UT5_TA2- 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 que anti-patrón se ajusta mejor al código presentado.

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

## Ejercicio 1

```
public class TODOController
{
    private List<Todo> todos;

    public TODOController()
    {
        this.todos = new List<Todo>();
    }

    public void Add(Todo todo)
    {
        todos.Add(todo);
    }

    public void Delete(int id)
    {
        Todo todo = todos.Find(t => t.Id == id);
        if (todo != null)
            todos.Remove(todo);
    }

    public void Update(Todo todo)
    {
        Todo oldTodo = todos.Find(t => t.Id == todo.Id);
        if (oldTodo != null)
        {
            oldTodo.Title = todo.Title;
            oldTodo.Description = todo.Description;
            oldTodo.Completed = todo.Completed;
        }
    }
```

```
    // Aquí hay más métodos relacionados a la gestión de tareas (TODOs)
    // ...
    // Y después, también hay métodos de gestión de usuarios.
    // ...
    // Y aún más, métodos para la gestión de permisos.
    // ...
}
```

El antipatrón presente es el de "The Blob" o "God Object". Este antipatrón se manifiesta cuando una sola clase se encarga de demasiadas responsabilidades, acumulando una gran cantidad de métodos y datos. Este tipo de clase se convierte en un punto centralizado que gestiona múltiples aspectos del sistema, lo que va en contra de los principios de diseño orientado a objetos.

Para resolver el antipatrón "The Blob" en la clase `TODOController`, se debe refactorizar el código para distribuir las responsabilidades en clases más pequeñas y específicas. Esto sigue el Principio de Responsabilidad Única (SRP) y ayuda a evitar la acumulación de lógica en una sola clase.

# Ejercicio 2

```
public class UserManager
{
    // A lot of code here
    // ...

    public void CreateUser(string name, string email, string password)
    {
        // Validation code
        // ...
        // Save user to database
        // ...
    }

    public void DeleteUser(int id)
    {
        // Validation code
        // ...
        // Delete user from database
        // ...
    }

    public void UpdateUser(int id, string name, string email, string
password)
    {
```

```
        // Validation code
        // ...
        // Update user in database
        // ...
    }

    public void ChangePassword(int id, string oldPassword, string
newPassword)
    {
        // Validation code
        // ...
        // Update password in database
        // ...
    }

    // More methods about user management
    // ...
}
```

El antipatrón presente es el de "The Blob" o "God Object". Este antipatrón ocurre cuando una sola clase asume demasiadas responsabilidades, manejando una gran cantidad de código y lógica que deberían estar distribuidos en clases más especializadas.

Para resolver el antipatrón "The Blob" en la clase `UserManager`, se debe refactorizar el código para distribuir las responsabilidades en clases más pequeñas y especializadas. Esto sigue el Principio de Responsabilidad Única (SRP) y ayuda a evitar la acumulación de lógica en una sola clase.

# Ejercicio 3

```
public void ProcessData()
{
    // Lots of logic here...
    int x = GetData();
    int y = x + 10;
    // More code...
    if (y > 50)
    {
        // Lots of logic here...
    }
    else
    {
        // Lots of logic here...
    }
    // Even more code...
```

```
    SaveData(y);
}
```

El antipatrón presente es el "Spaghetti Code". Este término se usa para describir un código fuente desorganizado, con una estructura compleja y enredada. El código que no sigue un diseño coherente y está lleno de condicionales y bloques de código entrelazados es difícil de leer, mantener y extender.

Para resolver el antipatrón "Spaghetti Code", es necesario refactorizar el código para hacerlo más modular y estructurado. Esto implica separar la lógica en métodos más pequeños y claros, siguiendo principios de diseño como la responsabilidad única (SRP) y la cohesión.

# Ejercicio 4

```
public void ValidateUserInput(string input)
{
// Code for validating input
}

public void ValidateUserPassword(string password)
{
// Code very similar to input validation
}
```

El antipatrón presente es el "Cut-and-Paste". Este ocurre cuando el mismo código o código muy similar se copia y pega en diferentes partes de la aplicación, en lugar de abstraer la lógica común en métodos reutilizables. Esto puede conducir a problemas de mantenimiento, errores repetidos y dificultad para realizar cambios consistentes.

Para resolver el antipatrón "Cut-and-Paste", es necesario refactorizar el código para eliminar duplicaciones. Se puede extraer la lógica de validación común en un método separado que sea reutilizable por ambos métodos de validación específicos.

# Ejercicio 5

```
public class OldUnusedClass
{
// Lots of unused code here...
}
```

El antipatrón presente es el "Lava Flow". Este término se usa para describir el código antiguo y no utilizado que permanece en el sistema "por si acaso". Este tipo de código no

tiene una función clara y ocupa espacio innecesario en la base de código, haciendo más difícil la mantenibilidad y la legibilidad del sistema.

La mejor manera de abordar el antipatrón "Lava Flow" es eliminar el código no utilizado. Antes de hacerlo, es importante probar la eliminación para asegurarse de que no haya impacto negativo en el sistema. Si el código es necesario, se debe hacer un esfuerzo para entender y refactorizarlo.

# Ejercicio 6

```
public class Superclass
{
	public virtual void DoWork()
	{
		// Do some work
	}
}

public class Subclass : Superclass
{
	public override void DoWork()
	{
		// Do completely different work, unrelated to the
superclass's work
	}
}
```

Se observa la presencia del antipatrón "Spaghetti Code". Este término se refiere a un estilo de programación donde la estructura del código es desorganizada y compleja. En lugar de seguir un diseño modular y claro, el código se entrelaza con condicionales y bloques de código complicados, dificultando su comprensión y mantenimiento.

# Ejercicio 7

```
public class DataProcessor
{
	// This is a specific library or tool used everywhere
	SpecificLibrary library = new SpecificLibrary();

	public void ProcessData(List<int> data)
	{
		library.Method1(data);
		library.Method2(data);
		library.Method3(data);
```

```
        }
}
```

El antipatrón que se presenta en el código es "Golden Hammer". Este término se refiere a la tendencia de utilizar una herramienta o librería específica de manera generalizada en múltiples partes del código, a pesar de que podría no ser la opción más adecuada para todos los casos. En el ejemplo dado, la clase `DataProcessor` utiliza la `SpecificLibrary` para procesar datos mediante los métodos `Method1`, `Method2`, y `Method3`.

Para resolver el antipatrón "Golden Hammer", es recomendable realizar una evaluación crítica de la elección de la librería en cuestión. Esto implica considerar si la `SpecificLibrary` es la mejor opción en términos de rendimiento, eficiencia, mantenibilidad y alineación con los requisitos específicos de cada parte del código.

# Ejercicio 8

```
public class MyClass
{
      public void DoManyThings()
      {
            // Lot of code for task 1 ...
            // Lot of code for task 2 ...
            // Lot of code for task 3 ...
            // Lot of code for task 4 ...
      }
}
```

Se identifica la presencia del antipatrón conocido como "The Blob". Este antipatrón se refiere a una clase que asume demasiadas responsabilidades diferentes, concentrando una cantidad significativa de lógica y funcionalidad dentro de un único método o clase. En este caso específico, la clase `MyClass` tiene un método `DoManyThings` que probablemente realiza múltiples tareas distintas.

Para resolver el antipatrón "The Blob", la estrategia recomendada es la refactorización hacia una estructura más modular y cohesiva. Esto implica dividir el método `DoManyThings` en métodos más pequeños y específicos, cada uno encargado de una tarea única. Cada método debería ser responsable de una única responsabilidad (principio de responsabilidad única).

# Ejercicio 9

```
public double CalculateNetSalary(double grossSalary)
{
      double taxRate;
      double netSalary;

      if (grossSalary > 10000)
      {
            taxRate = 0.3;
      }
      if (grossSalary > 5000)
      {
            taxRate = 0.3;
      }
      else
      {
            taxRate = 0.1;
      }

      netSalary = grossSalary - (grossSalary * taxRate);

      if (netSalary < 0)
      {
            return netSalary
      }
}
```

Se puede identificar la presencia del antipatrón "Spaghetti Code", debido a que el estilo de programación tiene una estructura de código compleja y enredada, dificultando su legibilidad y comprensión.

# Ejercicio 10

```
public class ShoppingCart
{
    private Dictionary<Product, int> _items = new Dictionary<Product,
int>();

    public void AddProduct(Product product, int quantity)
    {
        if (_items.ContainsKey(product))
        {
            _items[product] += quantity;
        }
```

```
        else
        {
            _items.Add(product, quantity);
        }
    }

    // ... Otros métodos ...
}
```

Luego de unas iteraciones dev-test:

```
public class ShoppingCart
{
    private Dictionary<Product, int> _items = new Dictionary<Product,
int>();
    private const int MAX_QUANTITY = 10;

    public void AddProduct(Product product, int quantity)
    {
        if (quantity > MAX_QUANTITY)
        {
            throw new ArgumentException($"Can't add more than
{MAX_QUANTITY} of a product at once");
        }

        if (_items.ContainsKey(product))
        {
            _items[product] += quantity;
        }
        else
        {
            _items.Add(product, quantity);
        }
    }

    // ... Otros métodos ...
}
```

Se puede identificar la presencia del antipatrón "Tester Driven Development". Este antipatrón ocurre cuando las decisiones de diseño y código están predominantemente influenciadas por pruebas de software o informes de errores específicos, en lugar de seguir un enfoque más equilibrado y centrado en el diseño del sistema.

Inicialmente, la clase `ShoppingCart` implementa la funcionalidad básica de agregar productos con cantidades específicas a través del método `AddProduct`. Después de

algunas iteraciones de pruebas, se introduce un cambio significativo: se agrega una validación adicional para asegurar que la cantidad de productos añadidos no supere un límite máximo (`MAX_QUANTITY`).