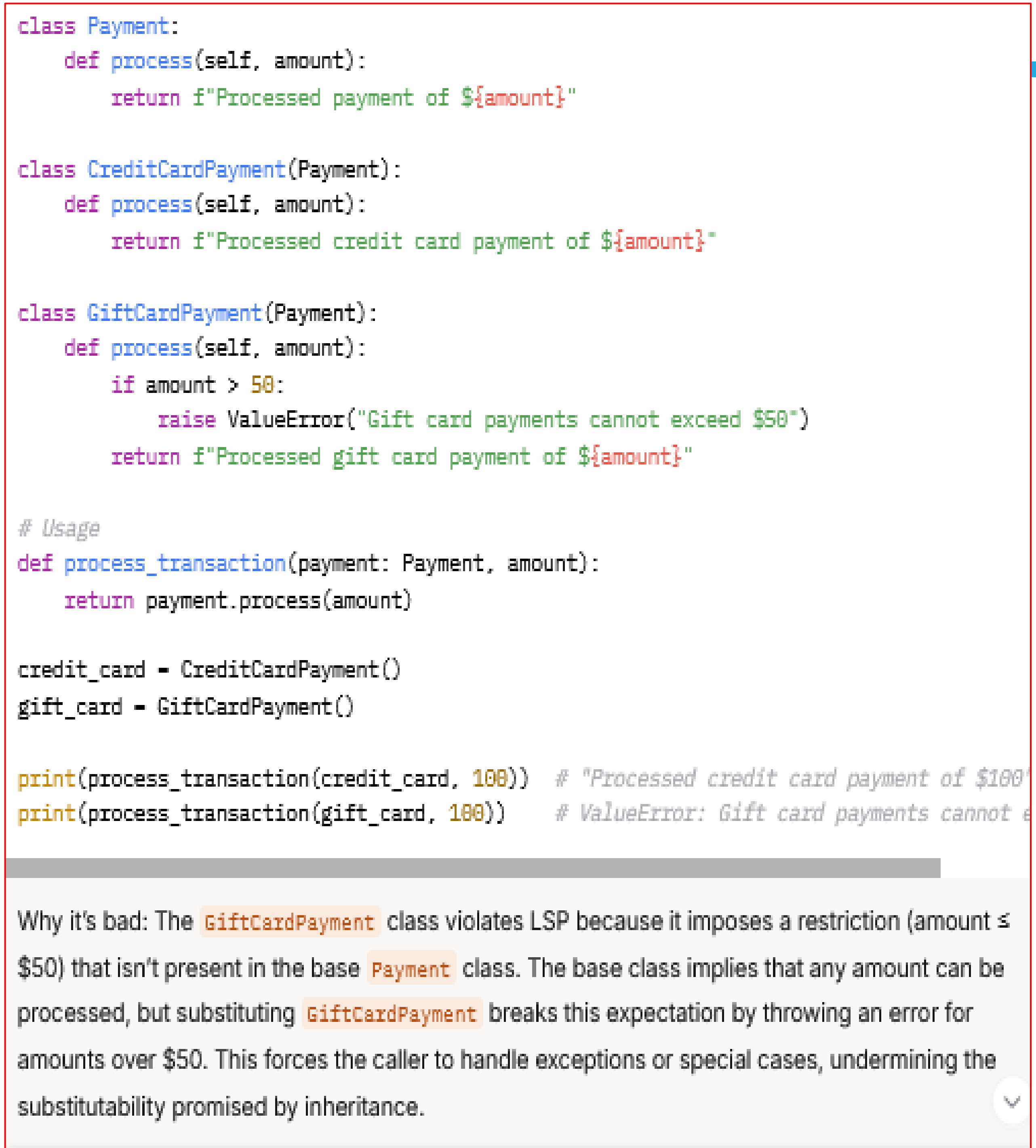
It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.



```
namespace SolidDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Fruit fruit = new Orange();
            Console.WriteLine(fruit.GetColor());
            fruit = new Apple();
            Console.WriteLine(fruit.GetColor());
        }
    }

    public abstract class Fruit
    {
        public abstract string GetColor();
    }

    public class Apple : Fruit
    {
        public override string GetColor()
        {
            return "Red";
        }
    }

    public class Orange : Apple
    {
        public override string GetColor()
        {
            return "Orange";
        }
    }
}
```

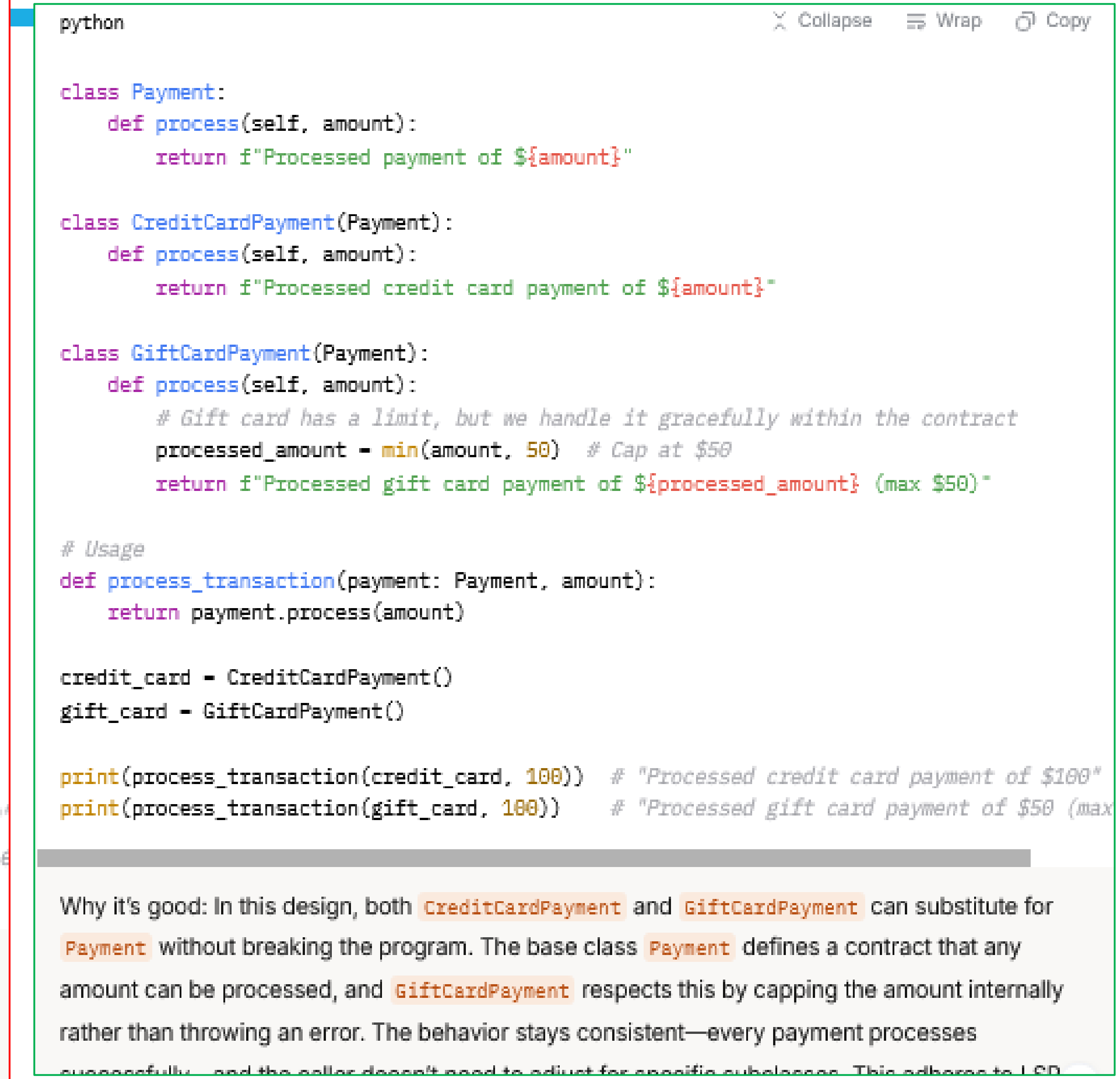
Fruit

https://www.codeguru.com/columns/experts/solid-principles-in-c-an-overview.htm



```python
class Payment:
    def process(self, amount):
        return f"Processed payment of ${amount}"

class CreditCardPayment(Payment):
    def process(self, amount):
        return f"Processed credit card payment of ${amount}"

class GiftCardPayment(Payment):
    def process(self, amount):
        if amount > 50:
            raise ValueError("Gift card payments cannot exceed $50")
        return f"Processed gift card payment of ${amount}"

# Usage
def process_transaction(payment: Payment, amount):
    return payment.process(amount)

credit_card = CreditCardPayment()
gift_card = GiftCardPayment()

print(process_transaction(credit_card, 100))  # "Processed credit card payment of $100"
print(process_transaction(gift_card, 100))    # ValueError: Gift card payments cannot e
```

Why it's bad: The `GiftCardPayment` class violates LSP because it imposes a restriction (amount ≤ $50) that isn't present in the base `Payment` class. The base class implies that any amount can be processed, but substituting `GiftCardPayment` breaks this expectation by throwing an error for amounts over $50. This forces the caller to handle exceptions or special cases, undermining the substitutability promised by inheritance.



```python
class Payment:
    def process(self, amount):
        return f"Processed payment of ${amount}"

class CreditCardPayment(Payment):
    def process(self, amount):
        return f"Processed credit card payment of ${amount}"

class GiftCardPayment(Payment):
    def process(self, amount):
        # Gift card has a limit, but we handle it gracefully within the contract
        processed_amount = min(amount, 50)  # Cap at $50
        return f"Processed gift card payment of ${processed_amount} (max $50)"

# Usage
def process_transaction(payment: Payment, amount):
    return payment.process(amount)

credit_card = CreditCardPayment()
gift_card = GiftCardPayment()

print(process_transaction(credit_card, 100))  # "Processed credit card payment of $100"
print(process_transaction(gift_card, 100))    # "Processed gift card payment of $50 (max
```

Why it's good: In this design, both `CreditCardPayment` and `GiftCardPayment` can substitute for `Payment` without breaking the program. The base class `Payment` defines a contract that any amount can be processed, and `GiftCardPayment` respects this by capping the amount internally rather than throwing an error. The behavior stays consistent—every payment processes successfully—and the caller doesn't need to adjust for specific subclasses. This adheres to LSP

INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.

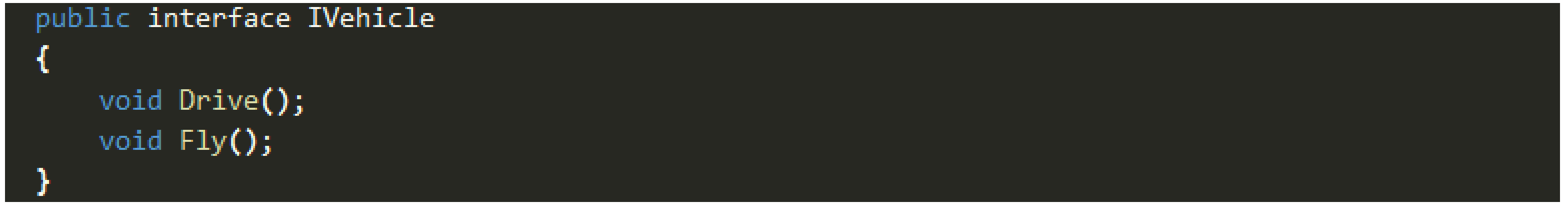
INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

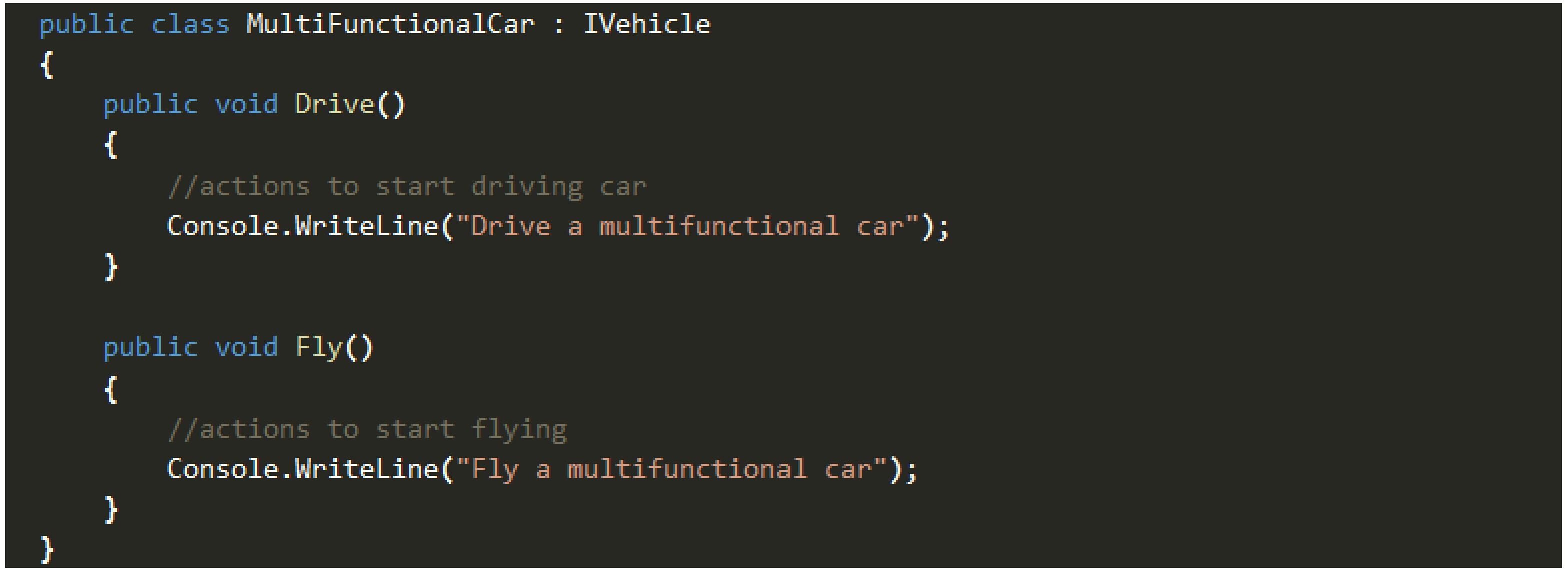
INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.



```
public interface IVehicle
{
    void Drive();
    void Fly();
}
```



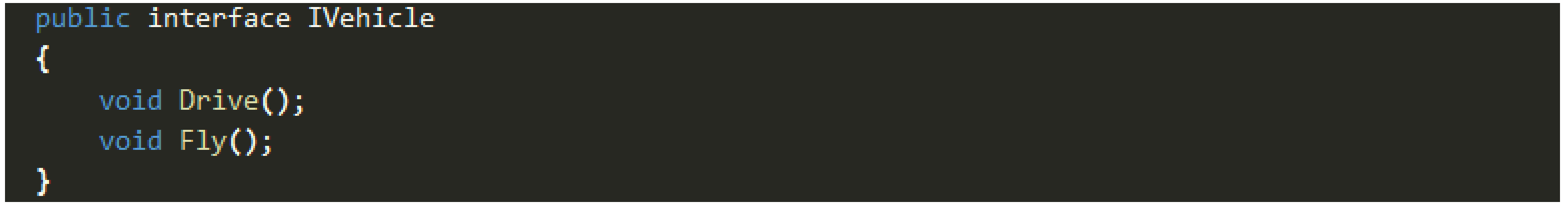
```
public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
}
```

https://code-maze.com/interface-segregation-principle/

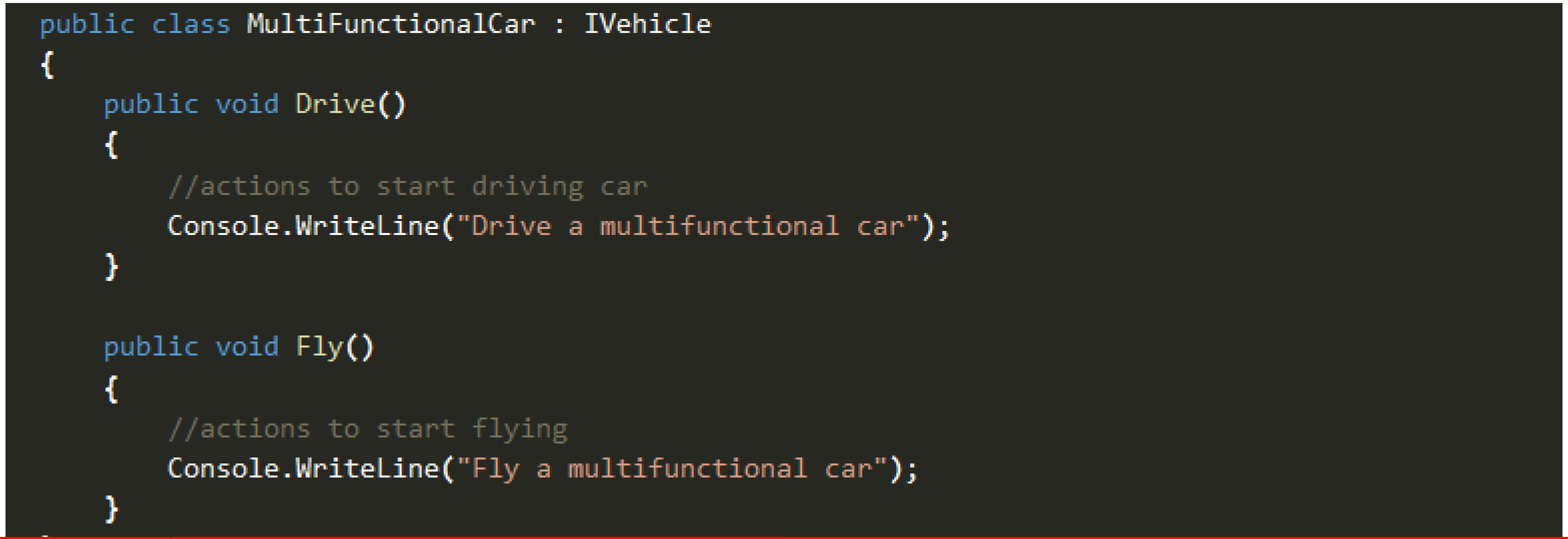
INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

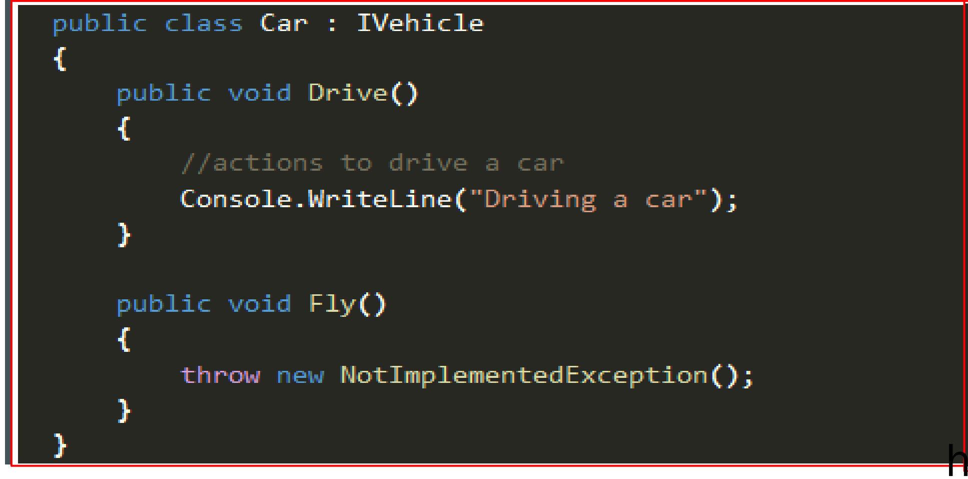
Instead of one fat interface, many small interfaces are



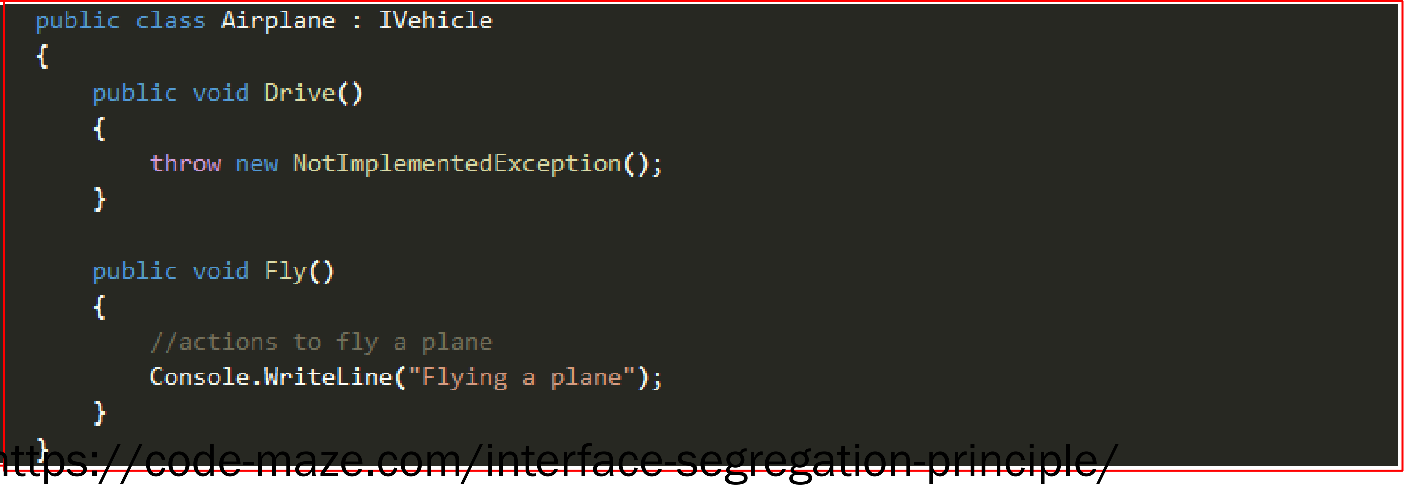
```
public interface IVehicle
{
    void Drive();
    void Fly();
}
```



```
public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
```



```
public class Car : IVehicle
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }

    public void Fly()
    {
        throw new NotImplementedException();
    }
}
```



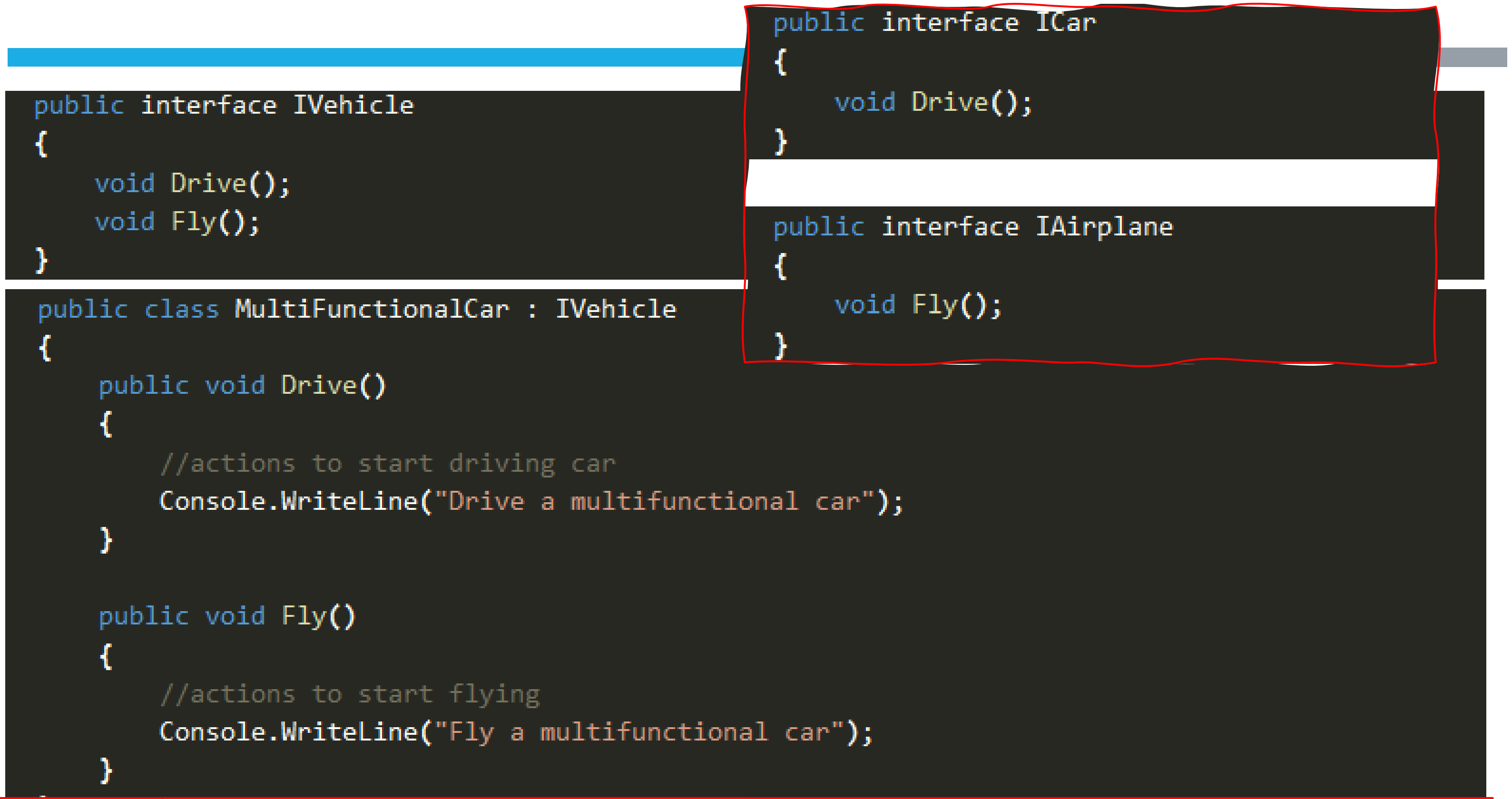
```
public class Airplane : IVehicle
{
    public void Drive()
    {
        throw new NotImplementedException();
    }

    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```

https://code-maze.com/interface-segregation-principle/

INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

Instead of one fat interface, many small interfaces are

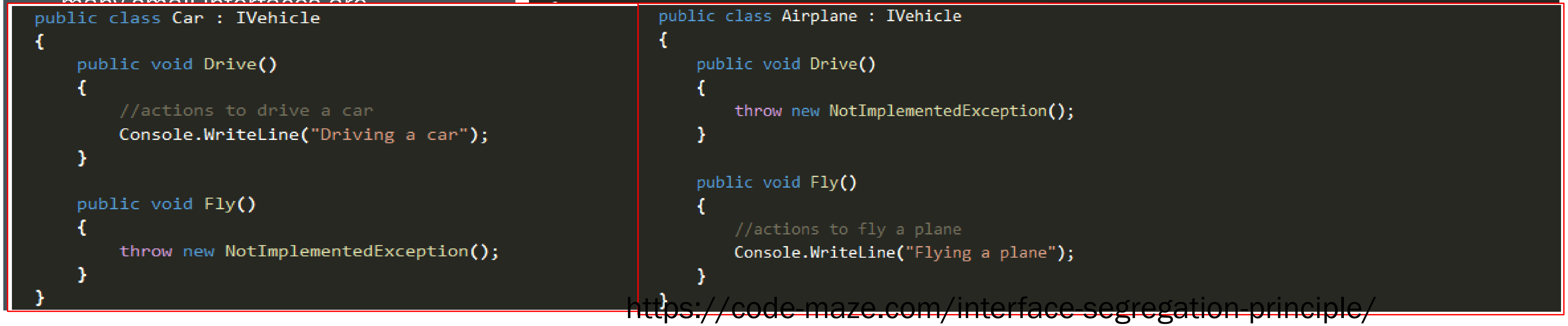


```
public interface IVehicle
{
    void Drive();
    void Fly();
}
```

```
public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
```

```
public interface ICar
{
    void Drive();
}
```

```
public interface IAirplane
{
    void Fly();
}
```



```
public class Car : IVehicle
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }

    public void Fly()
    {
        throw new NotImplementedException();
    }
}
```

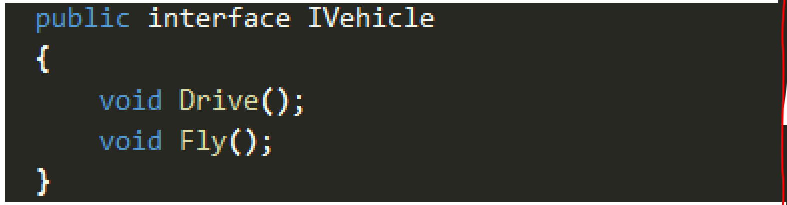
```
public class Airplane : IVehicle
{
    public void Drive()
    {
        throw new NotImplementedException();
    }

    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```

https://code-maze.com/interface-segregation-principle/

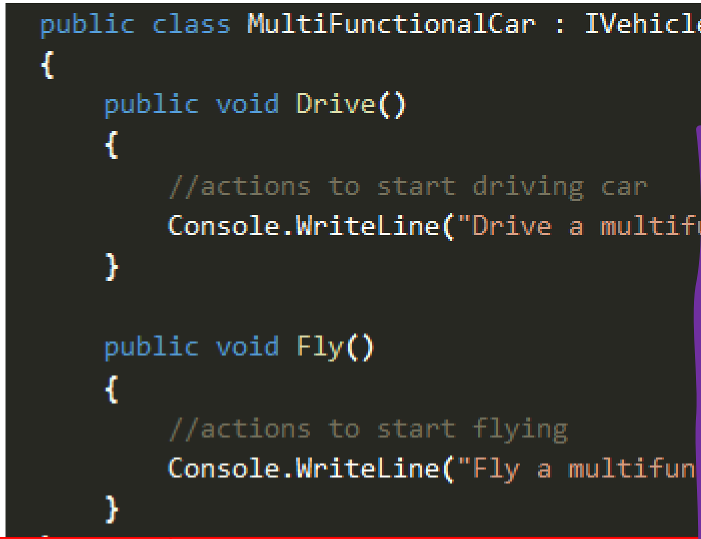
INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

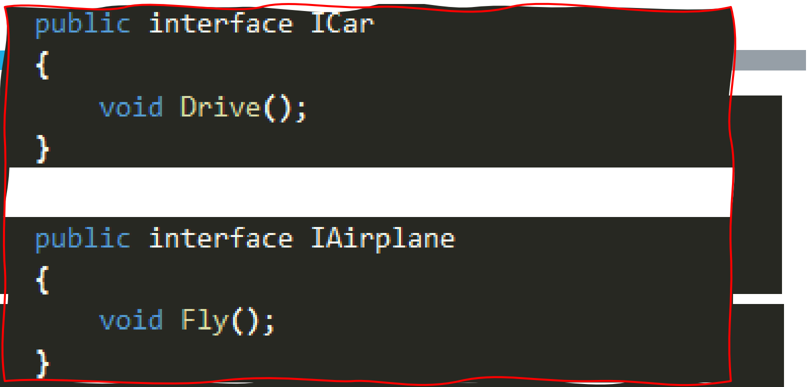
Instead of one fat interface, many small interfaces are



```
public interface IVehicle
{
    void Drive();
    void Fly();
}
```

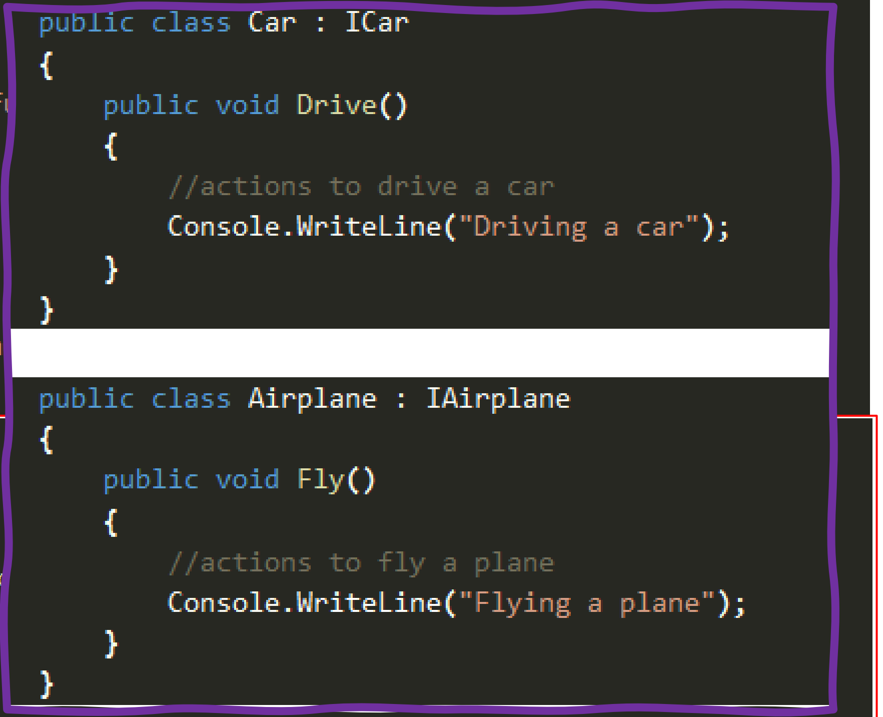


```
public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifu
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifun
    }
```



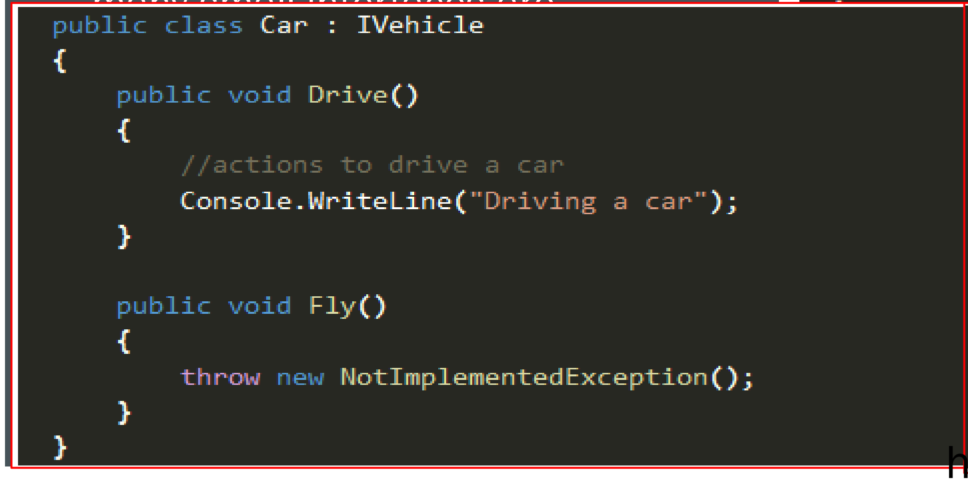
```
public interface ICar
{
    void Drive();
}
```

```
public interface IAirplane
{
    void Fly();
}
```

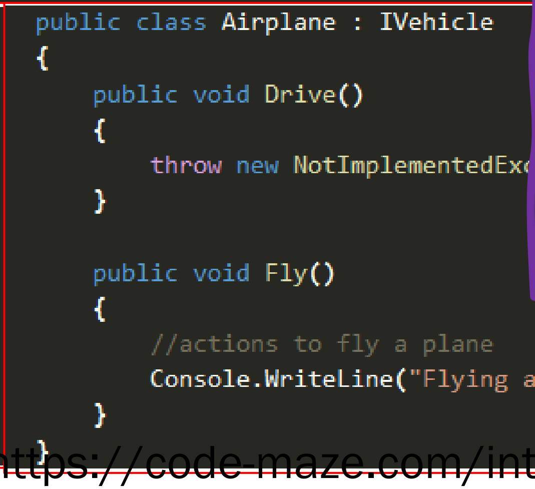


```
public class Car : ICar
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }
}
```

```
public class Airplane : IAirplane
{
    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```



```
public class Car : IVehicle
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }

    public void Fly()
    {
        throw new NotImplementedException();
    }
}
```



```
public class Airplane : IVehicle
{
    public void Drive()
    {
        throw new NotImplementedExc
    }

    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```

https://code-maze.com/interface-segregation-principle/

INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

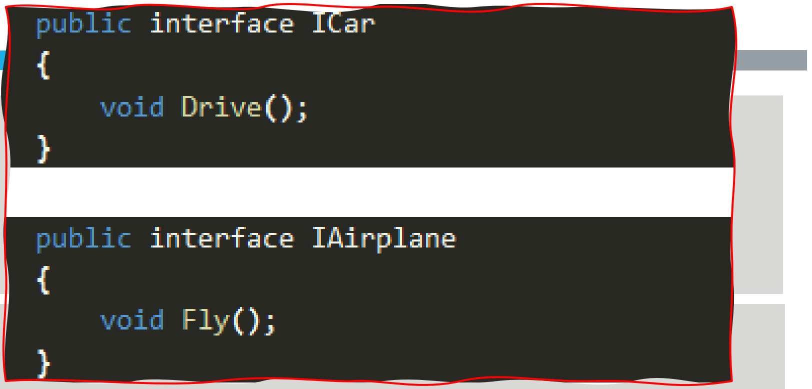
Instead of o

many small

```
public interface IVehicle
{
    void Drive();
    void Fly();
}
```

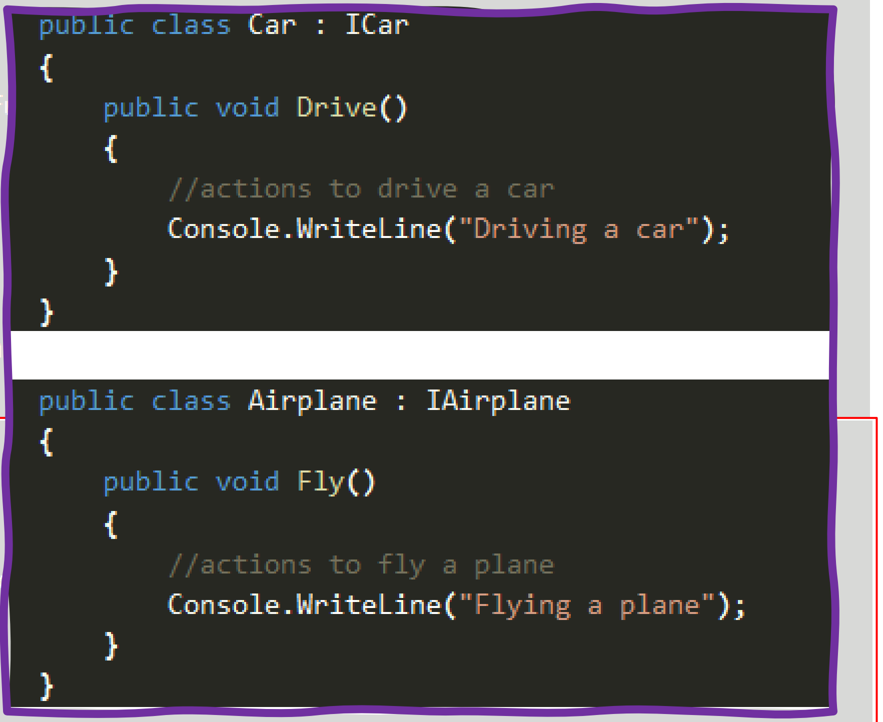
```
public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multif
    }

    public void Fly()
```

```
public class Car
{
    public void
    {
        //action
        Console.
    }

    public void
    {
        throw ne
    }
}
```



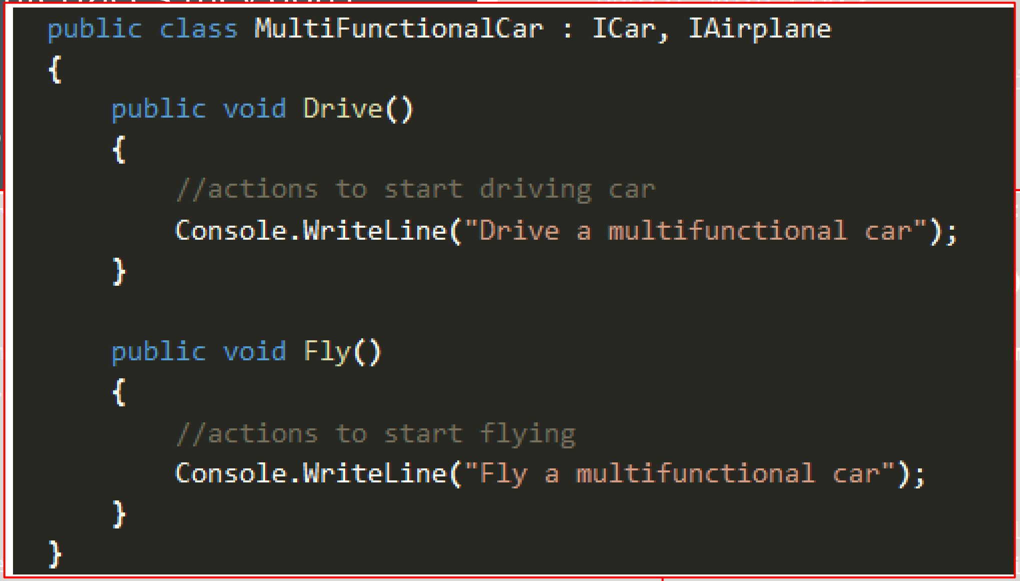
```
public interface ICar
{
    void Drive();
}
```

```
public interface IAirplane
{
    void Fly();
}
```



```
public class Car : ICar
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }
}
```

```
public class Airplane : IAirplane
{
    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```



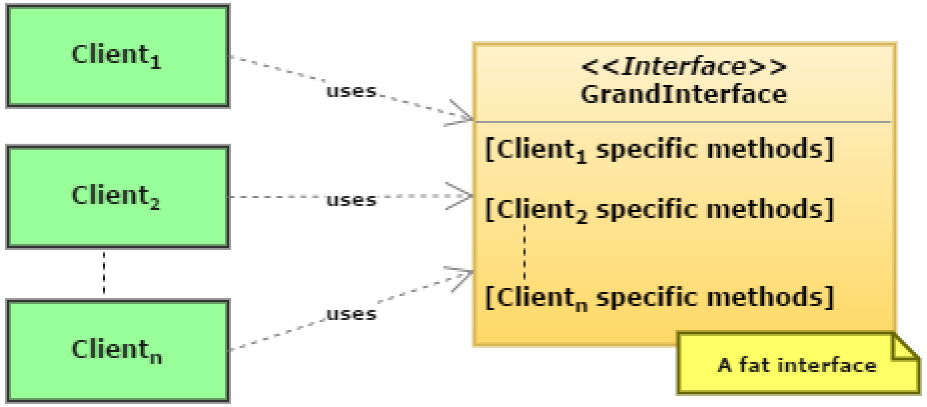
```
public class MultiFunctionalCar : ICar, IAirplane
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
}
```

https://code-maze.com/interface-segregation-principle/

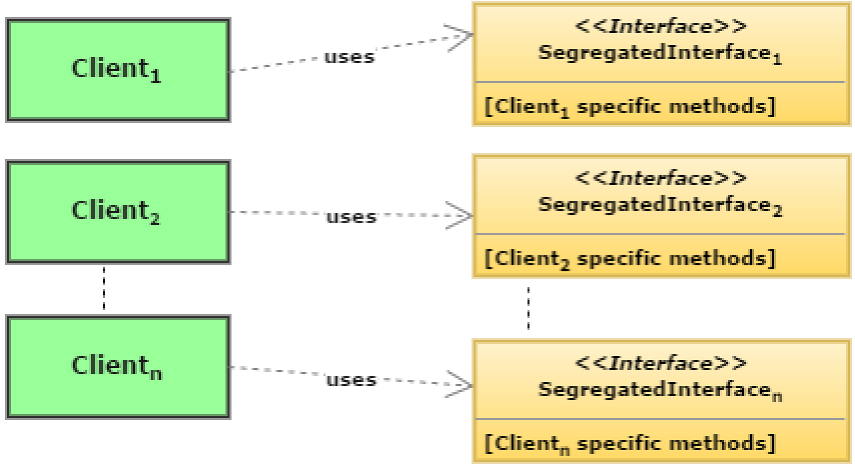
INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to implement interfaces they don't use.

Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.


Client$_1$
Client$_2$
Client$_n$
uses
uses
uses
<<Interface>>
GrandInterface
[Client$_1$ specific methods]
[Client$_2$ specific methods]
[Client$_n$ specific methods]
A fat interface

Before applying Interface Segregation Principle



Refactored interfaces *after* applying Interface Segregation Principle

DEPENDENCY INVERSION PRINCIPLE

High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

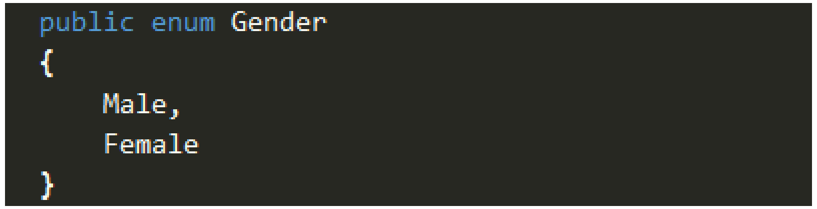
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

DEPENDENCY INVERSION PRINCIPLE

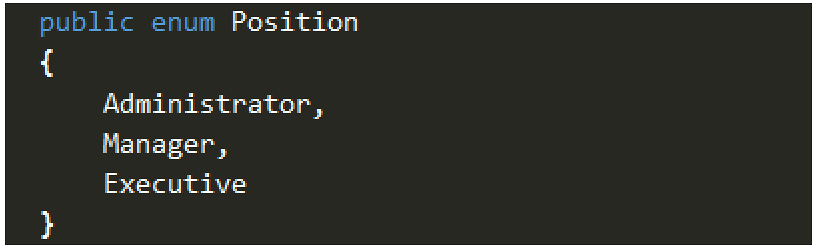
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

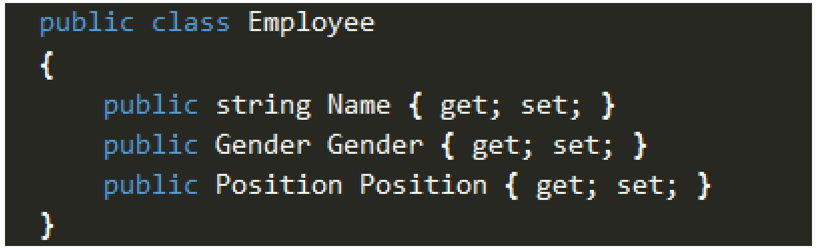
Abstractions should not depend upon details. Details should depend upon abstractions.



```
public enum Gender
{
    Male,
    Female
}
```



```
public enum Position
{
    Administrator,
    Manager,
    Executive
}
```



```
public class Employee
{
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public Position Position { get; set; }
}
```

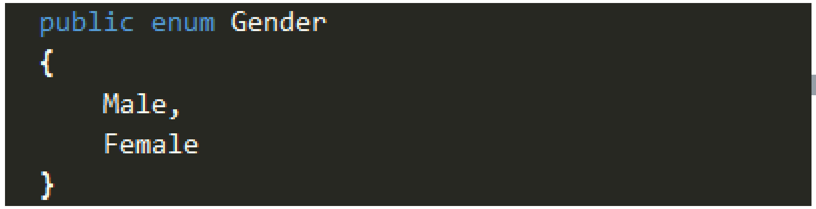
https://code-maze.com/dependency-inversion-principle/

DEPENDENCY INVERSION PRINCIPLE

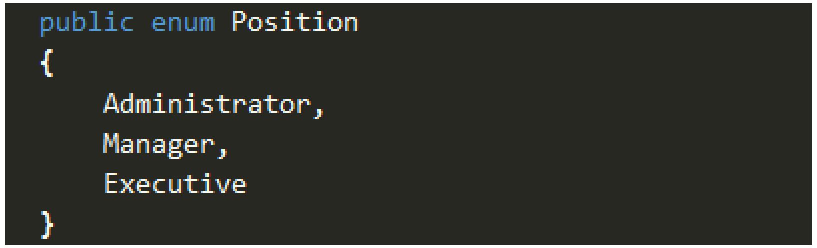
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

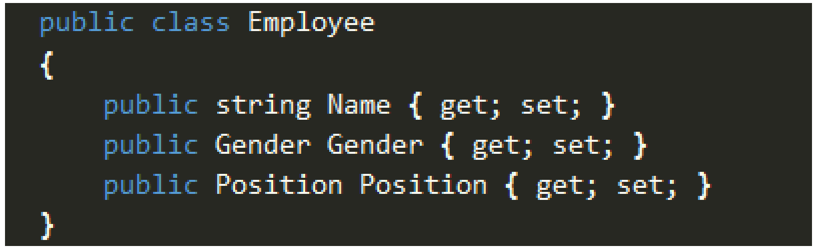
Abstractions should not de upon details. Details shou depend upon abstractions



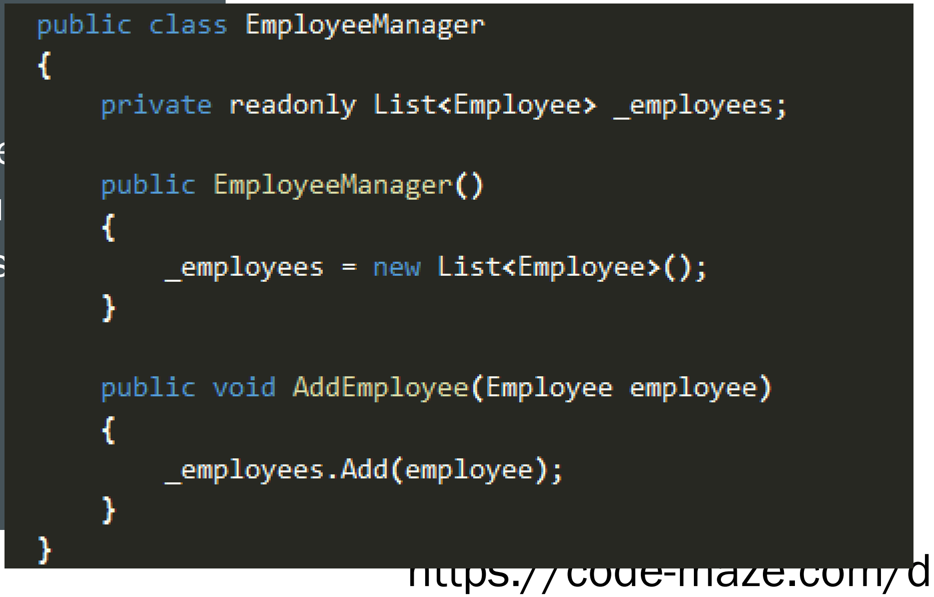
```
public enum Gender
{
    Male,
    Female
}
```



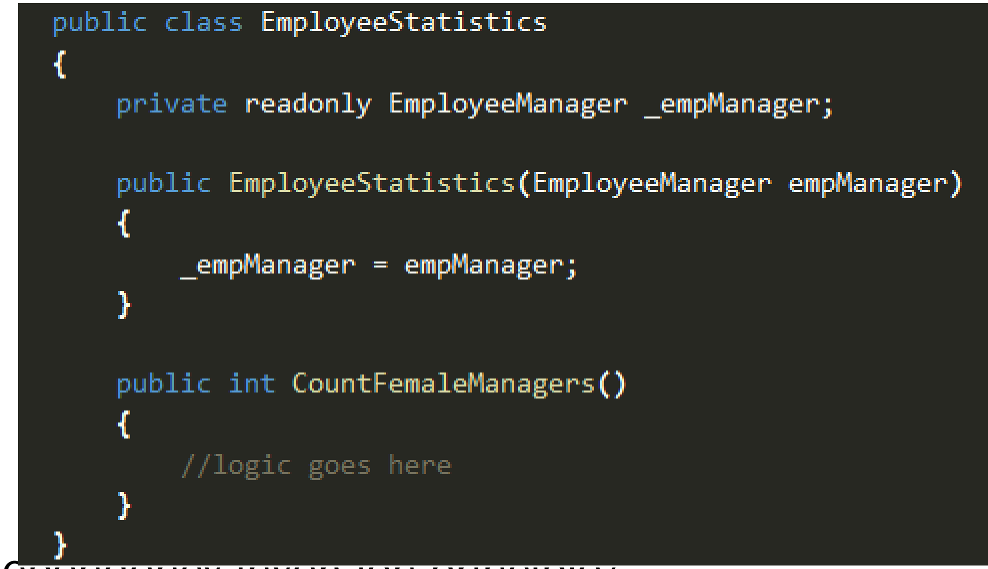
```
public enum Position
{
    Administrator,
    Manager,
    Executive
}
```



```
public class Employee
{
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public Position Position { get; set; }
}
```



```
public class EmployeeManager
{
    private readonly List<Employee> _employees;

    public EmployeeManager()
    {
        _employees = new List<Employee>();
    }

    public void AddEmployee(Employee employee)
    {
        _employees.Add(employee);
    }
}
```



```
public class EmployeeStatistics
{
    private readonly EmployeeManager _empManager;

    public EmployeeStatistics(EmployeeManager empManager)
    {
        _empManager = empManager;
    }

    public int CountFemaleManagers()
    {
        //logic goes here
    }
}
```

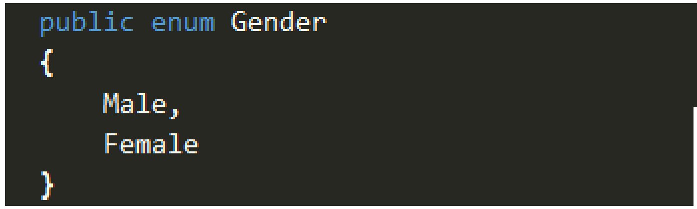
https://code-maze.com/dependency-inversion-principle/

DEPENDENCY INVERSION PRINCIPLE

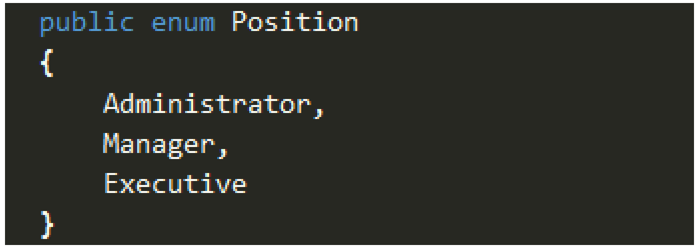
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

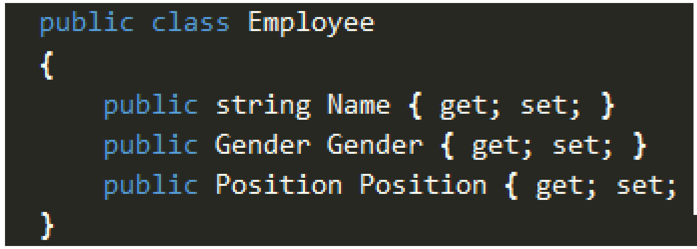
Abstractions should not de
upon details. Details shou
depend upon abstractions



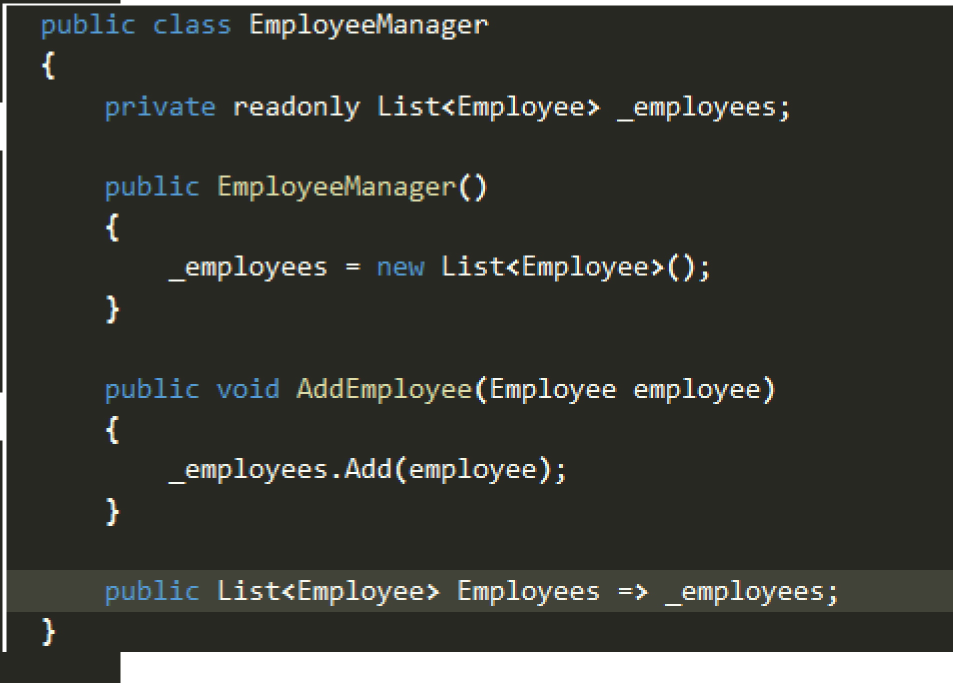
```
public enum Gender
{
    Male,
    Female
}
```



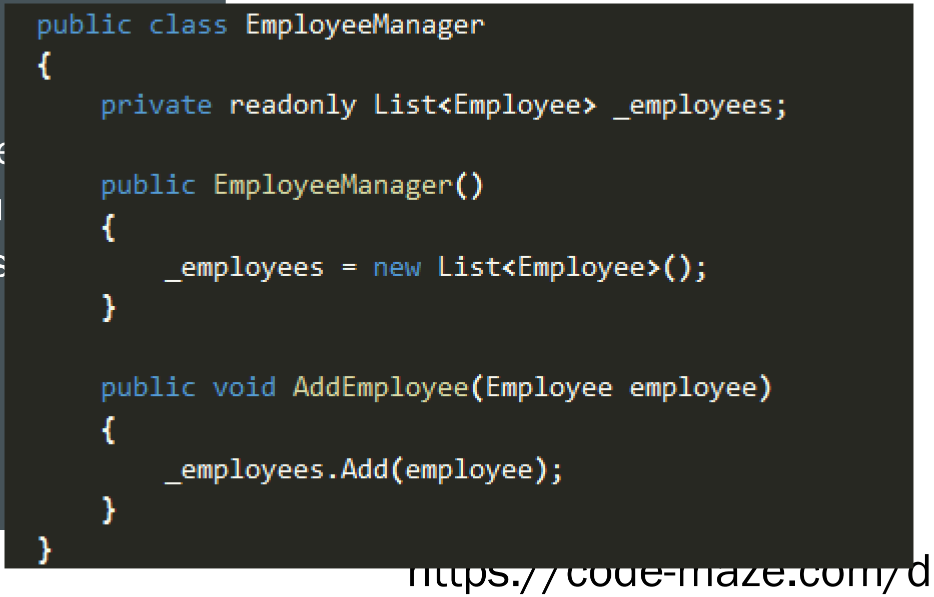
```
public enum Position
{
    Administrator,
    Manager,
    Executive
}
```



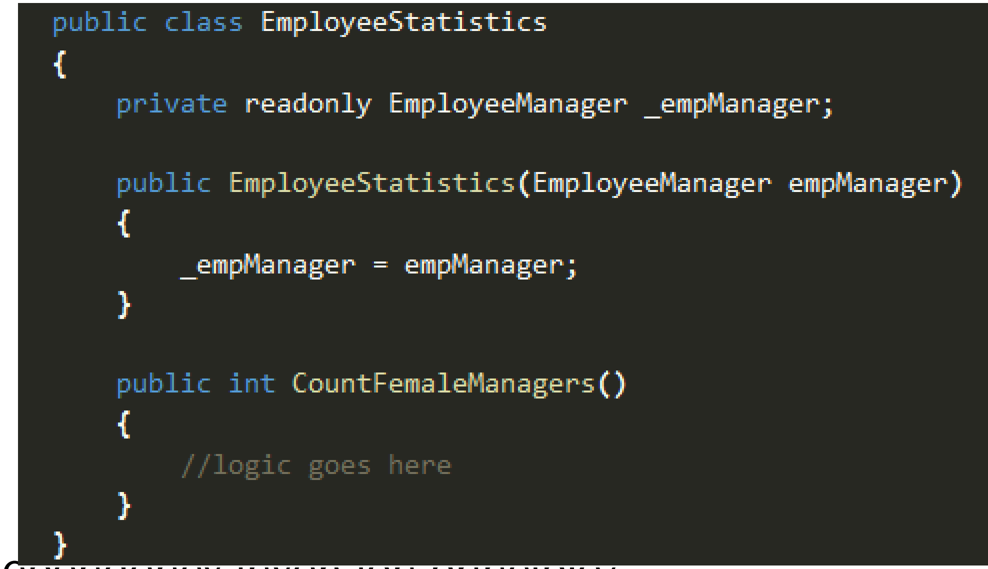
```
public class Employee
{
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public Position Position { get; set;
}
```



```
public class EmployeeManager
{
    private readonly List<Employee> _employees;

    public EmployeeManager()
    {
        _employees = new List<Employee>();
    }

    public void AddEmployee(Employee employee)
    {
        _employees.Add(employee);
    }

    public List<Employee> Employees => _employees;
}
```



```
public class EmployeeManager
{
    private readonly List<Employee> _employees;

    public EmployeeManager()
    {
        _employees = new List<Employee>();
    }

    public void AddEmployee(Employee employee)
    {
        _employees.Add(employee);
    }
}
```



```
public class EmployeeStatistics
{
    private readonly EmployeeManager _empManager;

    public EmployeeStatistics(EmployeeManager empManager)
    {
        _empManager = empManager;
    }

    public int CountFemaleManagers()
    {
        //logic goes here
    }
}
```

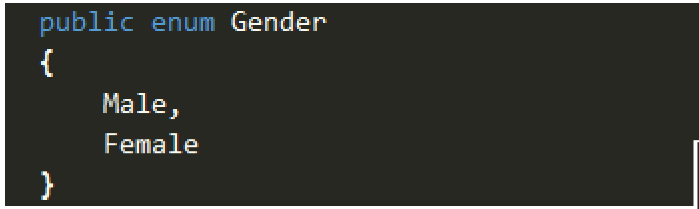
https://code-maze.com/dependency-inversion-principle/

DEPENDENCY INVERSION PRINCIPLE

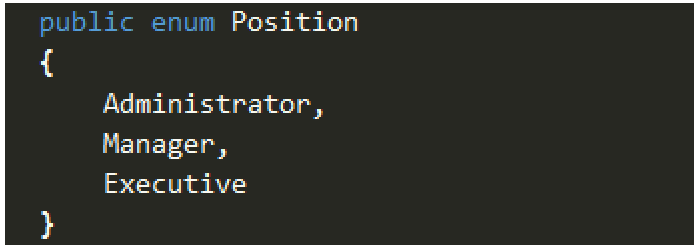
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

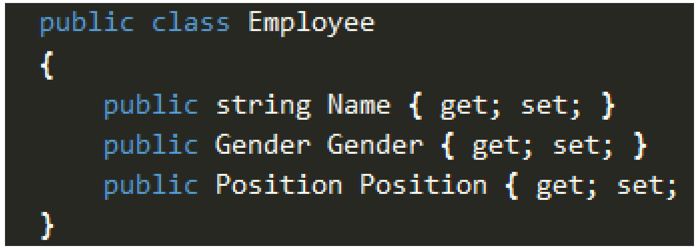
Abstractions should not depend upon details. Details should depend upon abstractions.



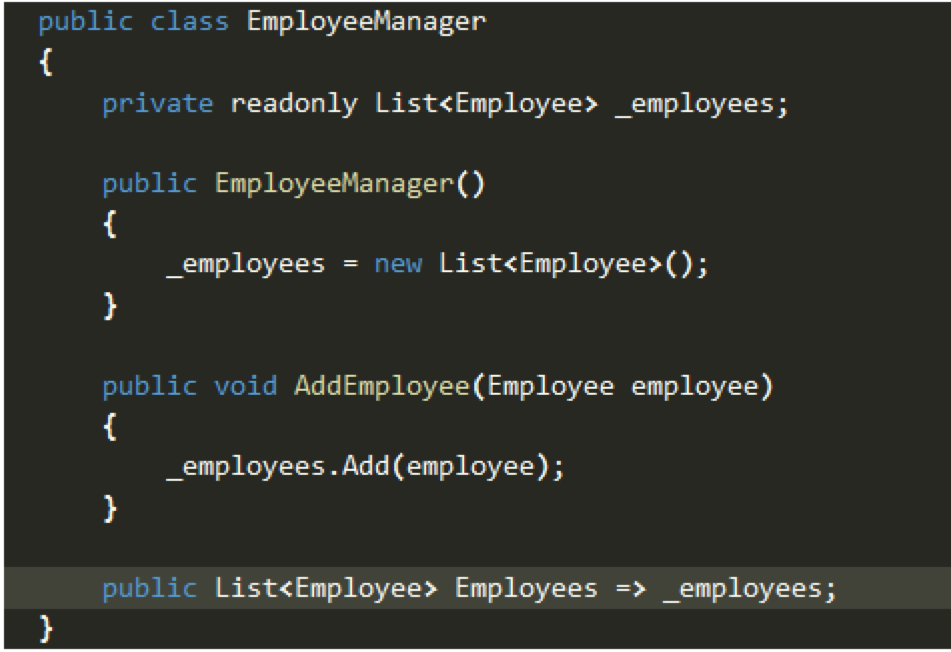
```
public enum Gender
{
    Male,
    Female
}
```



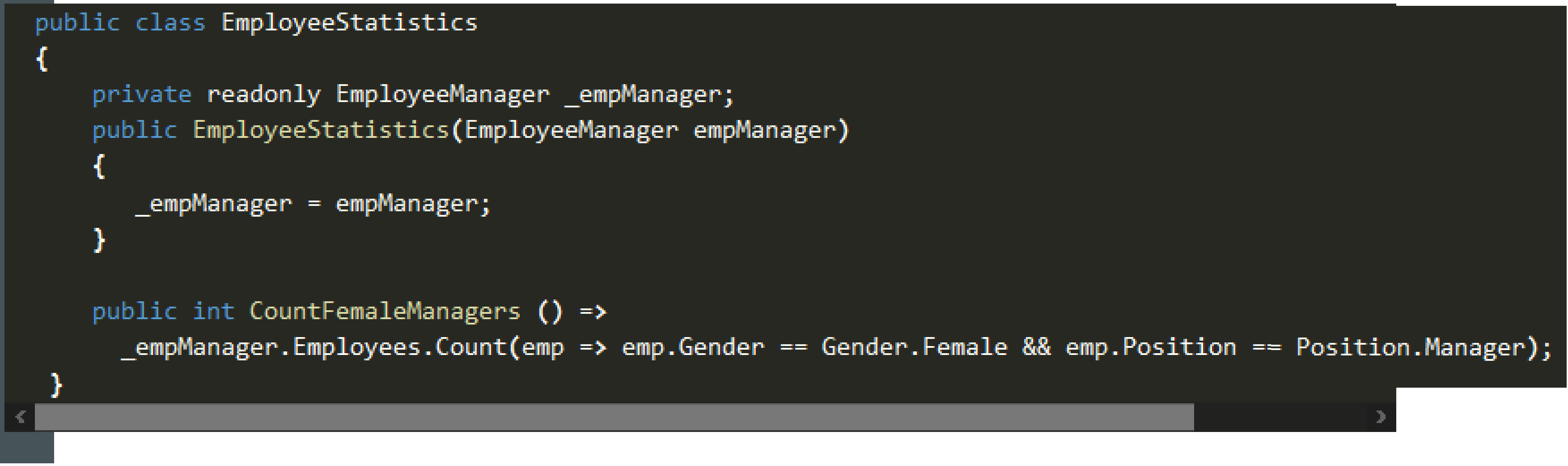
```
public enum Position
{
    Administrator,
    Manager,
    Executive
}
```



```
public class Employee
{
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public Position Position { get; set;
}
```



```
public class EmployeeManager
{
    private readonly List<Employee> _employees;

    public EmployeeManager()
    {
        _employees = new List<Employee>();
    }

    public void AddEmployee(Employee employee)
    {
        _employees.Add(employee);
    }

    public List<Employee> Employees => _employees;
}
```



```
public class EmployeeStatistics
{
    private readonly EmployeeManager _empManager;
    public EmployeeStatistics(EmployeeManager empManager)
    {
        _empManager = empManager;
    }

    public int CountFemaleManagers () =>
      _empManager.Employees.Count(emp => emp.Gender == Gender.Female && emp.Position == Position.Manager);
 }
```

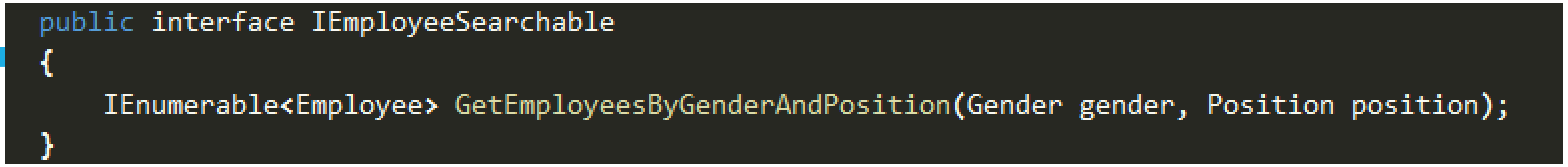
https://code-maze.com/dependency-inversion-principle/

DEPENDENCY INVERSION PRINCIPLE

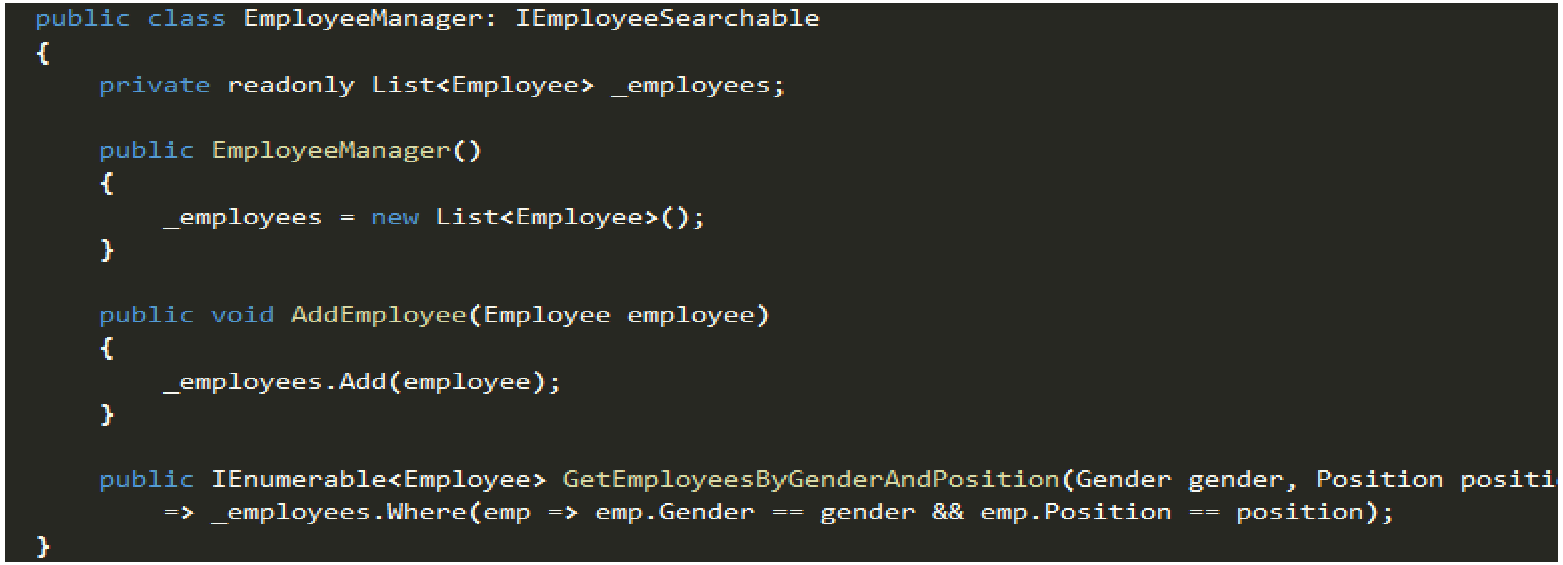
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

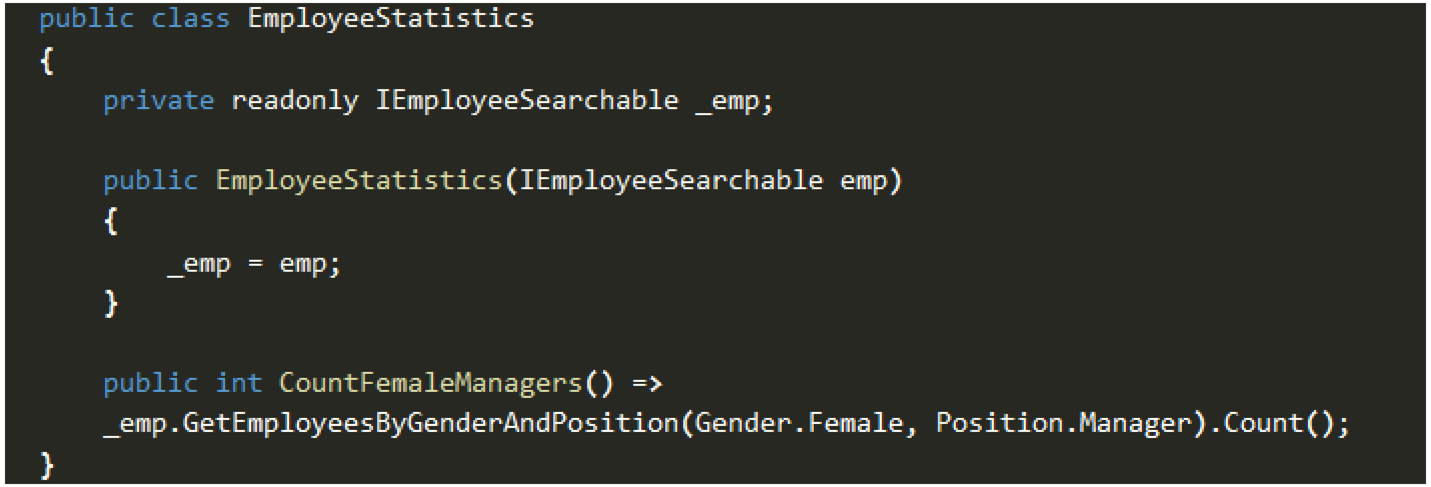
Abstractions should not depend upon details. Details should depend upon abstractions.



```
public interface IEmployeeSearchable
{
    IEnumerable<Employee> GetEmployeesByGenderAndPosition(Gender gender, Position position);
}
```



```
public class EmployeeManager: IEmployeeSearchable
{
    private readonly List<Employee> _employees;

    public EmployeeManager()
    {
        _employees = new List<Employee>();
    }

    public void AddEmployee(Employee employee)
    {
        _employees.Add(employee);
    }

    public IEnumerable<Employee> GetEmployeesByGenderAndPosition(Gender gender, Position positi
        => _employees.Where(emp => emp.Gender == gender && emp.Position == position);
}
```



```
public class EmployeeStatistics
{
    private readonly IEmployeeSearchable _emp;

    public EmployeeStatistics(IEmployeeSearchable emp)
    {
        _emp = emp;
    }

    public int CountFemaleManagers() =>
    _emp.GetEmployeesByGenderAndPosition(Gender.Female, Position.Manager).Count();
}
```

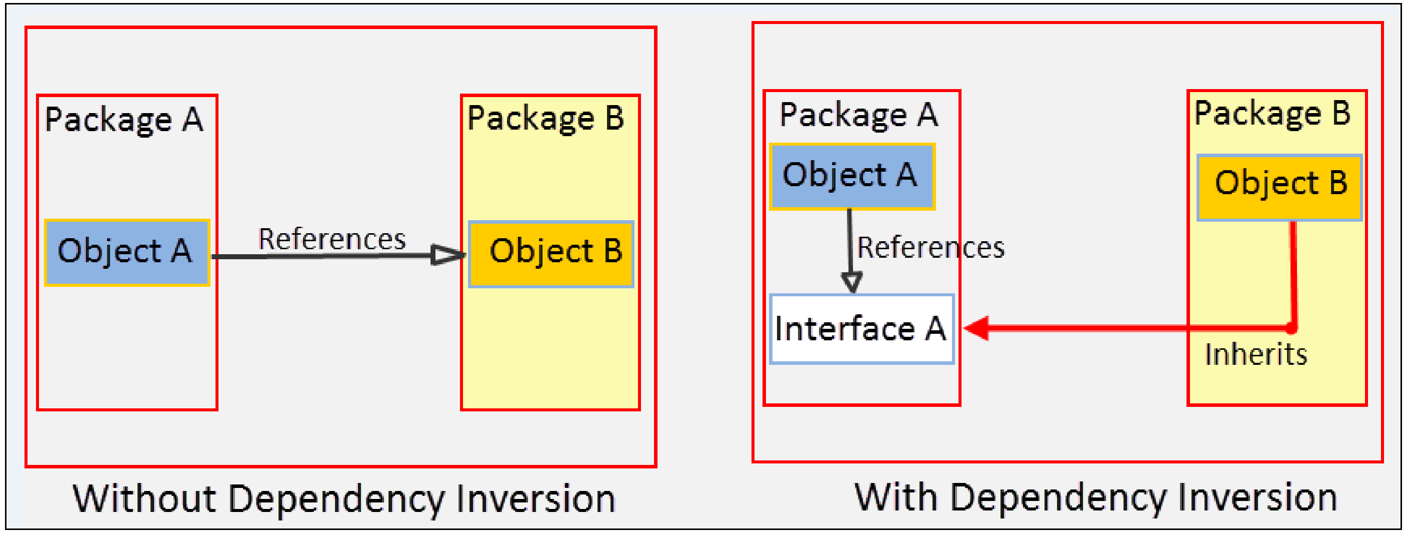
https://code-maze.com/dependency-inversion-principle/

DEPENDENCY INVERSION PRINCIPLE

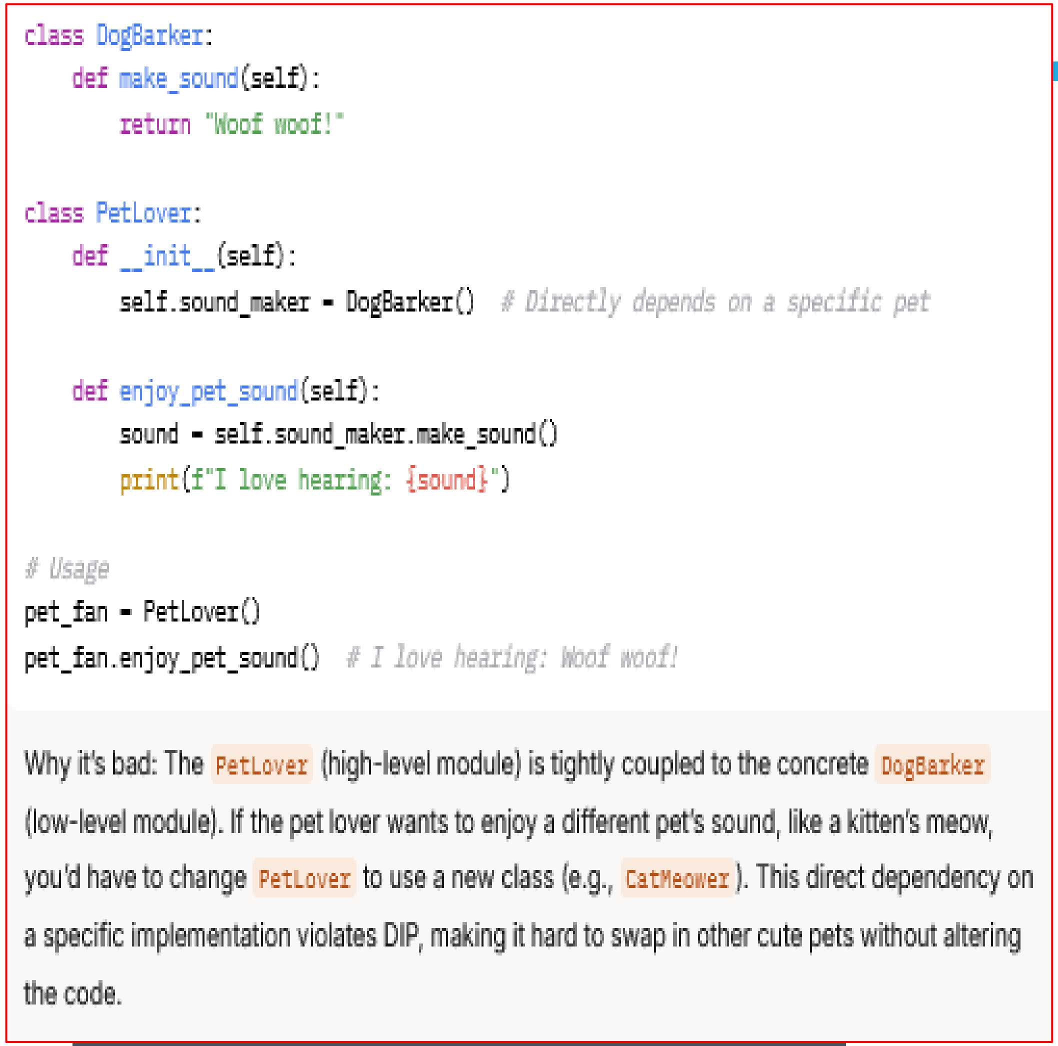
High-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

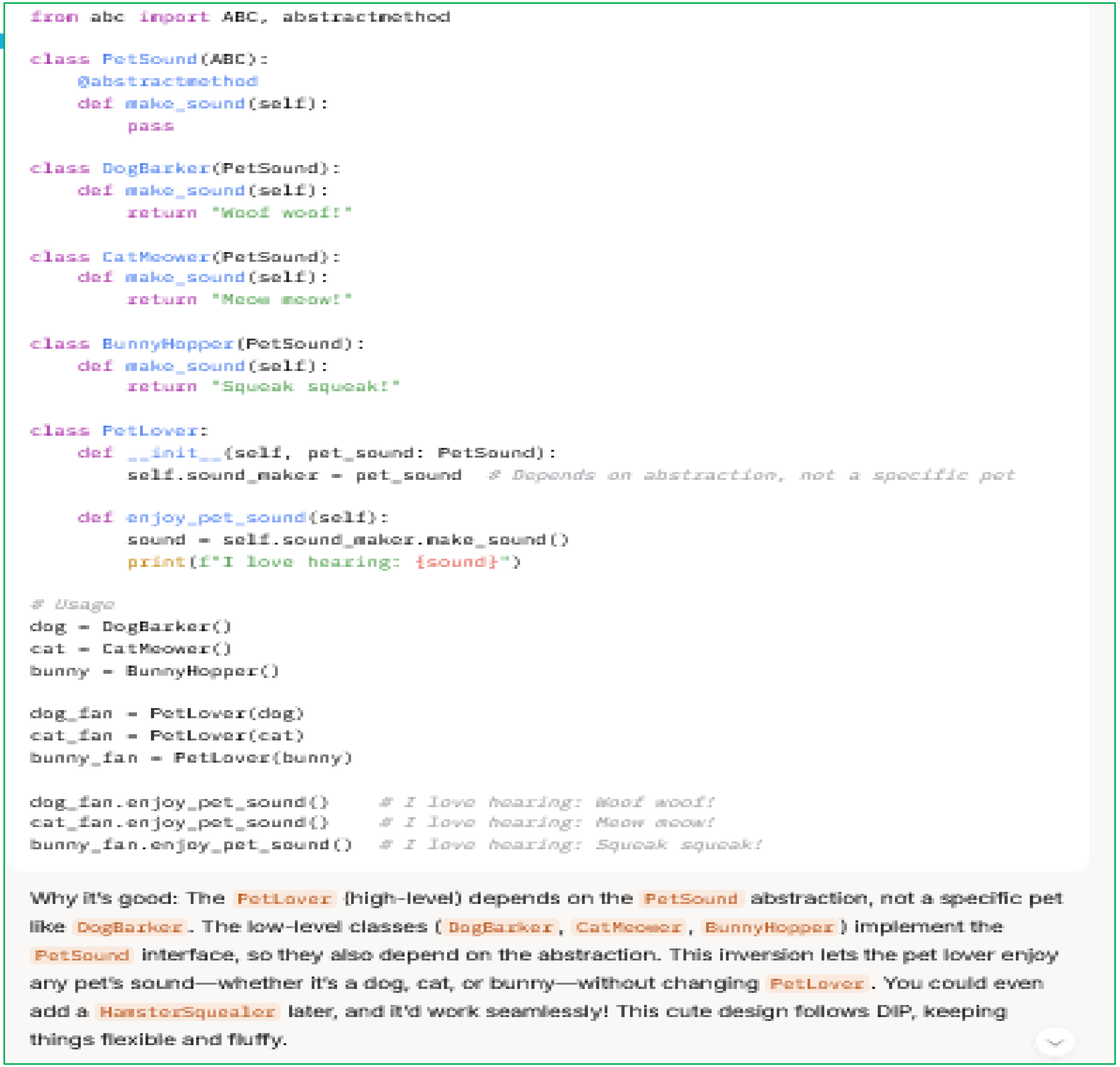


https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/



```python
class DogBarker:
    def make_sound(self):
        return "Woof woof!"

class PetLover:
    def __init__(self):
        self.sound_maker = DogBarker()  # Directly depends on a specific pet

    def enjoy_pet_sound(self):
        sound = self.sound_maker.make_sound()
        print(f"I love hearing: {sound}")

# Usage
pet_fan = PetLover()
pet_fan.enjoy_pet_sound()  # I love hearing: Woof woof!
```

Why it's bad: The `PetLover` (high-level module) is tightly coupled to the concrete `DogBarker` (low-level module). If the pet lover wants to enjoy a different pet's sound, like a kitten's meow, you'd have to change `PetLover` to use a new class (e.g., `CatMeower`). This direct dependency on a specific implementation violates DIP, making it hard to swap in other cute pets without altering the code.



```python
from abc import ABC, abstractmethod

class PetSound(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class DogBarker(PetSound):
    def make_sound(self):
        return "Woof woof!"

class CatMeower(PetSound):
    def make_sound(self):
        return "Meow meow!"

class BunnyHopper(PetSound):
    def make_sound(self):
        return "Squeak squeak!"

class PetLover:
    def __init__(self, pet_sound: PetSound):
        self.sound_maker = pet_sound  # Depends on abstraction, not a specific pet

    def enjoy_pet_sound(self):
        sound = self.sound_maker.make_sound()
        print(f"I love hearing: {sound}")

# Usage
dog = DogBarker()
cat = CatMeower()
bunny = BunnyHopper()

dog_fan = PetLover(dog)
cat_fan = PetLover(cat)
bunny_fan = PetLover(bunny)

dog_fan.enjoy_pet_sound()    # I love hearing: Woof woof!
cat_fan.enjoy_pet_sound()    # I love hearing: Meow meow!
bunny_fan.enjoy_pet_sound()  # I love hearing: Squeak squeak!
```

Why it's good: The `PetLover` (high-level) depends on the `PetSound` abstraction, not a specific pet like `DogBarker`. The low-level classes (`DogBarker`, `CatMeower`, `BunnyHopper`) implement the `PetSound` interface, so they also depend on the abstraction. This inversion lets the pet lover enjoy any pet's sound—whether it's a dog, cat, or bunny—without changing `PetLover`. You could even add a `HamsterSquealer` later, and it'd work seamlessly! This cute design follows DIP, keeping things flexible and fluffy.

SOLID PRINCIPLES

✓ Anlaşılır
✓ Tekrar Kullanılabilir
✓ Esnek

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp