UT5_TA1 - Ignacio Villarreal, Bruno Albín, Santiago Aurrecochea, Joaquin Gasco, José Varela y Gonzalo Paz.

# Patrones de diseño

Para cada uno de los siguientes ejercicios, en equipo:

1- Determine qué principio SOLID se está violando, agregando una justificación.

2- Agregue las clases, interfaces, métodos que considere necesarios para remediar la situación.

## Ejercicio 1

```
public abstract class Animal {
    public abstract void Comer();
    public abstract void Volar();
}

public class Perro : Animal {
    public override void Comer() {
        // El perro come
    }

    public override void Volar() {
        throw new NotImplementedException();
    }
}
```

El principio de segregación de interfaces (ISP) establece que una clase no debería estar obligada a implementar interfaces que no usa. En este caso, la clase `Animal` tiene un método `Volar` que no es aplicable a todos los animales. Al forzar a `Perro` a implementar `Volar`, estamos violando este principio porque `Perro` no vuela. La solución es dividir la interfaz para que cada clase solo implemente lo que realmente necesita.

```
public abstract class Animal {
    public abstract void Comer();
}

public abstract class AnimalVolador: Animal {
    public abstract void Volar();
}

public class Perro : Animal {
    public override void Comer() {
```

```
        // El perro come
    }
}

public class Pajaro: AnimalVolador {
    public override void Comer() {
        // El pájaro come
    }

    public override void Volar() {
        // El pájaro vuela
    }
}
```

Separamos las responsabilidades creando una nueva clase abstracta `AnimalVolador` que extiende de `Animal` e incluye el método `Volar`.

`Perro` ahora solo extiende de `Animal`, evitando la necesidad de implementar el método `Volar`.

`Pajaro` extiende de `AnimalVolador`, implementando ambos métodos `Comer` y `Volar`.

# Ejercicio 2

```
public class Documento {
    public string Contenido { get; set; }
}

public class Impresora {
    public void Imprimir(Documento documento) {
        Console.WriteLine(documento.Contenido);
    }

    public void Escanear(Documento documento) {
        // Código complejo para escaneo...
    }
}
```

El principio de responsabilidad única (SRP) establece que una clase debería tener una única responsabilidad. En el caso de la clase `Impresora`, se están manejando dos responsabilidades: imprimir y escanear. Esto va en contra del SRP. Para cumplir con este principio, deberíamos separar estas responsabilidades en clases distintas.

```
public class Documento {
    public string Contenido { get; set; }
}

public class Impresora {
    public void Imprimir(Documento documento) {
        Console.WriteLine(documento.Contenido);
    }
}

public class Escaner {
      public void Escanear(Documento documento) {
            // Código para escaneo...
      }
}

public class ImpresoraEscaner {

      private Impresora impresora = new Impresora();
      private Escaner escaner = new Escaner();

      public void Imprimir(Documento documento) {
            impresora.Imprimir(documento);
      }

      public void Escanear(Documento documento) {
            escaner.Escanear(documento);
      }
}
```

Se crean dos clases separadas: `Impresora` y `Escaner`, cada una con su propia responsabilidad.

La clase `Impresora` contiene el método `Imprimir`.

La clase `Escaner` contiene el método `Escanear`.

Si se necesita una clase que combine ambas funcionalidades, se crea una clase `ImpresoraEscaner` que utiliza instancias de `Impresora` y `Escaner` para delegar las responsabilidades adecuadamente.

# Ejercicio 3

```
public class BaseDeDatos {
	public void Guardar(Object objeto) {
		// Guarda el objeto en la base de datos
	}

	public void EnviarCorreo(string correo, string mensaje) {
		// Envía un correo electrónico
	}
}
```

El principio de responsabilidad única (SRP) establece que una clase debería tener una única responsabilidad. En este caso, la clase `BaseDeDatos` está manejando tanto la manipulación de datos como el envío de correos electrónicos, lo cual viola el SRP. Para cumplir con este principio, deberíamos separar estas responsabilidades en clases distintas.

```
public class BaseDeDatos {
	public void Guardar(Object objeto) {
		// Guarda el objeto en la base de datos
	}

public class ServicioCorreo{
	public void EnviarCorreo(string correo, string mensaje) {
		// Envía un correo electrónico
	}
}
```

Se crea una clase `BaseDeDatos` que solo contiene el método `Guardar`, dedicado exclusivamente a la manipulación de datos.

Se crea una clase `ServicioCorreo` que contiene el método `EnviarCorreo`, dedicado exclusivamente al envío de correos electrónicos.

# Ejercicio 4

```
public class Robot{
	public void Cocinar() {
		// Cocina algo
	}

	public void Limpiar() {
		// Limpia algo
	}
```

```
	public void RecargarBateria() {
		// Recarga la bateria
	}
}
```

El principio de responsabilidad única (SRP) establece que una clase debería tener una única responsabilidad. En este caso, la clase `Robot` está manejando múltiples responsabilidades: cocinar, limpiar y recargar la batería. La responsabilidad de recargar la batería debería ser manejada por otra clase, ya que no es parte del funcionamiento del robot mientras realiza tareas.

```
public class Robot{
	public void Cocinar() {
		// Cocina algo
	}

	public void Limpiar() {
		// Limpia algo
	}

public class Persona{
	public void RecargarBateria(Robot robot) {
		// Recarga la bateria
	}
}
```

Se mantiene la clase `Robot` con los métodos `Cocinar` y `Limpiar`, que representan las tareas que el robot puede realizar mientras está en funcionamiento.

Se crea una nueva clase `Persona` que contiene el método `RecargarBateria`, encargado de recargar la batería del robot, ya que esta acción no se realiza mientras el robot está en funcionamiento.

# Ejercicio 5

```
public class Cliente{
	public void CrearPedido() {
		// Crear pedido
	}
}
```

El principio de responsabilidad única (SRP) establece que una clase debería tener una única responsabilidad. En este caso, la clase `Cliente` está manejando la creación de

pedidos, lo cual debería ser responsabilidad de la `Empresa`. El cliente debería solicitar la creación de un pedido, pero la lógica de creación del pedido debería estar en la clase `Empresa`.

```
public class Pedido {
      // cuerpo pedido
}

public class Cliente{
      public void HacerPedido() {
            // Hacer pedido
      }
}

public class Empresa{
      public void CrearPedido() {
            // Crear pedido
      }
}
```

La clase `Cliente` se enfoca en hacer pedidos, delegando la responsabilidad de la creación del pedido a la `Empresa`.

Se introduce una clase `Pedido` que representa los detalles del pedido.

La clase `Empresa` contiene el método `CrearPedido`, que maneja la lógica de creación del pedido.

# Ejercicio 6

```
public class Pato {
    public void Nadar() {
        // Nada
    }

    public void Graznar() {
        // Grazna
    }

    public void Volar() {
        // Vuela
    }
}
```

```
public class PatoDeGoma : Pato {
    public override void Volar() {
        throw new NotImplementedException();
    }
}
```

El Principio de Sustitución de Liskov (LSP) establece que los objetos de una clase derivada deben poder reemplazar objetos de la clase base sin alterar el comportamiento del programa. En este caso, si un `PatoDeGoma` reemplaza a un `Pato` y se llama al método `Volar`, se lanza una excepción, lo cual altera el comportamiento esperado.

Solución:

```
public abstract class PatoLogico {
    public abstract void Nadar();

    public abstract void Graznar();
}

public class PatoDeGoma: PatoLogico {
    public override void Nadar() {
        // Nada
    }

    public override void Graznar() {
        // Grazna
    }
}

public class Pato: PatoLogico {
    public override void Nadar() {
        // Nada
    }

    public override void Graznar() {
        // Grazna
    }

    public void Volar() {
        // Vuela
    }
}
```

Se introduce una clase abstracta `PatoLogico` que define los comportamientos comunes (`Nadar` y `Graznar`) para todos los patos.

La clase `PatoDeGoma` hereda de `PatoLogico` e implementa los métodos `Nadar` y `Graznar`, pero no implementa `Volar`.

La clase `Pato` hereda de `PatoLogico` e implementa los métodos `Nadar` y `Graznar`, además se introduce el método `Volar`.

# Ejercicio 7

```
public interface IDatabase {
    void Connect();
    void Disconnect();
    void WriteData();
}

public class Database : IDatabase {
    public void Connect() {
        // logic for connecting
    }

    public void Disconnect() {
        // logic for disconnecting
    }

    public void WriteData() {
        // logic for writing data
    }
}

public class ReadDatabase : IDatabase {
    public void Connect() {
        // logic for connecting
    }

    public void Disconnect() {
        // logic for disconnecting
    }

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

El Principio de Segregación de Interfaces (ISP) establece que una clase no debería estar obligada a implementar interfaces que no usa. En este caso, la clase `ReadDatabase` está

obligada a implementar el método `WriteData` de la interfaz `IDatabase`, aunque no lo necesita.

```
public interface IDatabase {
    void Connect();
    void Disconnect();
}

public interface IWritableDatabase : IDatabase {
    void WriteData();
}

public interface IReadableDatabase : IDatabase {
    void ReadData();
}

public class Database : IWritableDatabase {
    public void Connect() {
        // Lógica para conectar
    }

    public void Disconnect() {
        // Lógica para desconectar
    }

    public void WriteData() {
        // Lógica para escribir datos
    }
}


public class ReadDatabase : IReadableDatabase {
    public void Connect() {
        // Lógica para conectar
    }

    public void Disconnect() {
        // Lógica para desconectar
    }

    public void ReadData() {
        // Lógica para leer datos
    }
}
```

Se divide la interfaz `IDatabase` en varias interfaces más pequeñas: `IDatabase` para operaciones comunes (conectar y desconectar), `IWritableDatabase` para operaciones de escritura y `IReadableDatabase` para operaciones de lectura.

La clase `Database` implementa `IWritableDatabase`, manejando tanto las operaciones comunes como las de escritura.

La clase `ReadDatabase` implementa `IReadableDatabase`, manejando tanto las operaciones comunes como las de lectura.

# Ejercicio 8

```
public class FileSaver {
    public void SaveToFile(string fileName, Document doc) {
        if (string.IsNullOrEmpty(fileName))
            throw new ArgumentException();
        // logic for saving the document
    }
}

public class AutoSave : FileSaver {
    public void Save(Document doc) {
        SaveToFile("", doc);
    }
}
```

El principio de sustitución de Liskov (LSP) establece que los objetos de una clase derivada deben poder reemplazar objetos de la clase base sin alterar el comportamiento del programa. En este caso, `AutoSave` extiende `FileSaver` pero llama al método `SaveToFile` con un `fileName` vacío, lo que lanza una excepción. Esto rompe la expectativa de que `AutoSave` pueda reemplazar a `FileSaver` sin problemas.

```
public class FileSaver {
    public void SaveToFile(string fileName, Document doc) {
        if (string.IsNullOrEmpty(fileName))
            throw new ArgumentException();
        // logic for saving the document
    }
}

public class AutoSave {
    private FileSaver fileSaver = new FileSaver();
    private string autoSaveFileName = "autosave.txt";
```

```
    public void Save(Document doc) {
        fileSaver.SaveToFile(autoSaveFileName, doc);
    }
}
```

La clase `FileSaver` se mantiene como está, con la lógica para guardar un documento en un archivo especificado.

La clase `AutoSave` ya no extiende `FileSaver`, sino que contiene una instancia de `FileSaver` para delegar la funcionalidad de guardado. También define un nombre de archivo por defecto para el guardado automático.

El método `Save` de `AutoSave` utiliza `fileSaver.SaveToFile` con un nombre de archivo predeterminado (`autoSaveFileName`), asegurando que no se lance la excepción.

# Ejercicio 9

```
public class User {
    public bool IsAdmin{ get; set; }

    public bool CanEditPost(Post post){
          return IsAdmin || post.Author == this;
    }
}

public class Post {
    public User Author{ get; set; }
}
```

El Principio de Inversión de Dependencias (DIP) sugiere que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. En este caso, la lógica para determinar si un usuario puede editar un post está embebida en la clase `User`, lo que viola este principio.

```
public interface IPermissionService {
    bool CanEditPost(User user, Post post);
}

public class PermissionService : IPermissionService {
    public bool CanEditPost(User user, Post post) {
        return user.IsAdmin || post.Author == user;
    }
}
```

```
public class User {
    private IPermissionService _permissionService;

    public User(IPermissionService permissionService) {
        _permissionService = permissionService;
    }

    public bool IsAdmin { get; set; }

    public bool CanEditPost(Post post) {
        return _permissionService.CanEditPost(this, post);
    }
}

public class Post {
    public User Author { get; set; }
}
```

Se define una interfaz `IPermissionService` que declara el método `CanEditPost`.
Se crea una clase `PermissionService` que implementa `IPermissionService` y contiene la lógica para determinar si un usuario puede editar un post.

La clase `User` tiene una dependencia de `IPermissionService`, que se inyecta a través del constructor. De esta manera, `User` delega la lógica de permisos al servicio de permisos, en lugar de manejarlo directamente.

La clase `Post` permanece sin cambios, ya que su estructura no está relacionada con la lógica de permisos.

# Ejercicio 10

```
public Class MusicPlayer {
      public void PlayMp3(string fileName){
           // Lógica para reproducir archivo MP3;
      }

      public void PlayWav(string fileName){
            // Lógica para reproducir archivo WAV
      }
```

```
      public void PlayFlac(string fileName){
            // Lógica para reproducir archivo FLAC
      }
}
```

El Principio de Abierto/Cerrado (OCP) establece que una clase debe estar abierta para extensión pero cerrada para modificación. En este caso, la clase `MusicPlayer` requiere ser modificada para añadir nuevas formas de reproducir música, lo que viola este principio.

```
public interface IMusicPlayer {
    void Play(string fileName);
}

public class Mp3Player : IMusicPlayer {
    public void Play(string fileName) {
        // Lógica para reproducir archivo MP3
    }
}

public class WavPlayer : IMusicPlayer {
    public void Play(string fileName) {
        // Lógica para reproducir archivo WAV
    }
}

public class FlacPlayer : IMusicPlayer {
    public void Play(string fileName) {
        // Lógica para reproducir archivo FLAC
    }
}

public class MusicPlayer {
    private Dictionary<string, IMusicPlayer> _players = new
Dictionary<string, IMusicPlayer>();

    public MusicPlayer() {
        // Registrando los diferentes reproductores de música
    }

    public void Play(string fileName, string fileType) {
       // implementacion
    }
}
```

Se define una interfaz `IMusicPlayer` que declara el método `Play`.

Se crean clases específicas (`Mp3Player`, `WavPlayer`, `FlacPlayer`) que implementan `IMusicPlayer` y contienen la lógica para reproducir los diferentes tipos de archivos.

La clase `MusicPlayer` contiene un diccionario de reproductores registrados y un método `Play` que selecciona el reproductor adecuado basado en el tipo de archivo.

Para añadir soporte a nuevos formatos de música, solo se necesita crear una nueva clase que implemente `IMusicPlayer` y registrarla en el diccionario de `MusicPlayer`.