

## UNIDAD TEMÁTICA 5 – Patrones de diseño– Trabajo de Aplicación 1

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.
    // ...
}
```

Parte 1: Dado que dice que hay métodos de gestión de usuarios y de gestión de permisos en la clase TODOController se podría decir que el antipatrón de The Blob.

Parte 2:

```
public class TODOController {  
    private List<Todo> todos;  
    public TODOController(){  
        this.todos = new List<Todo>();  
    }  
    public void add(Todo todo){  
        this.todos.Add(todo);  
    }  
    //Codigo con metodos de gestion de tareas  
}  
  
public class GestionUsuarios{  
    //Codigo con metodos de gestion de usuarios  
}  
  
public class GestionTareas{  
    //Codigo con metodos de gestion de tareas  
}
```

## 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
    // ...
}
```

Parte 1: El antipatrón detectado es Spaghetti code porque le falta estructura y se lo ve difícil de mantener, pues cada método tiene validaciones

Parte 2

```
public class UserManager{
    public void add(User user){
        System.out.println(user.getFirstName() + " " + user.getLastName() + " added.");
    }
    public void delete(User user){
        System.out.println(user.getFirstName() + " " + user.getLastName() + " deleted.");
    }
}
```

```

    }
    public void update(User user){
        System.out.println(user.getFirstName() + " " + user.getLastName() + " updated.");
    }
    public void addMultiple(User[] users){
        for(User user : users){
            add(user);
        }
    }
    public void deleteMultiple(User[] users){
        for(User user : users){
            delete(user);
        }
    }
    public void updateMultiple(User[] users){
        for(User user : users){
            update(user);
        }
    }
}

public class ValidationCode{
    // Validation code
}

```

### 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);
}

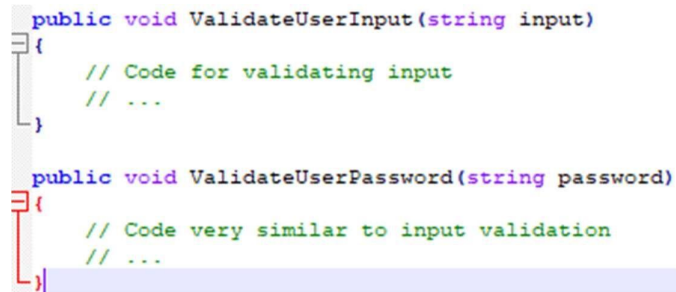
```

Parte 1: Spaghetti code es el antipatrón detectado acá porque hay mucho código difícil de mantener y desestructurado, ejemplo si se desea agregar otro condicional para if hay que buscarlo en todo el método.

Parte 2

```
public void ProcessData()
{
    int x = GetData();
    int y = x+ 10;
    Logic(y);
    SaveData(y);
}
public void Logic(int y){
    if (y>50) {
        // do something
    } else {
        // do something else
    }
}
```

#### EJERCICIO 4



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

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

Parte 1: Se detecta antipatron cut and paste programming pues el codigo de ValidateUserPassword es muy similar al de ValidateUserInput

Parte 2:

```
public void ValidateUserInput(string input){
    Validation(input);
}
```

```

public void ValidateUserInput(string password){
    Validation(password);
}
public void Validation (string input){
}

```

#### EJERCICIO 5

```

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

```

Parte 1: Dado que hay mucho código sin uso se detecta el antipatrón LavaFlow ya que se incluye código de investigación o funcionalidades innecesarias.

#### 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
    }
}

```

Parte 1:

Se detecta the Golden Hammer haciendo mucho uso de herencia de manera innecesaria pues el metodo que se sobrescribe está totalmente no relacionado con la de SuperClass.

Parte 2:

La solución sería quitar esa herencia innecesaria, creando el método tal como la SubClass necesita.

## 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);
    }
}
```

Parte 1:

The golden hammer: Dice que esta herramienta se usa en todos lados.

Parte 2:

La solución es solo hacer uso de esa herramienta en los lugares que es necesario.

## 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
        // ...
    }
}
```

Parte 1:

Spaghetti code pues hay mucho código para cada Task y The Blob, pues es el método DoManyThings que está monopolizando todas las tareas.

Parte 2:

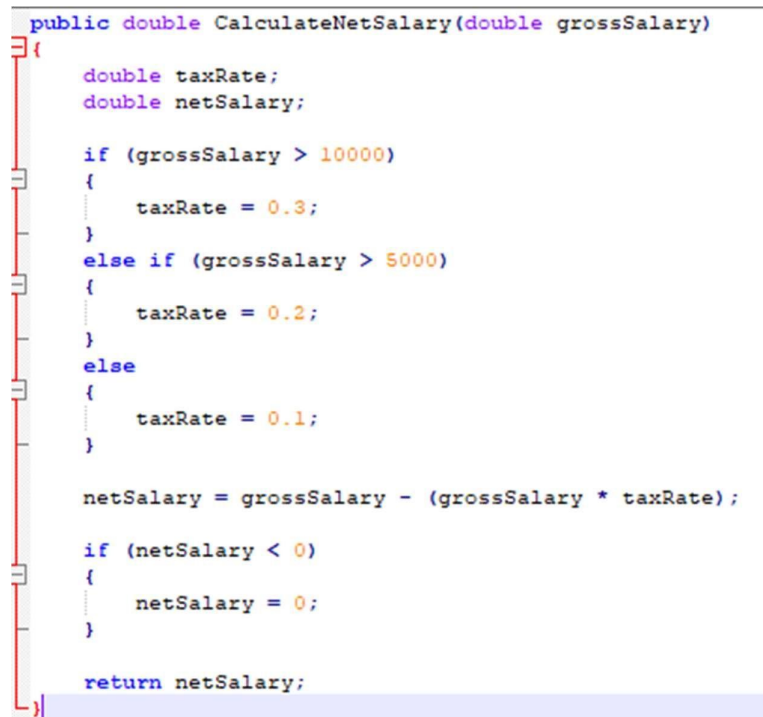
```
public class MyClass {
    public void Task1(){
        // Task 1
    }
    public void Task2(){
```

```

    // Task 2
}
public void Task3(){
    // Task 3
}
public void Task4(){
    // Task 4
}
}

```

## EJERCICIO 9



```

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

    if (grossSalary > 10000)
    {
        taxRate = 0.3;
    }
    else if (grossSalary > 5000)
    {
        taxRate = 0.2;
    }
    else
    {
        taxRate = 0.1;
    }

    netSalary = grossSalary - (grossSalary * taxRate);

    if (netSalary < 0)
    {
        netSalary = 0;
    }

    return netSalary;
}

```

Parte 1: Spaghetti Code pues en el metodo hay demasiada logica que se podria separar, pues cualquier cambio que se quiera realizar resulta complicado.

Parte 2:

```

public double CalculateNetSalary (double grossSalary){
    double netSalary;
    double taxRate;
    final double taxRate = GetTaxRate(grossSalary);
    netSalary = grossSalary - (grossSalary * taxRate);
    if (netSalary < 0){

```



```

        netSalary = 0;
    }
    return netSalary;
}
public double GetTaxRate(double grossSalary){
    double taxRate;
    if (grossSalary > 1000){
        taxRate = 0.3;
    }
    else if (grossSalary > 500){
        taxRate = 0.2;
    }
    else {
        taxRate = 0.1;
    }
    return taxRate;
}

```

## 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 ...
}

```

Parte 1:

Tester-driven development: Luego de las iteraciones del test se agrego un nuevo requerimiento.

Parte 2: La solución es hacer un buen y completo análisis de requerimientos desde el inicio.