**UCR | ECCI | CI-0156- Práctica Supervisada**
**Tarea - SOLID**
**Fabián Orozco Chaves — B95690**
**Octubre, 2023**

---

## Tabla de Contenidos



## Enunciado

Explique en que consiste los principios SOLID e implemente un proyecto con cada uno de los principios.

## Descripción: SOLID

SOLID es un acrónimo en inglés y representa las iniciales de 5 principios explícitos (y otros implícitos como encapsulamiento, separación de asuntos o DRY), seleccionados entre docenas por ser fundamentales para mantener la escalabilidad, mantenibilidad y comprensión de un diseño de software complejo en el paradigma de Programación Orientado a Objetos. Fueron presentados por Robert C. Martin a principios de la década de los 2000.

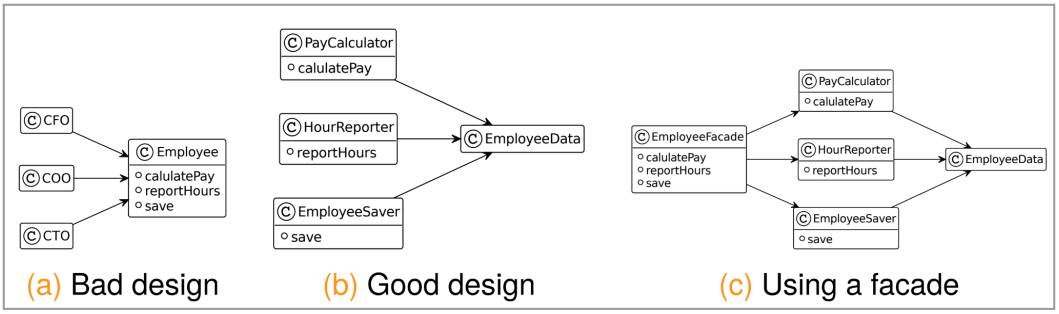
## Principios de diseño SOLID

A diferencia de los **patrones** de software, en los **principios** no hay una estructura, y tampoco están dirigidos a resolver problemas que se repiten (no tienen núcleo de la solución), sino que son reglas generales que nos permiten desarrollar software de calidad, adaptándonos al contexto del sistema.

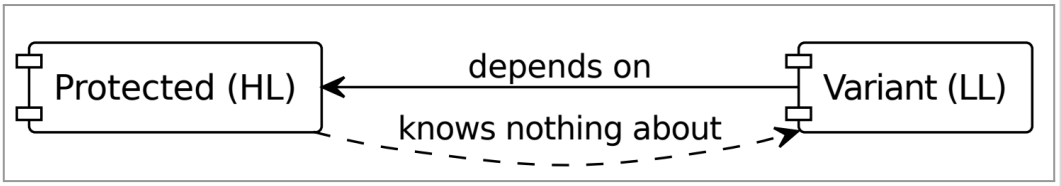
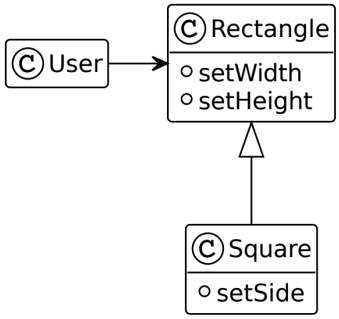
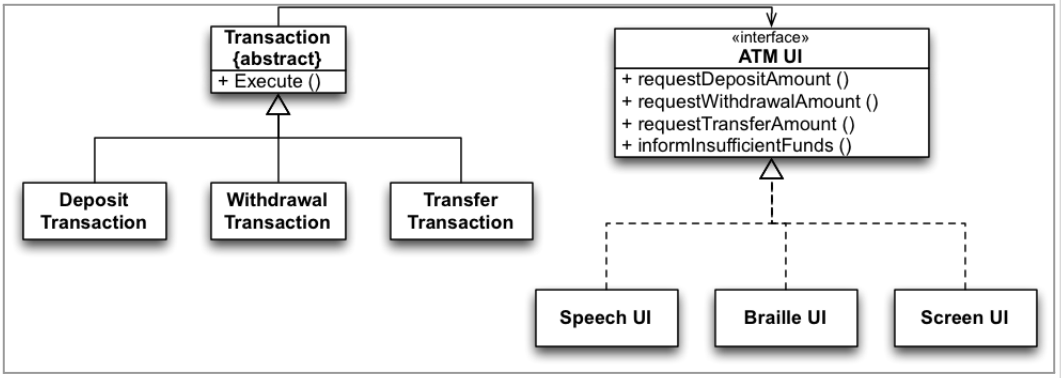
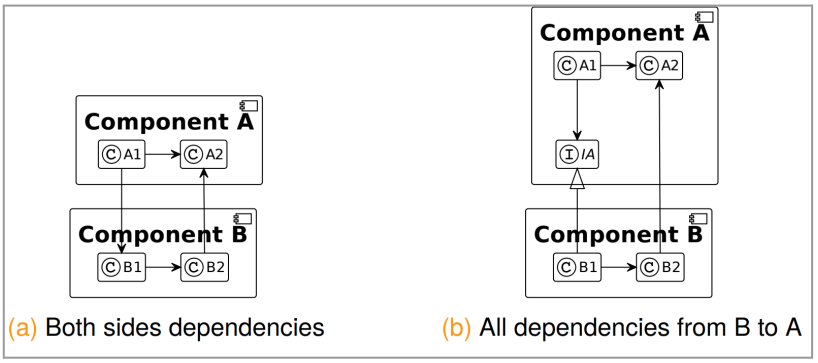
Permiten tener una primera aproximación al diseño y decidir si el diseño es bueno o malo antes de implementar (codificar). En base a esto se puede refinar.

Si no se implementan se genera una deuda técnica con costos muy grandes (muy difícil).

| | |
|---|---|
| **1) Single Responsibility Principle (SRP)** | Cada componente realiza una única tarea específica, es decir, tiene una única responsabilidad. Es medible por la interacción entre variables y métodos. |




(a) Bad design (b) Good design (c) Using a facade

| | |
|---|---|
| **2) Open Close Principle (OCP)** | El software es extensible sin modificar código existente, fomentando adaptabilidad. Los componentes permiten extenderse en un futuro pero no cambian las interfaces públicas. (Ej: Herencia(abstracta-concreta) y extensión (interface).<br>  |
| **3) Liskov Substitution Principle (LSP)** | Las subclases pueden reemplazar a sus clases base sin problemas. Representa un código independiente donde los objetos son sustituibles unos por otros porque utilizan una interfaz común.<br>  <br>This illustrates the square/rectangle problem, which violates the principle, as `Square` and `Rectangle` interfaces and behaviours are not compatible. It forces to add extra mechanisms to distinguish each types during runtime, which lowers the software maintainability. |
| **4) Interface Segregation Principle (ISP)** | Interfaces específicas para cada cliente. Ej: utilizar el mismo grafo de diferentes formas para representar distintos tableros en un juego de ajedrez.<br>  |
| **5) Dependency inversion Principle (DIP)** | Los módulos deben depender de abstracciones en lugar de detalles concretos. Los objetos son independientes de otros (Ej: usando herencia o plantillas) de todos los tipos de sus partes.<br>  |
| **Otro) Don't Repeat Yourself (DRY)** | No duplicar código, para evitar tener distintos puntos de modificación y errores. |

## Ejemplo - VendingMachine

(adjunto zip)

**SRP**

Se cumple porque “VendingMachine” mediante inyección de dependencias por constructor no conoce sobre la instancia o la función “JsonMenuLoader” ni de “PriceTagMakers”.⬇️

```
static void Main(string[] args)
{
    Console.WriteLine("hello");
    var menuLoader = new JsonMenuLoader();
    var vendingMachine = new VendingMachine(menuLoader);
    vendingMachine.Start();

}
```

program.cs

```
public VendingMachine(JsonMenuLoader menuLoader)
{
    this._menuLoader = menuLoader;
}
```

VendingMachine.cs

Inyección de dependencias por constructor.

```
1 referencia
public void DisplayMenu()
{
    // read menu
    var products = _menuLoader.LoadMenu();
```

VendingMachine.cs

Llamado de otra responsabilidad

```
using System.Text.Json;

3 referencias
public class JsonMenuLoader
{
    1 referencia
    public List<Product> LoadMenu()
    {
        string text = File.ReadAllText(@"./menu.json");
        return JsonSerializer.Deserialize<List<Product>>(text);
    }
}
```

JsonMenuLoader.cs

Responsable de cargar los productos.

**OCP**

Si necesitamos poner un extra al precio de ciertos productos dependiendo de su tipo, solo creamos una clase que derive de “PriceTagMaker”, sin tener que modificar el código existente.⬇️

```
public class PriceTagMaker
{
    5 referencias
    public virtual string MakePriceTagForProduct(Product product)
    {
        return $"${product.Price:0.##}";
    }
}
```

PriceTagMaker.cs

Superclase (usa virtual)

```
public class DrinkTypePriceTagMaker : PriceTagMaker
{
    4 referencias
    public override string MakePriceTagForProduct(Product product)
    {
        var basePrice = base.MakePriceTagForProduct(product);
        return $"{basePrice} + ${product.Price * 0.1m:0.##} deposit
        fee";
    }
}
```

PriceTagMaker.cs

Subclase derivada (sobrecarga)

```
public class FruitTypePriceTagMaker : PriceTagMaker
{
    4 referencias
    public override string MakePriceTagForProduct(Product product)
    {
        var basePrice = base.MakePriceTagForProduct(product);
        return $"{basePrice} + ${product.Price * 0.05m:0.##} wrapping
        fee";
    }
}
```

PriceTagMaker.cs

Subclase derivada (sobrecarga)

**LSP**

“Extensions” hace que retorne una superclase (tipo de productos normales) o una subclase (para tipos de productos fruta o bebida) y con esa clase calcula el precio. Sin embargo el ToString() del producto sigue sin saber que por debajo el subtipo utilizado.⬇️

```
public static PriceTagMaker GetPriceTagMaker(this Product product)
{
    switch (product.Type)
    {
        case "Drink":
            return new DrinkTypePriceTagMaker();
        case "Fruit":
            return new FruitTypePriceTagMaker();
        default:
            return new PriceTagMaker();
    }
}
```

Extensions.cs

Retorna una instancia dependiendo el tipo de producto.

```
public override string ToString()
{
    return $"ID: {ID} - {Name} - {this.GetPriceTagMaker().MakePriceTagForProduct(this)} ";
}
```

Product.cs

No conoce sobre subclases, cualquier subclase puede sustituir a la superclase.

**DIP**

Ahora no se tiene que modificar la clase VendingMachine con cada lector de archivos que queramos utilizar (como por API), es decir, los modulos de alto nivel no dependen de los de bajo nivel, sino de una abstracción (IMenuLoader).⬇️

```
private IMenuLoader _menuLoader;

1 referencia
public VendingMachine(IMenuLoader menuLoader)
{
    this._menuLoader = menuLoader;
}
```

VendingMachine.cs

Utiliza Interface como inyección (independiente de si es json o csv)

```
public interface IMenuLoader
{
    3 referencias
    List<Product> LoadMenu();
}
```

IMenuLoader.cs

Interface con método LoadMenu()

```
public class CsvMenuLoader : IMenuLoader
{
    2 referencias
    public List<Product> LoadMenu()
    {
        var list = new List<Product>();

        using (var reader = new StreamReader(@"./menu.csv"))
        {
            while (!reader.EndOfStream)
            {
                var line = reader.ReadLine();
                var values = line.Split(',');
                list.Add(new Product { ID = int.Parse(values[0]), Name = values[1],
                Price = decimal.Parse(values[2]), Type = values[3] });
            }
        }

        return list;
    }
}
```

CsvMenuLoader.cs

Implementa método LoadMenu() de interface IMenuLoader

```
public class JsonMenuLoader : IMenuLoader
{
    2 referencias
    public List<Product> LoadMenu()
    {
        string text = File.ReadAllText(@"./menu.json");
        return JsonSerializer.Deserialize<List<Product>>(text);
    }
}
```

JsonMenuLoader.cs

Implementa método LoadMenu() de interface IMenuLoader

```
static void Main(string[] args)
{
    Console.WriteLine("hello");
    var menuLoader = new CsvMenuLoader();
    var vendingMachine = new VendingMachine(menuLoader);
    vendingMachine.Start();
}
```

Program.cs
Se envía la instancia del lector deseada.

**ISP**

Permite tener interfaces separadas (rápida o amigable) dependiendo del usuario.
En el futuro se puede reutilizar incluso “ICanGreet” o “ICanTellTime” aunque ni siquiera sea de una máquina expendedora.⬇️

```
public interface ICanGreet
{
    0 referencias
    void SayHello();
    0 referencias
    void SayBye();
}

1 referencia
public interface ICanTellTime
{
    0 referencias
    void TellTime();
}

3 referencias
public interface IVendingMachine
{
    4 referencias
    void DisplayMenu();
    3 referencias
    void Start();
}

0 referencias
public interface IFriendlyVendingMachine : ICanGreet, ICanTellTime, IVendingMachine { }
```

IVendingMachine.cs

Interface de VendingMachine para poder tener muchos tipos de VendingMachine que implementen los métodos a su manera.
Es como SRP pero hecho con interfaces.

```
public class HighEfficiencyVendingMachine : IVendingMachine
{
    2 referencias
    private IMenuLoader _menuLoader;

    0 referencias
    public HighEfficiencyVendingMachine(IMenuLoader menuLoader)
    {
        this._menuLoader = menuLoader;
    }


    2 referencias
    public void DisplayMenu()
    {
        // read menu
        var products = _menuLoader.LoadMenu();

        // show menu
        foreach (var product in products)
        {
            Console.WriteLine(product.ToString());
        }
    }

    1 referencia
    public void Start()
    {
        DisplayMenu();
    }
}
```

HighEfficiencyVendingMachine.cs

Implementa Interface pero sin los métodos “SayHello”, “Tellme” y “SayBye”.

**Salidas del proyecto**

```
namespace VendingMachineProject
{
    0 referencias
    class Program
    {
        0 referencias
        static void Main(string[] args)
        {
            Console.WriteLine("\nVending Machine from JSON products:");
            var menuLoader = new JsonMenuLoader();
            var vendingMachine = new VendingMachine(menuLoader);
            vendingMachine.Start();

            Console.WriteLine("\n\nVending Machine from CSV products:");
            var menuLoader2 = new CsvMenuLoader();
            var vendingMachine2 = new VendingMachine(menuLoader2);
            vendingMachine2.Start();
            Console.WriteLine("\n");

            Console.WriteLine("\n\nHighEfficiencyVendingMachine from JSON products:");
            var menuLoader3 = new JsonMenuLoader();
            var vendingMachine3 = new HighEfficiencyVendingMachine(menuLoader3);
            vendingMachine3.Start();
            Console.WriteLine("\n");

        }
    }
}
```

```
Vending Machine from JSON products:
Please enter your name
Fabián
Hello Dear Fabián! How are you?

Current time is 30/9/2023 at 15:20!

ID: 1 - Apple - $1 + $0.05 wrapping fee
ID: 2 - Banana - $2 + $0.1 wrapping fee
ID: 3 - Coke - $3 + $0.3 deposit fee
ID: 4 - Donut - $4
ID: 5 - Eggnog - $5 + $0.5 deposit fee
Press any key to exit...
```

Vending machine + Json

```
Vending Machine from CSV products:
Please enter your name
Fabián
Hello Dear Fabián! How are you?

Current time is 30/9/2023 at 15:20!

ID: 1 - Apple Cider - $1.5 + $0.15 deposit fee
ID: 2 - Brazil Coffee - $2.5 + $0.25 deposit fee
ID: 3 - Canada Dry - $3.5 + $0.35 deposit fee
ID: 4 - Doctor Pepper - $4.5 + $0.45 deposit fee
Press any key to exit...
```

Vending machine + Csv

```
HighEfficiencyVendingMachine from JSON products:
ID: 1 - Apple Cider - $1.5 + $0.15 deposit fee
ID: 2 - Brazil Coffee - $2.5 + $0.25 deposit fee
ID: 3 - Canada Dry - $3.5 + $0.35 deposit fee
ID: 4 - Doctor Pepper - $4.5 + $0.45 deposit fee
```

HighEfficiencyVendingMachine + json

## Referencias Bibliográficas


1. **(SOLID | resumen)** Clean Architecture and principles
2. **(SOLID | Ejemplos)** Nesteruk, D. (2021). Design patterns in modern C++20: Reusable Approaches for Object-Oriented Software Design. Apress.
3. **(Principios y patrones)** Material de curso (2022) CI0136 - Diseño de Software. Prof. Alan Calderón Castro.
4. **(namespaces C#)** How to call method from one namespace to another and calculate the final result.
5. **(Ejemplo)** Learn SOLID Principles by Building a Simple App