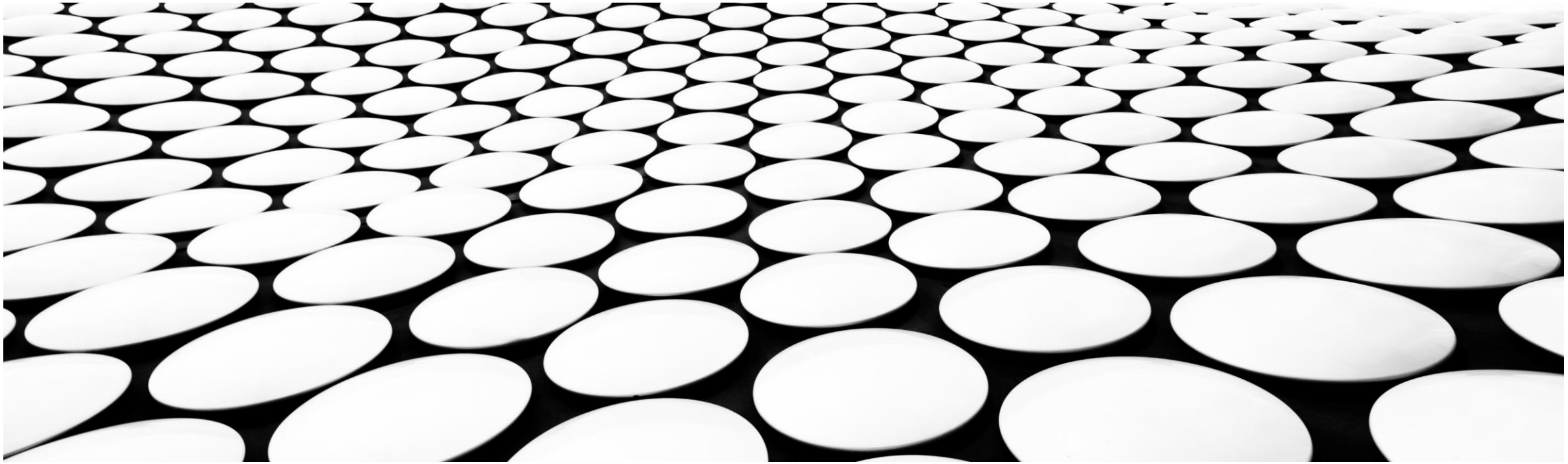
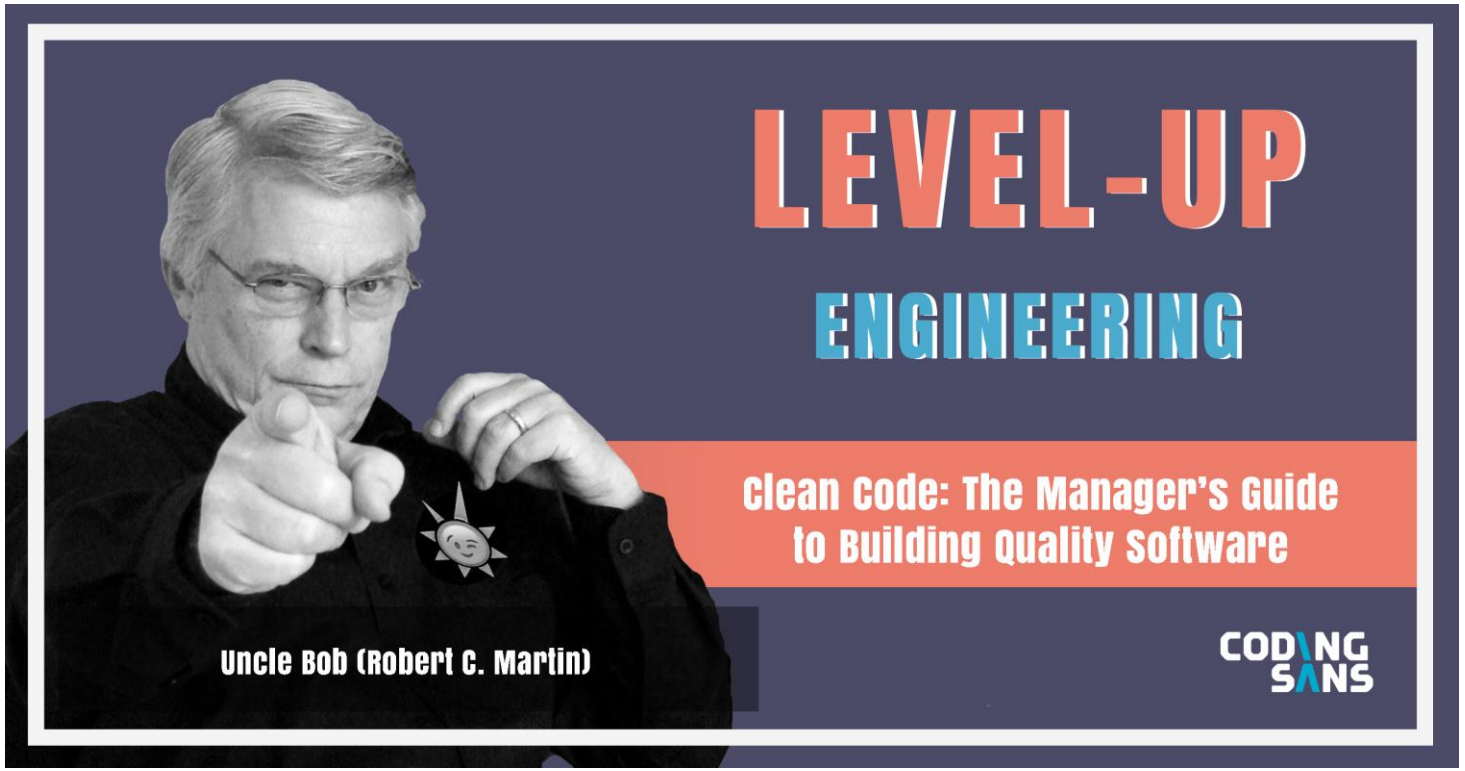
SOLID PRINCIPLES

OOP DESIGN



SOLID PRINCIPLES

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



https://cleancoders.com/

✓ Anlaşılır
✓ Tekrar Kullanılabilir
✓ Esnek

SINGLE RESPONSIBILITY PRINCIPLE

your code should have only one job to do

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

SINGLE RESPONSIBILITY PRINCIPLE

your code should have only one job to do

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();
}
```

OPEN/CLOSED PRINCIPLE

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

You shouldn't have to rewrite an existing class for implementing new features.

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.

```
01. public class Rectangle{
02.    public double Height {get;set;}
03.    public double Wight {get;set; }
04. }
```

- 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.

```
01. public class Rectangle{
02.    public double Height {get;set;}
03.    public double Wight {get;set; }
04. }
```

```
01. public class AreaCalculator {
02.    public double TotalArea(Rectangle[] arrRectangles)
03.    {
04.       double area;
05.       foreach(var objRectangle in arrRectangles)
06.       {
07.          area += objRectangle.Height * objRectangle.Width;
08.       }
09.       return area;
10.    }
11. }
```

- 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.

```
01. public class Rectangle{
02.    public double Height {get;set;}
03.    public double Wight {get;set; }
04. }
05. public class Circle{
06.    public double Radius {get;set;}
07. }
08. public class AreaCalculator
09. {
10.    public double TotalArea(object[] arrObjects)
11.    {
12.       double area = 0;
13.       Rectangle objRectangle;
14.       Circle objCircle;
15.       foreach(var obj in arrObjects)
16.       {
17.          if(obj is Rectangle)
18.          {
19.             area += obj.Height * obj.Width;
20.          }
21.          else
22.          {
23.             objCircle = (Circle)obj;
24.             area += objCircle.Radius * objCircle.Radius * Math.PI;
25.          }
26.       }
27.       return area;
28.    }
29. }
```

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.

```
01. public class Rectangle{
02.    public double Height {get;set;}
03.    public double Wight {get;set; }
04. }
05. public class Circle{
06.    public double Radius {get;set;}
07. }
08. public class AreaCalculator
09. {
10.    public double TotalArea(object[] arrObjects)
11.    {
12.       double area = 0;
13.       Rectangle objRectangle;
14.       Circle objCircle;
15.       foreach(var obj in arrObjects)
16.       {
17.          if(obj is Rectangle)
18.          {
19.             area += obj.Height * obj.Width;
20.          }
21.          else
22.          {
23.             objCircle = (Circle)obj;
24.             area += objCircle.Radius * objCircle.Radius * Math.PI;
25.          }
26.       }
27.       return area;
28.    }
29. }
```

```
01. public class AreaCalculator
02. {
03.    public double TotalArea(Shape[] arrShapes)
04.    {
05.       double area=0;
06.       foreach(var objShape in arrShapes)
07.       {
08.          area += objShape.Area();
09.       }
10.       return area;
11.    }
12. }
```

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.

```
01. public abstract class Shape
02. {
03.     public abstract double Area();
04. }
```

```
01. public class Rectangle: Shape
02. {
03.     public double Height {get;set;}
04.     public double Width {get;set;}
05.     public override double Area()
06.     {
07.         return Height * Width;
08.     }
09. }
10. public class Circle: Shape
11. {
12.     public double Radius {get;set;}
13.     public override double Area()
14.     {
15.         return Radius * Radus * Math.PI;
16.     }
17. }
```

```
01. public class AreaCalculator
02. {
03.     public double TotalArea(Shape[] arrShapes)
04.     {
05.         double area=0;
06.         foreach(var objShape in arrShapes)
07.         {
08.             area += objShape.Area();
09.         }
10.         return area;
11.     }
12. }
```

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.

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

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

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.

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 interface ICar
{
    void Drive();
}

public interface IAirplane
{
    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 multif
    }

    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 NotImplementedEx
    }

    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

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

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

```
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");
    }
}
```

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

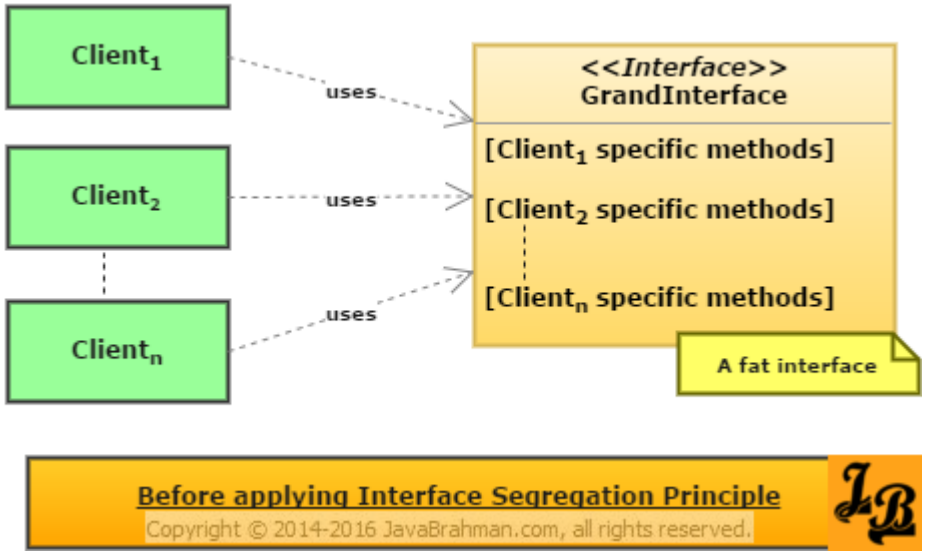
```
public class Airplane : IAirplane
{
    public void Fly()
    {
        //actions to fly a plane
        Console.WriteLine("Flying a plane");
    }
}
```

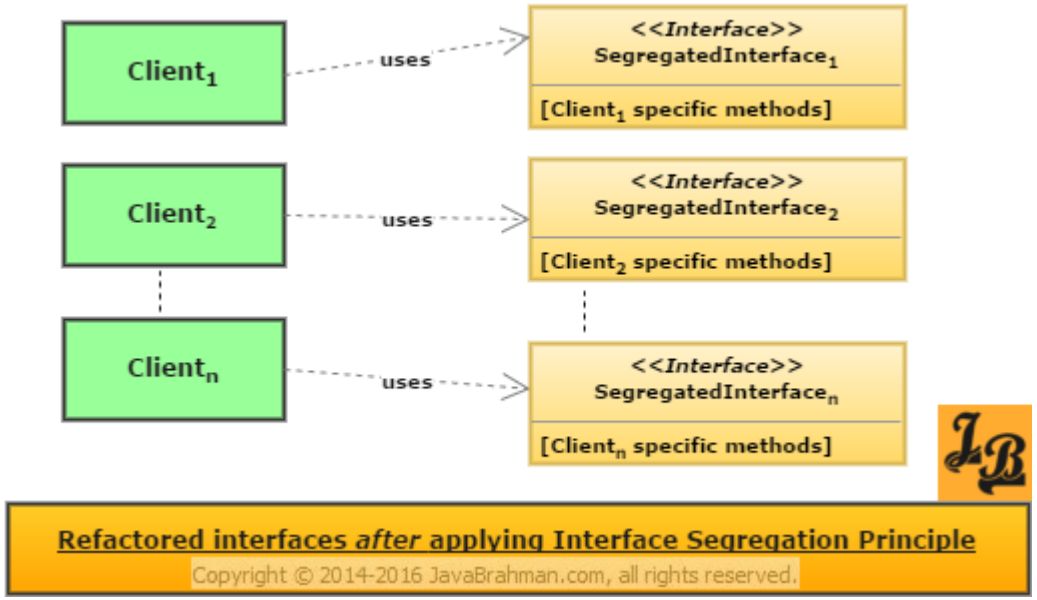
https://code-maze.com/interface-segregation-principle/

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.





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://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();
}
```

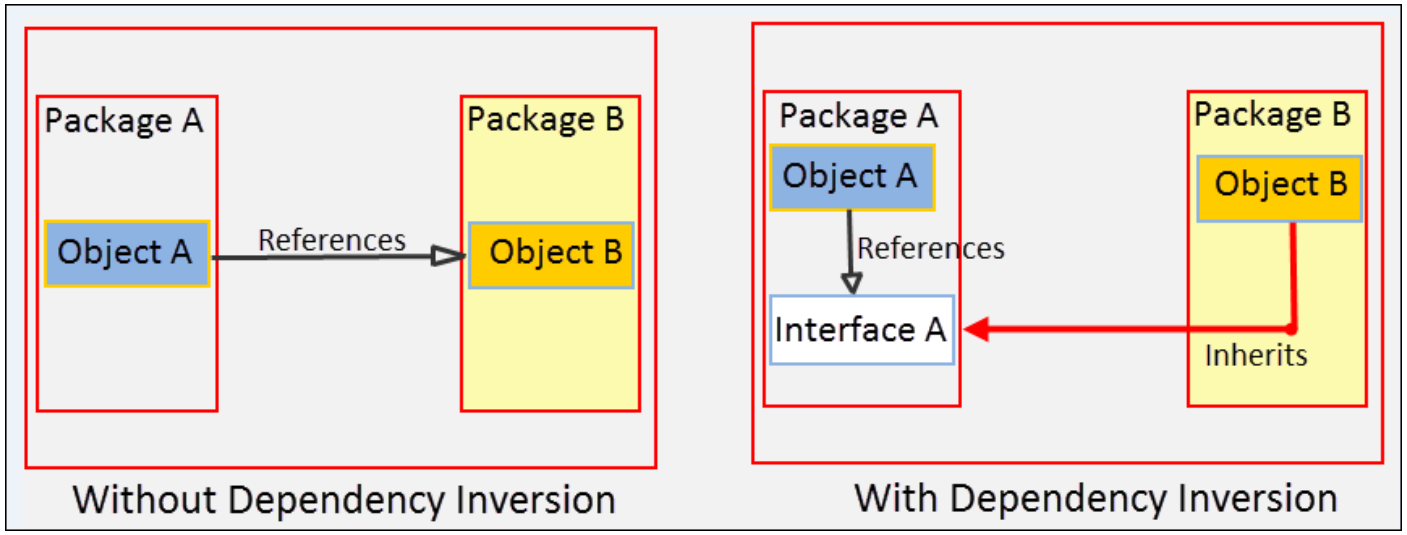
https://code-maze.com/dependency-inversion-principle/

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/

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