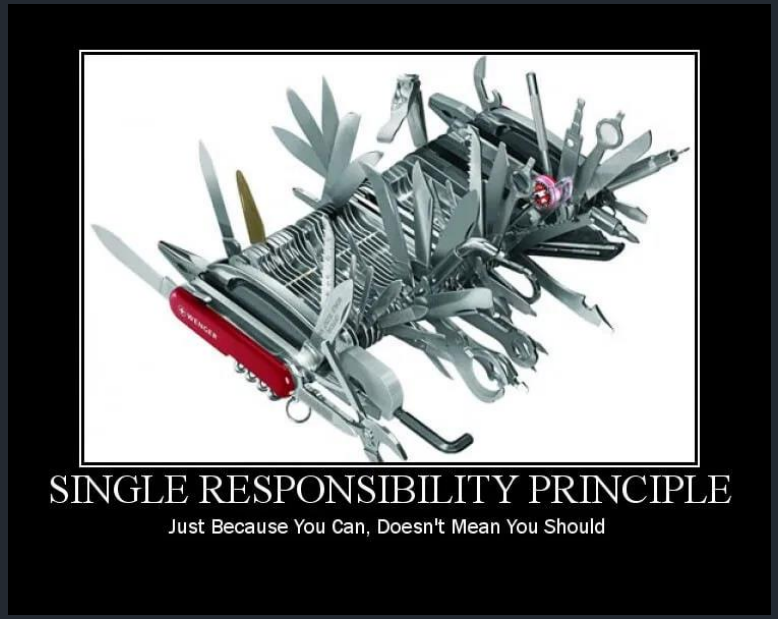
Principios SOLID

Principios SOLID

- Directrices para diseñar software orientado a objetos
- Facilitan la creación de sistemas más robustos, flexibles y mantenibles
- Reducción de complejidad, mejora en la mantenibilidad y escalabilidad del código
- Diseño más limpio y modular, facilitando la evolución y adaptación del software
- Promueven un enfoque de diseño que reduce el acoplamiento y mejora la cohesión
- Permiten una mayor facilidad para agregar nuevas funcionalidades y corregir errores sin afectar el sistema existente
- Ayudan a evitar problemas comunes como la rigidez del diseño y el acoplamiento excesivo entre componentes

Principios SOLID - **SRP**

- **S**OLID
  - **Single Responsibility Principle** (SRP)
    - Una sola responsabilidad



Ejemplo Single Responsibility Principle

- **Ejemplo SIN SRP**: UserCreator para crear a un usuario también valida su email Y lo guarda en la DB.

```
UserCreator userCreator = new UserCreator();
userCreator.CreateUser("Paco", "pacopaco@gmail.com", "53453ca");
```

```
public class UserCreator
{
    0 references
    public void CreateUser(string username, string email, string password)
    {
        // Validation logic
        if (!ValidateEmail(email))
        {
            throw new ArgumentException("Invalid email format.");
        }

        // Business rules
        // Database persistence
        SaveUserToDatabase(username, email, password);
    }
    1 reference
    private bool ValidateEmail(string email)
    {
        // Validation logic
    }
    1 reference
    private void SaveUserToDatabase(string username, string email, string password)
    {
        // Database persistence logic
    }
}
```

Ejemplo Single Responsibility Principle

- **Ejemplo CON SRP**: Cada clase se encarga de la lógica que indica su nombre.

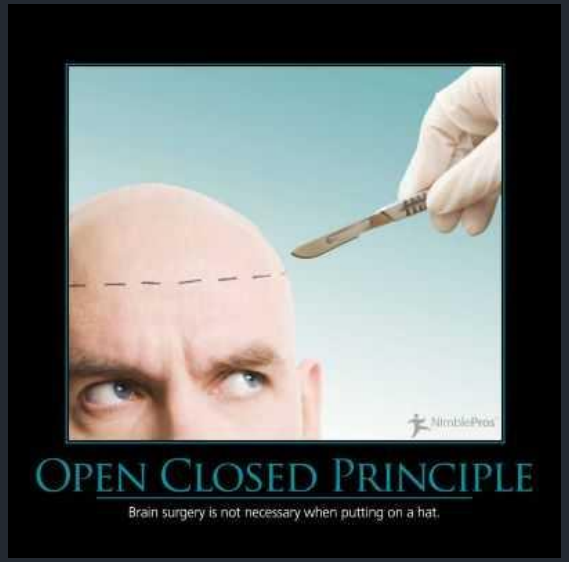
```
var email = "pacopaco@gmail.com";
if (EmailValidator.ValidateEmail(email))
{
    userCreator.CreateUser("Paco", email, "53453ca");
    var databaseSaver = new DatabaseSaver();
    databaseSaver.SaveUserToDatabase("Paco", email, "53453ca");
}
```

```
0 references
public class UserCreator
{
    // Email already comes verified from another class
    0 references
    public bool CreateUser(
        string username,
        string email,
        string password)
    {
        // Logic related to user creation and nothing else
        bool isUserCreated = false;

        // could return a boolean checking if user creation
        // was correct so another class saves it to Database
        // OR a user class
        return isUserCreated;
    }

}
```

```
// Static class since we don't need an instance
// to check string validation
1 reference
public static class EmailValidator
{
    1 reference
    public static bool ValidateEmail(string email)
    {
        // Check wether email was correct or not
        return email.Contains("@");
    }
}


1 reference
public class DatabaseSaver
{
    1 reference
    public void SaveUserToDatabase(string username,
        string email,
        string password)
    {
        // Database logic
    }
}
```

Principios de Diseño Software – SOLID - **OCP**

- SOLID
  - **Open/Close Principle** (OCP)
    - Abierto para su extensión, pero cerrado para su modificación



Ejemplo Open Closed Principle

- **Ejemplo SIN O**: La clase **AreaCalculator** depende directamente de Rectangle para calcular su área Y no puede calcular áreas de figuras otro tipo.

```
public class AreaCalculator
{
    0 references
    public float TotalArea(Rectangle[] rectangles)
    {
        float area = 0;
        foreach (var objRectangle in rectangles)
        {
            area += objRectangle.Height * objRectangle.Width;
        }
        return area;
    }
}
```

```
public class Rectangle
{
    1 reference
    public float Height { get; set; }

    1 reference
    public float Width { get; set; }
}
```

Ejemplo Open Closed Principle

- **Primer paso**: Generalizar AreaCalculator para poder aceptar otras figuras geométricas.
- Sigue siendo necesario modificar TotalArea cada vez que añadimos una nueva figura geométrica. No es lo que queremos.

```
public float TotalArea(object[] arrObjects)
{
    float area = 0;
    Rectangle objRectangle;
    Circle objCircle;
    foreach (var obj in arrObjects)
    {
        if (obj is Rectangle rectangle)
        {
            area += rectangle.Height * rectangle.Width;
        }
        else
        {
            objCircle = (Circle)obj;
            area += objCircle.Radius * objCircle.Radius * (float)Math.PI;
        }
        // else if(...
    }
    return area;
}
```

Ejemplo Open Closed Principle

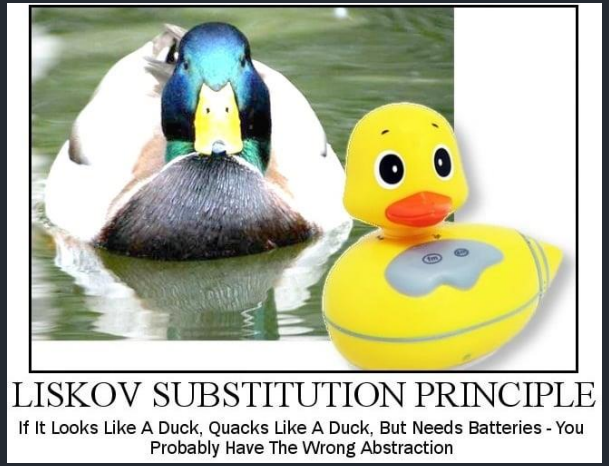
- Ejemplo CON O: Delegar en cada figura el cálculo de su área.
- AreaCalculator está abierta a aceptar nuevas figuras geométricas sin tener que modificar el código base.

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

```
1 reference
public class Rectangle : Shape
{
    1 reference
    public float Height { get; set; }

    1 reference
    public float Width { get; set; }

    2 references
    public override float Area()
    {
        return Height * Width;
    }
}
```

Principios de Diseño Software – SOLID - **LSP**

- SOLID
  - **Liskov Substitution Principle** (LSP)
    - Una clase derivada no debe modificar el comportamiento de la clase base
    - Una clase derivada debe poder comportarse siempre como la clase base



Ejemplo Liskov Substitution Principle

- Hemos de crear un sistema de vuelo para pájaros.
- **Ejemplo SIN Liskov**: Si añadimos un pingüino como hijo de Bird, no podrá volar. La abstracción Bird no es suficiente.

```
List<Bird> birds = new List<Bird>()
{
    new Seagull(),
    new Penguin()
};

for (int i = 0; i < birds.Count; i++)
{
    birds[i].Fly();
}
```

```
4 references
public abstract class Bird
{
    3 references
    public abstract void Fly();
}

0 references
public class Seagull : Bird
{
    2 references
    public override void Fly()
    {
        Console.WriteLine("I flyyy in the air");
    }
}

0 references
public class Penguin : Bird
{
    2 references
    public override void Fly()
    {
        Console.WriteLine("I cannot fly you dummie");
        throw new NotImplementedException();
    }
}
```

Ejemplo Liskov Substitution Principle

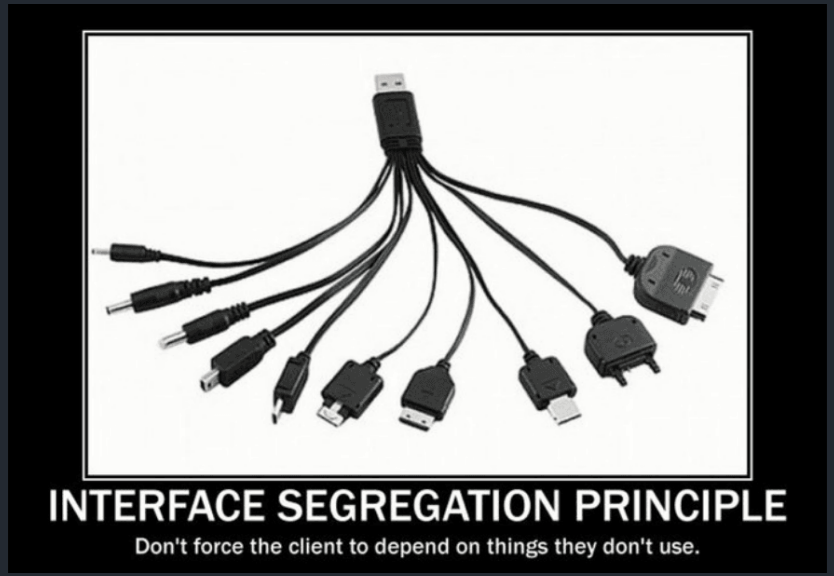
- **Ejemplo CON Liskov**: Abstraer en una capa intermedia los pájaros que puedan volar y no, e interactuaremos con la clase que englobe los requisitos (Pájaros voladores)

```
List<Bird> birds = new List<Bird>()
{
    new Seagull(),
    new Penguin()
};

for (int i = 0; i < birds.Count; i++)
{
    if (birds[i] is FlyingBird flyingBird)
    {
        flyingBird.Fly();
    }
    else if (birds[i] is WalkingBird walkingBird)
    {
        walkingBird.Walk();
    }
}
```

```
public abstract class Bird { }
1 reference
public abstract class WalkingBird : Bird
{
    1 reference
    public abstract void Walk();
}
1 reference
public abstract class FlyingBird : Bird
{
    1 reference
    public abstract void Fly();
}
1 reference
public class Seagull : FlyingBird
{
    1 reference
    public override void Fly()
    {
        Console.WriteLine("I flyyy in the air");
    }
}

1 reference
public class Penguin : WalkingBird
{
    1 reference
    public override void Walk()
    {
        Console.WriteLine("I walk, tap tap");
    }
}
```

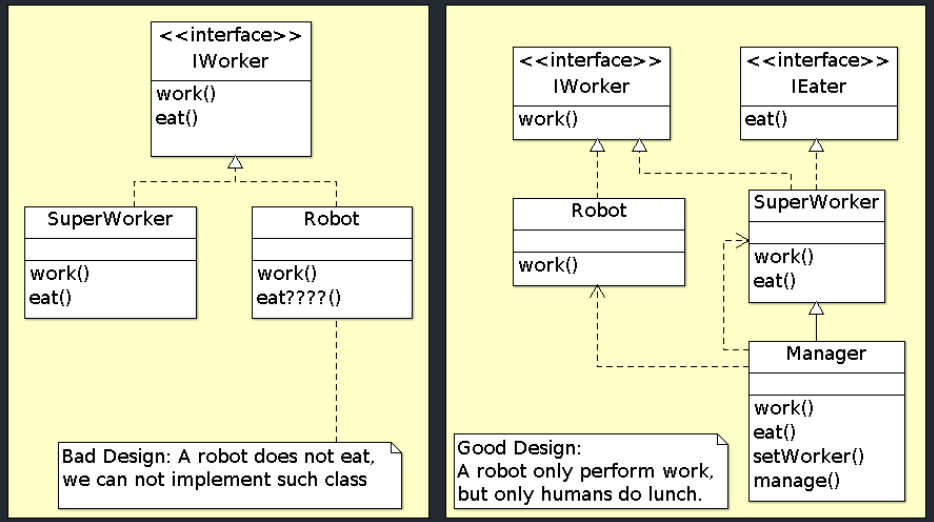
Principios de Diseño Software – SOLID - **ISP**

- SOLID
  - **Interface Segregation Principle** (ISP)
    - Una clase que implementa una interfaz no debe depender de métodos que no utiliza.



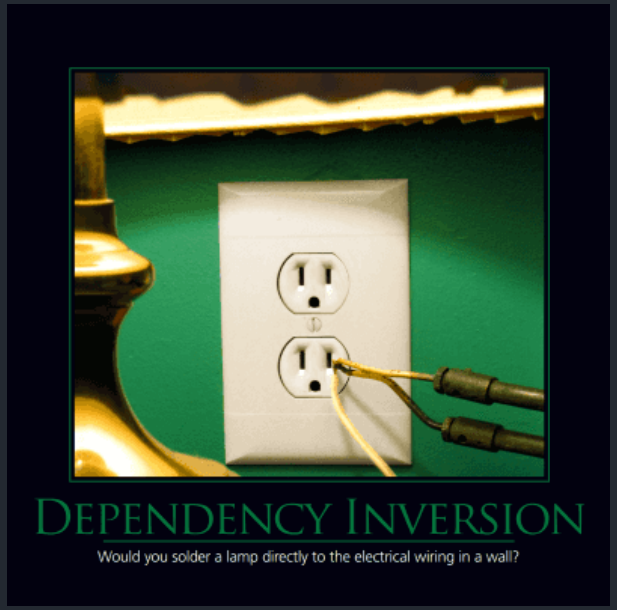
Ejemplo Interface Segregation Principle

- ¿Por qué debería tener que implementar Robot el método Eat() si no puede comer?
- Siguiendo ISP, IWorker dejará de implementar Eat() para ser otra interfaz quien lo realice. Robot Y SuperWorker implementan únicamente lo que necesitan.



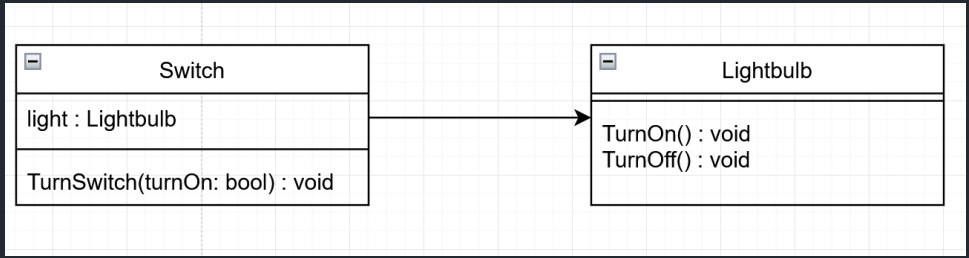
Principios de Diseño Software – SOLID - **DIP**

- SOLID
  - **Dependency Inversion Principle** (DIP)
    - Las clases de alto nivel no deben depender de clases de bajo nivel



Ejemplo Dependency Inversión Principle

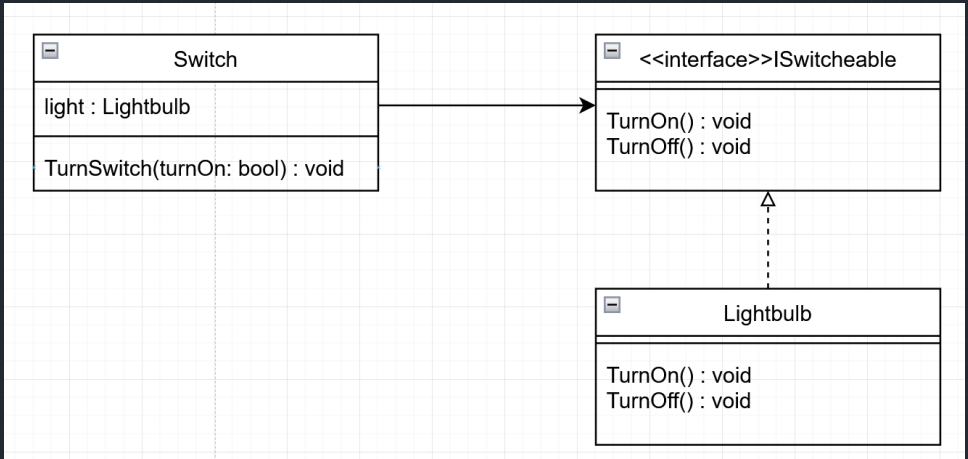
- **Ejemplo SIN DIP**: Dependencia directa entre Switch y Lightbulb. Si queremos añadir funcionalidad a Switch dependemos de los requisitos de Lightbulb.
- Podremos querer encender o apagar un motor con Switch pero actualmente tendría que heredar de Lightbulb.




```
1 reference
class Switch
{
    private Lightbulb light;

    0 references
    public Switch(Lightbulb light)
    {
        this.light = light;
    }

    0 references
    public void TurnSwitch(bool turnOn)
    {
        if(turnOn) light.TurnOn();
        else light.TurnOff();
    }
}

2 references
class Lightbulb
{
    1 reference
    public void TurnOn() { }
    1 reference
    public void TurnOff() { }
}
```

Ejemplo Dependency Inversión Principle

- **Ejemplo CON DIP**: La interfaz ISwitcheable separa la dependencia y nos permite añadir nuevos objetos switcheables al sistema.




```
class Switch
{
    private ISwitchable switchable;

    0 references
    public Switch(ISwitchable switchable)
    {
        this.switchable = switchable;
    }

    0 references
    public void TurnSwitch(bool turnOn)
    {
        if(turnOn) switchable.TurnOn();
        else switchable.TurnOff();
    }
}

3 references
interface ISwitchable
{
    2 references
    void TurnOn();
    2 references
    void TurnOff();
}
0 references
class Lightbulb : ISwitchable
{
    2 references
    public void TurnOn() { }
    2 references
    public void TurnOff() { }
}
```