UT5_TA3 Ignacio Villarreal, Bruno Albín, Santiago Aurrecochea, Joaquin Gasco, José Varela y Gonzalo Paz.

Patrones de diseño

Ejercicio 0

En el patrón Builder, ¿qué principios SOLID se aplican? ¿Qué principios se violan, si los hubiera? Justifiquen su respuesta.

El patrón Builder es un patrón de diseño creacional que permite la construcción de objetos complejos paso a paso.

Los principios que se aplican en el builder son:

- **SRP:** El patrón Builder divide la construcción de un objeto complejo en pasos discretos y delega la responsabilidad de cada paso a una clase concreta. La clase Builder tiene la única responsabilidad de construir el objeto y encapsula la lógica de construcción. Esto permite que la clase principal (el objeto complejo) no se encargue de los detalles de su propia construcción.
- **OCP:** El patrón Builder permite extender la funcionalidad de construcción de objetos sin modificar las clases existentes. Se pueden crear nuevos constructores específicos que heredan de un constructor abstracto para añadir nuevas formas de construcción.
- **ISP:** Si el patrón Builder está bien diseñado, cada builder concreto debe implementar una interfaz que defina solo los métodos necesarios para la construcción del objeto. De este modo, se evita que las clases cliente estén obligadas a depender de métodos que no usan.
- **DIP:** El patrón Builder aplica el DIP al depender de abstracciones en lugar de implementaciones concretas. La clase que utiliza el builder (el Director) se comunica con el builder a través de una interfaz.

Los principios que se podrian violar son:

- **LSP:** Podría haber una violación del LSP si los constructores concretos no se comportan de manera coherente con la interfaz del builder abstracto. Por ejemplo, si un builder concreto ignora algunos pasos de construcción que otros builders implementan, esto podría llevar a comportamientos inesperados cuando se sustituye un builder por otro.

Para cada uno de los siguientes ejercicios, en equipo:

- Determine que patrón puede resolver el problema de una forma más eficiente.
- Agregue las clases, interfaces, métodos que considere necesarios para remediar la situación.

Ejercicio 1

```
public class Sandwich
{
    public string Bread { get; set; }
    public string Cheese { get; set; }
    public string Meat { get; set; }
    public string Vegetables { get; set; }
    public string Condiments { get; set; }

    public Sandwich(string bread, string cheese, string meat, string
vegetables, string condiments)
    {
        Bread = bread;
        Cheese = cheese;
        Meat = meat;
        Vegetables = vegetables;
        Condiments = condiments;
    }

    public override string ToString()
    {
        return $"Sandwich with {Bread} bread, {Cheese} cheese, {Meat}
meat, {Vegetables} vegetables, and {Condiments} condiments.";
    }
}

class Program
{
      static void Main(string[] args)
      {
             Sandwich hamSandwich = new Sandwich("White", "Swiss",
"Ham", "Lettuce, Tomato", "Mayo, Mustard");
             Sandwich turkeySandwich = new Sandwich("Wheat", "Cheddar",
"Turkey", null, "Mayo");
            Console.WriteLine(hamSandwich);
            Console.WriteLine(turkeySandwich);
      }
}
```

Para mejorar la construcción de objetos `Sandwich` y hacer el código más limpio y flexible, se podría usar el patrón **Builder**. Este patrón permite crear objetos complejos paso a paso y es ideal para casos en los que hay muchas propiedades opcionales o configuraciones diferentes.

```
public class Sandwich
{
    public string Bread { get; private set; }
    public string Cheese { get; private set; }
    public string Meat { get; private set; }
    public string Vegetables { get; private set; }
    public string Condiments { get; private set; }

    private Sandwich() { }

    public override string ToString()
    {
        return $"Sandwich with {Bread} bread, {Cheese} cheese, {Meat}
meat, {Vegetables} vegetables, and {Condiments} condiments.";
    }
}

public class SandwichBuilder
{
    private readonly Sandwich _sandwich;

    public SandwichBuilder()
    {
        _sandwich = new Sandwich();
    }

    public SandwichBuilder SetBread(string bread)
    {
        _sandwich.Bread = bread;
        return this;
    }

    public SandwichBuilder SetCheese(string cheese)
    {
        _sandwich.Cheese = cheese;
        return this;
    }

    public SandwichBuilder SetMeat(string meat)
    {
        _sandwich.Meat = meat;
        return this;
    }

    public SandwichBuilder SetVegetables(string vegetables)
    {
```

```
        _sandwich.Vegetables = vegetables;
        return this;
    }

    public SandwichBuilder SetCondiments(string condiments)
    {
        _sandwich.Condiments = condiments;
        return this;
    }

    public Sandwich Build()
    {
        return _sandwich;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Sandwich hamSandwich = new SandwichBuilder()
            .SetBread("White")
            .SetCheese("Swiss")
            .SetMeat("Ham")
            .SetVegetables("Lettuce, Tomato")
            .SetCondiments("Mayo, Mustard")
            .Build();

        Sandwich turkeySandwich = new SandwichBuilder()
            .SetBread("Wheat")
            .SetCheese("Cheddar")
            .SetMeat("Turkey")
            .SetCondiments("Mayo")
            .Build();

        Console.WriteLine(hamSandwich);
        Console.WriteLine(turkeySandwich);
    }
}
```

En la clase `Sandwich` las propiedades se han hecho privadas para que solo puedan ser modificadas a través del builder. El constructor de la clase `Sandwich` es privado para forzar la creación de objetos `Sandwich` a través del `Builder`.

La clase `Builder` es una clase que tiene una instancia de `Sandwich`. Tiene métodos para establecer cada propiedad de `Sandwich` (`SetBread`, `SetCheese`, `SetMeat`, `SetVegetables`, `SetCondiments`). Cada uno de estos métodos devuelve la instancia actual del builder (`this`), permitiendo el encadenamiento de métodos. El método `Build` devuelve el objeto `Sandwich` construido.

En el `Main` se utilizan los métodos del builder para crear diferentes tipos de sándwiches.

Ejercicio 2

```
public abstract class GameUnit
{
    public int Health { get; set; }
    public int Attack { get; set; }
    public int Defense { get; set; }

    // Simula la carga de recursos costosos como modelos 3D, texturas, etc.
    public virtual void LoadResources()
    {
        Console.WriteLine("Loading resources...");
    }
}

public class Archer : GameUnit
{
    public Archer()
    {
        LoadResources();
        Health = 100;
        Attack = 15;
        Defense = 5;
    }
}

public class Knight : GameUnit
{
    public Knight()
    {
        LoadResources();
        Health = 200;
        Attack = 20;
        Defense = 10;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Creating original Archer...");
        Archer originalArcher = new Archer();

        Console.WriteLine("Copying Archers manually...");
        Archer copiedArcher1 = new Archer { Health =
originalArcher.Health, Attack = originalArcher.Attack, Defense =
originalArcher.Defense };
        Archer copiedArcher2 = new Archer { Health =
originalArcher.Health, Attack = originalArcher.Attack, Defense =
originalArcher.Defense };

        Console.WriteLine("Creating original Knight...");
        Knight originalKnight = new Knight();

        Console.WriteLine("Copying Knights manually...");
        Knight copiedKnight1 = new Knight { Health =
originalKnight.Health, Attack = originalKnight.Attack, Defense =
originalKnight.Defense };
        Knight copiedKnight2 = new Knight { Health =
originalKnight.Health, Attack = originalKnight.Attack, Defense =
originalKnight.Defense };
    }
}
```

Para mejorar la creación y clonación de instancias en el ejercicio presentado, el patrón Prototype es una opción adecuada. Este patrón permite clonar objetos de manera eficiente sin necesidad de realizar una nueva inicialización costosa (como la carga de recursos) cada vez.

```
public abstract class GameUnit
{
    public int Health { get; set; }
    public int Attack { get; set; }
    public int Defense { get; set; }

    // Simula la carga de recursos costosos como modelos 3D, texturas,
etc.
    public virtual void LoadResources()
    {
```

```
        Console.WriteLine("Loading resources...");
    }

    public abstract GameUnit Clone();
}

public class Archer : GameUnit
{
    public Archer()
    {
        LoadResources();
        Health = 100;
        Attack = 15;
        Defense = 5;
    }

    // Constructor privado para la clonación
    private Archer(int health, int attack, int defense)
    {
        Health = health;
        Attack = attack;
        Defense = defense;
    }

    public override GameUnit Clone()
    {
        return new Archer(Health, Attack, Defense);
    }
}

public class Knight : GameUnit
{
    public Knight()
    {
        LoadResources();
        Health = 200;
        Attack = 20;
        Defense = 10;
    }

    // Constructor privado para la clonación
    private Knight(int health, int attack, int defense)
    {
        Health = health;
        Attack = attack;
        Defense = defense;
```

```
    }

    public override GameUnit Clone()
    {
        return new Knight(Health, Attack, Defense);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Archer originalArcher = new Archer();

        Archer copiedArcher1 = (Archer)originalArcher.Clone();
        Archer copiedArcher2 = (Archer)originalArcher.Clone();

        Knight originalKnight = new Knight();

        Knight copiedKnight1 = (Knight)originalKnight.Clone();
        Knight copiedKnight2 = (Knight)originalKnight.Clone();
    }
}
```

`GameUnit` es una clase abstracta que define las propiedades `Health`, `Attack` y `Defense`. Contiene el método `LoadResources` y define un método abstracto `Clone` que debe ser implementado por las clases derivadas.

La clase `Archer` hereda de `GameUnit` y en su constructor inicializa las propiedades y carga los recursos. Define un constructor privado utilizado por el método `Clone` para crear una copia del objeto sin cargar los recursos de nuevo. Implementa el método `Clone` que retorna una nueva instancia de `Archer` con los mismos valores de propiedad.

La clase `Knight` es similar a `Archer`.

En el `Main` se crea una instancia original de `Archer` y `Knight`. Se crean copias de las instancias originales utilizando el método `Clone`.

Ejercicio 3

```
public class MessagingApp
{
    public void SendMessage(string serviceType, string message)
    {
```

```
        if (serviceType == "SMS")
        {
            Console.WriteLine($"Sending SMS message: {message}");
            // Lógica para enviar SMS...
        }
        else if (serviceType == "Email")
        {
            Console.WriteLine($"Sending Email: {message}");
            // Lógica para enviar Email...
        }
        else if (serviceType == "Facebook")
        {
            Console.WriteLine($"Sending Facebook Message: {message}");
            // Lógica para enviar mensaje de Facebook...
        }
    }
}
```

El más adecuado para resolver este problema sería el patrón Factory. Este patrón permite crear objetos sin especificar la clase exacta del objeto que se va a crear, proporcionando una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

```
public interface IMessageService
{
    void SendMessage(string message);
}

public class SmsService : IMessageService
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Sending SMS message: {message}");
        // Lógica para enviar SMS...
    }
}

public class EmailService : IMessageService
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Sending Email: {message}");
        // Lógica para enviar Email...
    }
}
```

```
public class FacebookService : IMessageService
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Sending Facebook Message: {message}");
        // Lógica para enviar mensaje de Facebook...
    }
}

public static class MessageServiceFactory
{
    public static IMessageService CreateMessageService(string
serviceType)
    {
        return serviceType switch
        {
            "SMS" => new SmsService(),
            "Email" => new EmailService(),
            "Facebook" => new FacebookService(),
            _ => throw new ArgumentException("Invalid service type",
nameof(serviceType))
        };
    }
}

public class MessagingApp
{
    private readonly IMessageService _messageService;

    public MessagingApp(IMessageService messageService)
    {
        _messageService = messageService ?? throw new
ArgumentNullException(nameof(messageService));
    }

    public void SendMessage(string message)
    {
        _messageService.SendMessage(message);
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
        var smsService =
MessageServiceFactory.CreateMessageService("SMS");
        var app1 = new MessagingApp(smsService);
        app1.SendMessage("SMS!");

        var emailService =
MessageServiceFactory.CreateMessageService("Email");
        var app2 = new MessagingApp(emailService);
        app2.SendMessage("Email!");

        var facebookService =
MessageServiceFactory.CreateMessageService("Facebook");
        var app3 = new MessagingApp(facebookService);
        app3.SendMessage("Facebook!");
    }
}
```

La Interfaz `IMessageService` define el método `SendMessage` que debe ser implementado por todas las clases de servicios de mensajería.

Las clases concretas (`SmsService`, `EmailService`, `FacebookService`) implementan la interfaz `IMessageService` y proporcionan la lógica específica para enviar mensajes a través de diferentes servicios.

La Clase `MessageServiceFactory` es una clase estática que contiene el método `CreateMessageService` para crear instancias de `IMessageService` basadas en el tipo de servicio proporcionado. Utiliza una expresión `switch` para devolver la instancia adecuada o lanzar una excepción si el tipo de servicio es inválido.

La clase `MessagingApp` tiene un campo `_messageService` que es del tipo `IMessageService`. El constructor acepta una instancia de `IMessageService` y la asigna al campo `_messageService`. El método `SendMessage` utiliza el servicio de mensajería configurado para enviar el mensaje.

En el `Main` se utiliza `MessageServiceFactory` para crear diferentes servicios de mensajería.

Ejercicio 4

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public List<string> BorrowedStudents { get; set; }
```

```
    public Book()
    {
        // Simular la carga de recursos.
        Console.WriteLine("Acquiring a new book...");
        BorrowedStudents = new List<string();
    }

    public void BorrowBook(string studentName)
    {
        BorrowedStudents.Add(studentName);
    }

    public void PrintBorrowedStudents()
    {
        Console.WriteLine($"Book: {Title}, Borrowed by: {string.Join(",
", BorrowedStudents)}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Adquirir el libro original.
        Book originalBook = new Book
        {
            Title = "Harry Potter",
            Author = "J.K. Rowling"
        };

        // Prestar el libro original a un estudiante.
        originalBook.BorrowBook("Alice");

        // Adquirir una copia adicional del mismo libro manualmente.
        Book additionalCopy = new Book
        {
            Title = originalBook.Title,
            Author = originalBook.Author,
            BorrowedStudents = new List<string>() // Inicializar la
lista vacía.
        };
        additionalCopy.BorrowBook("Bob");
        // Imprimir los estudiantes a los que se les prestó cada copia
del libro
        originalBook.PrintBorrowedStudents();
```

```
        additionalCopy.PrintBorrowedStudents();
    }
}
```

Para mejorar la creación de copias de libros, el patrón Prototype es una opción adecuada. Este patrón permite clonar objetos de manera eficiente, evitando la necesidad de repetir la lógica de inicialización costosa. Al aplicar el patrón Prototype, podemos crear copias de libros de manera más sencilla y con menos código duplicado.

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public List<string> BorrowedStudents { get; set; }

    public Book()
    {
        // Simular la carga de recursos.
        Console.WriteLine("Acquiring a new book...");
        BorrowedStudents = new List<string>();
    }

    // Constructor privado para clonación
    private Book(string title, string author, List<string>
borrowedStudents)
    {
        Title = title;
        Author = author;
        BorrowedStudents = new List<string>(borrowedStudents);
    }

    public void BorrowBook(string studentName)
    {
        BorrowedStudents.Add(studentName);
    }

    public void PrintBorrowedStudents()
    {
        Console.WriteLine($"Book: {Title}, Borrowed by: {string.Join(",
", BorrowedStudents)}");
    }

    // Método para clonar el libro
    public Book Clone()
    {
```

```
        return new Book(Title, Author, new List<string>());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Adquirir el libro original.
        Book originalBook = new Book
        {
            Title = "Harry Potter",
            Author = "J.K. Rowling"
        };

        // Prestar el libro original a un estudiante.
        originalBook.BorrowBook("Alice");

        // Adquirir una copia adicional del mismo libro utilizando
Prototype.
        Book additionalCopy = originalBook.Clone();
        additionalCopy.BorrowBook("Bob");

        // Imprimir los estudiantes a los que se les prestó cada copia
del libro.
        originalBook.PrintBorrowedStudents();
        additionalCopy.PrintBorrowedStudents();
    }
}
```

La clase `Book` tiene las propiedades `Title`, `Author` y `BorrowedStudents`. El constructor inicializa las propiedades. El constructor privado para clonación copia el título y el autor, y crea una nueva lista vacía para `BorrowedStudents`. El método `Clone` crea una nueva instancia de `Book` con los mismos valores de `Title` y `Author`, pero con una lista vacía para `BorrowedStudents`.

En el `Main` se crea una instancia original de `Book` y se le presta a un estudiante. Se clona el libro original utilizando el método `Clone` y se le presta a otro estudiante. Se imprimen los estudiantes a los que se les prestó cada copia del libro.

Ejercicio 5

```
public class TravelPlan
{
```

```
    public TravelPlan(string flight, string hotel, string carRental,
                      string[] activities, string[]
restaurantReservations,
                      /* otros parámetros opcionales aquí */)
    {
        // Constructor con muchos parámetros, algunos de los cuales
pueden
        // ser opcionales (nulos o valores predeterminados).
    }
    // Propiedades y métodos...
}

// Ejemplo de uso:
TravelPlan plan = new TravelPlan("Flight1", "Hotel1", null, new string[]
{"Tour1", "Tour2"}, null,...) /* otros argumentos opcionales aquí */);
```

Para resolver el problema de la creación de objetos con muchos parámetros, algunos de los cuales son opcionales, el patrón Builder es ideal. Este patrón permite construir objetos complejos paso a paso y proporciona una interfaz fluida para configurar los parámetros.

```
public class TravelPlan
{
    public string Flight { get; private set; }
    public string Hotel { get; private set; }
    public string CarRental { get; private set; }
    public List<string> Activities { get; private set; }
    public List<string> RestaurantReservations { get; private set; }

    // Constructor privado para el Builder
    private TravelPlan() { }

    public static TravelPlanBuilder CreateBuilder()
    {
        return new TravelPlanBuilder();
    }
}

    public class TravelPlanBuilder
    {
        private readonly TravelPlan _travelPlan;

        public TravelPlanBuilder()
        {
            _travelPlan = new TravelPlan();
        }
```

```
        public TravelPlanBuilder SetFlight(string flight)
        {
            _travelPlan.Flight = flight;
            return this;
        }

        public TravelPlanBuilder SetHotel(string hotel)
        {
            _travelPlan.Hotel = hotel;
            return this;
        }

        public TravelPlanBuilder SetCarRental(string carRental)
        {
            _travelPlan.CarRental = carRental;
            return this;
        }

        public TravelPlanBuilder SetActivities(IEnumerable<string>
activities)
        {
            _travelPlan.Activities = new List<string>(activities);
            return this;
        }

        public TravelPlanBuilder
SetRestaurantReservations(IEnumerable<string> restaurantReservations)
        {
            _travelPlan.RestaurantReservations = new
List<string>(restaurantReservations);
            return this;
        }

        public TravelPlan Build()
        {
            return _travelPlan;
        }
    }

class Program
{
    static void Main(string[] args)
    {
        TravelPlan plan = TravelPlan.CreateBuilder()
            .SetFlight("Flight1")
```

```
            .SetHotel("Hotel1")
            .SetActivities(new[] { "Tour1", "Tour2" })
            .Build();

        Console.WriteLine($"Flight: {plan.Flight}");
        Console.WriteLine($"Hotel: {plan.Hotel}");
        Console.WriteLine($"Car Rental: {plan.CarRental ?? "None"}");
        Console.WriteLine($"Activities: {string.Join(", ",
plan.Activities ?? new List<string>())}");
        Console.WriteLine($"Restaurant Reservations: {string.Join(", ",
plan.RestaurantReservations ?? new List<string>())}");
    }
}
```

La Clase `TravelPlan` tiene un constructor privado para que solo pueda ser invocado por `TravelPlanBuilder` y un método estático `CreateBuilder` que devuelve una instancia de `TravelPlanBuilder`.

La Clase `TravelPlanBuilder` es la encargada de construir una instancia de `TravelPlan`. Contiene métodos para establecer cada una de las propiedades del `TravelPlan`, que devuelven una referencia al `Builder` para permitir el encadenamiento de llamadas. Método `Build` que retorna la instancia configurada de `TravelPlan`.

En el `Main` se crea una instancia de `TravelPlan` utilizando el `Builder`, configurando solo los parámetros necesarios. Se muestran los detalles del `TravelPlan` creado, con valores predeterminados para los parámetros que no se configuraron.

Ejercicio 6

```
class Program
{
    static void Main()
    {
        // Crear un servicio
        SomeService service = new SomeService();

        // Realizar una tarea que requiere configuración
        service.PerformTask();

        // Otro ejemplo: acceder a la configuración desde otra parte de
la aplicación
        string apiEndpoint =
ConfigurationManager.Instance.GetConfiguration("apiEndpoint");
```

```
        Console.WriteLine($"API Endpoint: {apiEndpoint}");
    }
}
```

Podemos utilizar el patrón Singleton para implementar la clase `ConfigurationManager`, asegurándonos de que haya una única instancia de la clase en toda la aplicación. Esto es especialmente útil para manejar configuraciones globales.

```
public class ConfigurationManager
{
    private static ConfigurationManager instance;
    private Dictionary<string, string> configuration;

    // Constructor privado para evitar la creación de instancias desde
fuera de la clase
    private ConfigurationManager()
    {
        // Aquí puedes inicializar la configuración si es necesario
        configuration = new Dictionary<string, string>();
        configuration.Add("apiEndpoint", "https://api.example.com"); //
Ejemplo de configuración inicial
    }

    // Método estático para obtener la única instancia de
ConfigurationManager
    public static ConfigurationManager Instance
    {
        get
        {
            // Crear la instancia si no existe
            if (instance == null)
            {
                instance = new ConfigurationManager();
            }
            return instance;
        }
    }

    // Método para obtener configuraciones específicas
    public string GetConfiguration(string key)
    {
        if (configuration.ContainsKey(key))
        {
            return configuration[key];
        }
```

```
        else
        {
            return null; // O manejar el caso de configuración no
encontrada según tu lógica
        }
    }
}

class Program
{
    static void Main()
    {
        // Crear un servicio
        SomeService service = new SomeService();

        // Realizar una tarea que requiere configuración
        service.PerformTask();

        // Acceder a la configuración desde otra parte de la aplicación
usando el Singleton
        string apiEndpoint =
ConfigurationManager.Instance.GetConfiguration("apiEndpoint");
        Console.WriteLine($"API Endpoint: {apiEndpoint}");
    }
}
```

ConfigurationManager se convierte en una clase Singleton asegurando que solo exista una instancia de esta clase durante toda la ejecución del programa. El constructor es privado para evitar que se pueda crear una instancia desde fuera de la clase. El método estático `Instance` es el punto de acceso único para obtener la instancia de `ConfigurationManager`. Si `instance` es `null`, se crea una nueva instancia; de lo contrario, se devuelve la instancia existente. `GetConfiguration` permite acceder a las configuraciones almacenadas en `configuration`, un diccionario que podría ser cargado desde un archivo de configuración, base de datos, o definido estáticamente como en el ejemplo.

El Main crea una instancia de `SomeService` y realiza alguna tarea. Luego accede al `ConfigurationManager.Instance` para obtener el valor del `apiEndpoint` y lo muestra por consola.