# Desarrollo de un Proyecto en C# utilizando Principios de Orientación a Objetos

Autor: Matheus Nantes Y Abiel Gustavo

27 de noviembre de 2024

**Resumen**

Este documento describe el trabajo desarrollado en un proyecto en C# que utiliza los principios fundamentales de la programación orientada a objetos (POO), como clases, herencia, polimorfismo, abstracción y encapsulamiento. A lo largo del proyecto, se explican los detalles de la implementación de cada uno de estos conceptos dentro del sistema, y cómo contribuyen al diseño limpio y modular del mismo.

# Índice

# 1. Introducción

La programación orientada a objetos es un paradigma de programación que organiza el código en torno a objetos, que son instancias de clases. Este paradigma permite crear aplicaciones más modulares, reutilizables y fáciles de mantener. El presente trabajo tiene como objetivo implementar un sistema en C# que haga uso de los conceptos fundamentales de la programación orientada a objetos. Durante el desarrollo del proyecto, se implementaron diversas clases que interactúan entre sí, utilizando principios como la herencia, el polimorfismo, la abstracción y el encapsulamiento.

# 2. Clases y Objetos

Una **clase** es una plantilla para crear objetos. Define las propiedades y comportamientos que los objetos de esa clase tendrán. En el proyecto desarrollado, se crearon diversas clases que representaban entidades dentro del sistema, como `Agenda`, `ElementoAgenda`, `ContactoFamiliar`, `ContactoProfesional`, `Evento` y `Persona`.

## 2.1. Clase `Agenda`

La clase `Agenda` es responsable de gestionar una lista de contactos y eventos. Tiene métodos para agregar, eliminar, buscar y editar elementos en la agenda.

```
public class Agenda
{
    private List<ElementoAgenda> elementos;

    public Agenda()
    {
        elementos = new List<ElementoAgenda>();
    }

    public void AgregarElemento(ElementoAgenda elemento)
    {
        elementos.Add(elemento);
    }

    public void BuscarElemento(string nombre)
    {
```

```
        var elemento = elementos.FirstOrDefault(e => e.Nombre == nombre);
        if (elemento != null)
        {
            Console.WriteLine(elemento);
        }
    }

    public void EliminarElemento(string nombre)
    {
        var elemento = elementos.FirstOrDefault(e => e.Nombre == nombre);
        if (elemento != null)
        {
            elementos.Remove(elemento);
        }
    }
}
```

### 2.2. Clase `ElementoAgenda`

La clase `ElementoAgenda` representa los elementos de la agenda, como contactos o eventos. Cada elemento tiene propiedades como el nombre, teléfono y relación en el caso de los contactos familiares o profesionales, y nombre, fecha y descripción en el caso de los eventos.

```
public class ElementoAgenda
{
    public string Nombre { get; set; }

    public ElementoAgenda(string nombre)
    {
        Nombre = nombre;
    }
}
```

## 3. Herencia y Polimorfismo

La **herencia** es un principio fundamental de la programación orientada a objetos que permite que una clase derive de otra, heredando sus propiedades y métodos. El **polimorfismo** permite que las clases derivadas puedan sobrescribir los métodos de la clase base para comportarse de manera diferente.

## 3.1. Herencia en el Proyecto

En el proyecto, la clase `ElementoAgenda` es la clase base para diferentes tipos de elementos, como `ContactoFamiliar`, `ContactoProfesional` y `Evento`. Cada una de estas clases derivadas hereda de `ElementoAgenda` y agrega sus propios atributos y comportamientos.

```
public class ContactoFamiliar : ElementoAgenda
{
    public string Telefono { get; set; }
    public string Relacion { get; set; }

    public ContactoFamiliar(string nombre, string telefono, string relacion)
        : base(nombre)
    {
        Telefono = telefono;
        Relacion = relacion;
    }
}

public class ContactoProfesional : ElementoAgenda
{
    public string Empresa { get; set; }
    public string Correo { get; set; }

    public ContactoProfesional(string nombre, string telefono, string empresa, s
        : base(nombre)
    {
        Empresa = empresa;
        Correo = correo;
    }
}
```

### 3.2. Polimorfismo en el Proyecto

El polimorfismo se aplica cuando se invoca el mismo método en objetos de diferentes clases, pero con comportamientos distintos según la clase de cada objeto. En este caso, si bien todos los elementos de la agenda pueden ser tratados como `ElementoAgenda`, el comportamiento de algunos métodos puede cambiar dependiendo de si el objeto es un `ContactoFamiliar`, `ContactoProfesional` o `Evento`.

```
public void ImprimirDetalle()
{
    Console.WriteLine($"Nombre: {Nombre}");
    if (this is ContactoFamiliar cf)
    {
        Console.WriteLine($"Teléfono: {cf.Telefono}, Relación: {cf.Relacion}");
    }
    else if (this is ContactoProfesional cp)
    {
        Console.WriteLine($"Empresa: {cp.Empresa}, Correo: {cp.Correo}");
    }
}
```

## 4. Abstracción

La **abstracción** es el proceso de ocultar la complejidad del sistema y mostrar solo los detalles esenciales al usuario. En este proyecto, las clases `ContactoFamiliar`, `ContactoProfesional` y `Evento` implementan abstracción al definir interfaces y métodos que interactúan con los usuarios sin exponer los detalles internos.

## 5. Encapsulamiento

El **encapsulamiento** se refiere a la ocultación de los detalles internos de un objeto, lo que permite proteger el acceso a sus datos. Esto se logra mediante el uso de modificadores de acceso, como `private`, `protected` y `public`. En este proyecto, varias propiedades de las clases están encapsuladas para asegurar que no sean modificadas directamente desde fuera de la clase.

```
public class ElementoAgenda
{
    private string _nombre;

    public string Nombre
    {
        get { return _nombre; }
        set
        {
            if (value != null && value.Length > 0)
            {
                _nombre = value;
            }
        }
    }
}
```

# 6. Conclusiones

El proyecto implementa los principios de la programación orientada a objetos de forma eficaz, utilizando clases, herencia, polimorfismo, abstracción y encapsulamiento para construir un sistema modular y fácil de mantener. Estos principios han permitido diseñar una solución flexible, donde cada componente puede ser modificado sin afectar el resto del sistema. Además, el uso adecuado de estos conceptos facilita la ampliación futura del proyecto, permitiendo agregar nuevas funcionalidades sin comprometer la estructura existente.