# SOLID principles

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

## S: Single responsibility principle

*Todas las clases cumplen con una única responsabilidad*

* **Program.cs**. Arranque de la aplicación y creación del **Service Container** (aspectos puramente técnicos)
* **Application**. Aplicación de consola (interactuación del usuario con la consola)
* Clases de servicio cumpliendo su única responsabilidad: **DirectoryService.cs** (Gestión de directorios), **FileService.cs** (Gestión de ficheros), **TopService.cs** (Calculo de lista Top) **y CacheService.cs** (Memoria Caché)
* Clases de tests, para cada uno de los servicios

## O: Open/closed principle

*Abierto a añadir funcionalidades, cerrado a cambios (Open for extension, Closed for modification)*

* El método **Setup**() debe ser implementado por **DirectoryService** y **FileService**, pero cada uno tiene un Setup particular (son casi iguales, pero hay alguna diferencia).

Nota: Como hay una parte común en la implementación del método **Setup**(), esta se ha llevado a una clase base llamada **BaseService**, de la que heredan **DirectoryService** y **FileService**. Estas dos últimas pueden sobreescribirla si fuera necesario modificar el Setup de la clase base (Solo es necesario en **FileService**)

// No cumple Open/closed

BaseTest.cs: …

public void Setup()

{

if (this.IsInstanceOfType(DirectoryBase)) {...};

if (this.IsInstanceOfType(FileBase)) {...};

}

## L: Liskov substitution principle

*if S is a subtype of T, then objects of type T should be replaced with the objects of type S*

* También es aplicable al método Setup(), pues se puede sustituir la clase FileServiceTest por la clase de la que extiende BaseTest:

// Usar clase hijo

FileServiceTest test = new FileServiceTest();

test.Setup();

// Usar clase padre

BaseTest test = new FileServiceTest();

test.Setup();

Ejemplo clásico: Coche, Renault, Mercedes. Se usa la clase padre Coche y el método numAsientos(), que es sobre-escrito en cada subclase (Renault o Mercedes) en lugar de usar estas subclases.

**// Código incorrecto (no cumple el principio de sustitución de Liskov)**

**abstract class Coche**

**{**

**public int numAsientosRenault() { return 5 };**

**public int numAsientosAudi() { return 4 };**

**}**

**class Renault extends Coche {}**

**class Audi extends Coche {}**

**public static void imprimirNumAsientos(Coche[] arrayCoches)**

**{**

**for (Coche coche : arrayCoches)**

**{**

**if(coche instanceof Renault){ System.out.println(numAsientosRenault(coche))};**

**if(coche instanceof Audi){ System.out.println(numAsientosAudi(coche))};**

**}**

**}**

**// Código correcto (cumple el principio de sustitución de Liskov)**

**abstract class Coche**

**{**

**virtual public int numAsientos();**

**}**

**class Renault extends Coche**

**{**

**override public int numAsientos() { return 5 };**

**}**

**class Audi extends Coche**

**{**

**override public int numAsientos() { return 4 };**

**}**

**public static void imprimirNumAsientos(Coche[] arrayCoches)**

**{**

**for (Coche coche : arrayCoches)**

**System.out.println(coche.numAsientos());**

**}**

Otro ejemplo de Liskov sustitution principle:

<https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-liskov-substitution-principle/>

## I: Interface segregation principle

*Evitar “El coche volador”*

*Clients should not be forced to implement interfaces they don't use. Instead of one huge interface, many small interfaces are preferred based on groups of methods.*

* **FileServiceTest** y **DirectoryServiceTest** heredan de **BaseTest**, que implementa **ISetupTest** y **ITearDownTest**
* En cambio, **TopServiceTest** y **CacheServiceTest** implementan unicamente **ISetupTest**
* Si solo existiera un interfaz IBaseTest (con los metodos Setup() y TearDown()), y todos los tests implementaran este interface, entonces TopServiceTest debería implementar el método TearDown(), cuando no lo necesita

## D: Dependency inversion principle

*Se trabaja con la abstracción (interfaces), no con la implementación.*

* **Application** usa las interfaces **IDirectoryService**, **IFileService y ITopService**
* **TopService** usa la interfaz **IFileService**
* **FileService** usa **CacheService**

NOTA: Diferenciar **'inversion de dependencias**' con **'inyección de dependencias'**

Para la inyección de dependencias se ha usado el propio DI de Microsoft:

// Service container (Dependecy injection)

// Es necesario registrar la clase Application consigo misma, para incluirla en el Service

// Container y poder obtenerla desde el builder para ejecutarla

var builder = new ServiceCollection()

.AddSingleton<Application>()

.AddSingleton<IDirectoryService, DirectoryService>()

.AddSingleton<IFileService, FileService>()

.AddSingleton<ITopService, TopService>()

.AddSingleton<ICacheService, CacheService>()

.BuildServiceProvider();

// Obtención del objeto de tipo Application desde el Service Container

Application console = builder.GetRequiredService<Application>();

console.Run(args[0].Trim());

# NUNIT

* **Setup vs OneTimeSetup**. El primero se ejecuta una vez por método, mientras que el segundo se ejecuta una vez por clase (si esta clase tiene 3 tests, el método anotado con OneTimeSetup solo se ejecuta una vez)
* **TearDown vs OneTimeTearDown**. Idem. al caso anterior

# RESUMEN

IDE: Visual Studio 2022

Language: C#

Proyect type: ConsoleApp NET 6

Patterns: DI, IoC (using Interfaces), SOLID principles

Unit tests: JUnit and Moq

Repository: Git and GitHub as remote repository, included README.md