Parcial de patrones de diseño

Hernando José rumbo Núñez

Ingeniería de sistemas, Universidad popular del cesar

Patrones de diseño

Docente: Deivis De Jesús Martínez

Valledupar, 2022

1. **Describa la diferencia e importancia de cada uno de los pilares de la programación orientada a objetos demostrando cada caso con un ejemplo práctico (descrito y en código).**

   Abstracción: Consiste en el proceso de centrarnos en las características esenciales, importantes y relevantes de una situación o un objeto, es decir, de aquello que nos va a ayudar a dar solución al problema planteando, ignorando u omitiendo detalles que no son necesarios.
   Por ejemplo, un vehículo tiene infinidades de características como marca, modelo, color, tipo de vehículo, placa y tarifa, estos pueden ser relevantes en el contexto de un parqueadero. Sin embargo, el vehículo también tiene más características como número de puertas, número de llantas, kilometraje, entre otros, que en dicho contexto podríamos omitir.

```
public class Vehiculo
{
    0 referencias
    public string? Placa { get; set; }
    0 referencias
    public string? Marca { get; set; }
    0 referencias
    public string? Modelo { get; set; }
    0 referencias
    public double Tarifa { get; set; }
    0 referencias
    public string? TipoDeVehiculo { get; set; }
}
```

   Herencia: La herencia consiste en la creación de una clase tomando como base a otra, esto permite que la clase nueva (subclase o clase hija) obtenga los atributos y métodos no privados de la clase base (o clase padre), pero también permite a las subclases definir sus propias atributos y métodos. La herencia es muy útil porque cuando debemos crear una clase similar a una ya existente evita que creemos esa clase desde cero. Asimismo, con la herencia creamos una estructura jerárquica con clases cada vez más especializadas.

   Para el ejemplo tomaremos la clase anterior. Existen varios tipos de vehículos, pero para nuestro ejemplo tenemos dos tipos (carro y camión). Refactorizando nos quedarían las siguientes clases:

```
public class Vehiculo
{
    0 referencias
    public string? Placa { get; set; }
    0 referencias
    public string? Marca { get; set; }
    0 referencias
    public string? Modelo { get; set; }
    0 referencias
    public double Tarifa { get; set; }
}
```

Nota: Hemos eliminado el atributo “TipoDeVehiculo”, que ya no es necesario porque gracias a la herencia podemos conocer ese tipo dependiendo de la clase (carro, camión).

```
public class Camion : Vehiculo
{
    0 referencias
    public string? Carga { get; set; }
}
```

```
public class Carro : Vehiculo
{
    0 referencias
    public double DescuentoEspecial { get; set; }
}
```

Ambas subclases tienen los atributos de la clase padre (Placa, Marca, Modelo y Tarifa), de igual manera, vemos que el camión tiene un atributo adicional llamado carga (liviana, mediana y pesada), y el carro cuenta también con un atributo propio llamado “DescuentoEspecial”.

Polimorfismo: Polimorfismo significa “de muchas formas”, consiste en que las subclases pueden realizar la misma acción o responder a los mismos mensajes, pero cada una de ellas la realiza de forma distinta. Este es un concepto que está ligado a la herencia.

Para el ejemplo supongamos que debemos calcular la tarifa del parqueo y esta se realiza de la siguiente manera:

- Para los carros: $10.000 – descuento especial.
- Para los camiones:
    - $12.000 para carga liviana.
    - $18.000 para carga mediana.
    - $25.000 para carga pesada.

En código se vería así:

```
public abstract class Vehiculo
{
    0 referencias
    public string? Placa { get; set; }
    0 referencias
    public string? Marca { get; set; }
    0 referencias
    public string? Modelo { get; set; }
    0 referencias
    public double Tarifa { get; set; }

    0 referencias
    public abstract double CalcularTarifa();
}
```

Nota: Vemos la palabra “abstract” en nuestra clase vehículo, quiere decir que es una clase abstracta, lo que significa que al menos uno de sus métodos es abstracto (dice qué hacer, pero no como hacerlo), en este caso el método abstracto es “CalcularTarifa”.

```
public abstract class Vehiculo
{
    0 referencias
    public string? Placa { get; set; }
    0 referencias
    public string? Marca { get; set; }
    0 referencias
    public string? Modelo { get; set; }
    0 referencias
    public double Tarifa { get; set; }

    2 referencias
    public abstract double CalcularTarifa();
}
```

```
public class Carro : Vehiculo
{
    1 referencia
    public double DescuentoEspecial { get; set; }

    1 referencia
    public override double CalcularTarifa()
    {
        return 10000 - this.DescuentoEspecial;
    }
}
```

```
public class Camion : Vehiculo
{
    1 referencia
    public string? Carga { get; set; }

    1 referencia
    public override double CalcularTarifa()
    {
        double tarifa = 0;

        switch (this.Carga)
        {
            case "LIVIANA":
                tarifa = 12000;
                break;
            case "MEDIANA":
                tarifa = 18000;
                break;
            case "PESADA":
                tarifa = 25000;
                break;
            default:
                break;
        }

        return tarifa;
    }
}
```

Encapsulamiento: el encapsulamiento nos permite proteger la integridad interna de los datos de una clase, para así evitar que cualquiera pueda verlos o modificarlos de manera directa. Con el encapsulamiento definimos el acceso a los miembros de la clase.

Siguiendo con el ejemplo que venimos realizando, para hacer el encapsulamiento pondremos el nivel de acceso a privado de los atributos de nuestras clases.

Veamos cómo queda en el código.

```
public abstract class Vehiculo
{
    2 referencias
    private string? _placa { get; set; }
    2 referencias
    private string? _marca { get; set; }
    2 referencias
    private string? _modelo { get; set; }
    3 referencias
    private double _tarifa { get; set; }

    0 referencias
    public string? Placa {
        get => _placa;
        set { _placa = value; }
    }
    0 referencias
    public string? Marca
    {
        get => _marca;
        set { _marca = value; }
    }
    0 referencias
    public string? Modelo
    {
        get => _modelo;
        set { _modelo = value; }
    }
    0 referencias
    public double Tarifa
    {
        get => _tarifa;
        set { if (value < 0) { _tarifa = 0; }
            else { _tarifa = value; }
        }
    }

    2 referencias
    public abstract double CalcularTarifa();
}
```

```
public class Carro : Vehiculo
{
    3 referencias
    private double _descuentoEspecial { get; set; }
    1 referencia
    public double DescuentoEspecial
    {
        get => _descuentoEspecial;
        set
        {
            if (value < 0)
            {
                _descuentoEspecial = 0;
            }
            else
            {
                _descuentoEspecial = value;
            }
        }
    }

    1 referencia
    public override double CalcularTarifa()
    {
        return 10000 - this.DescuentoEspecial;
    }
}
```

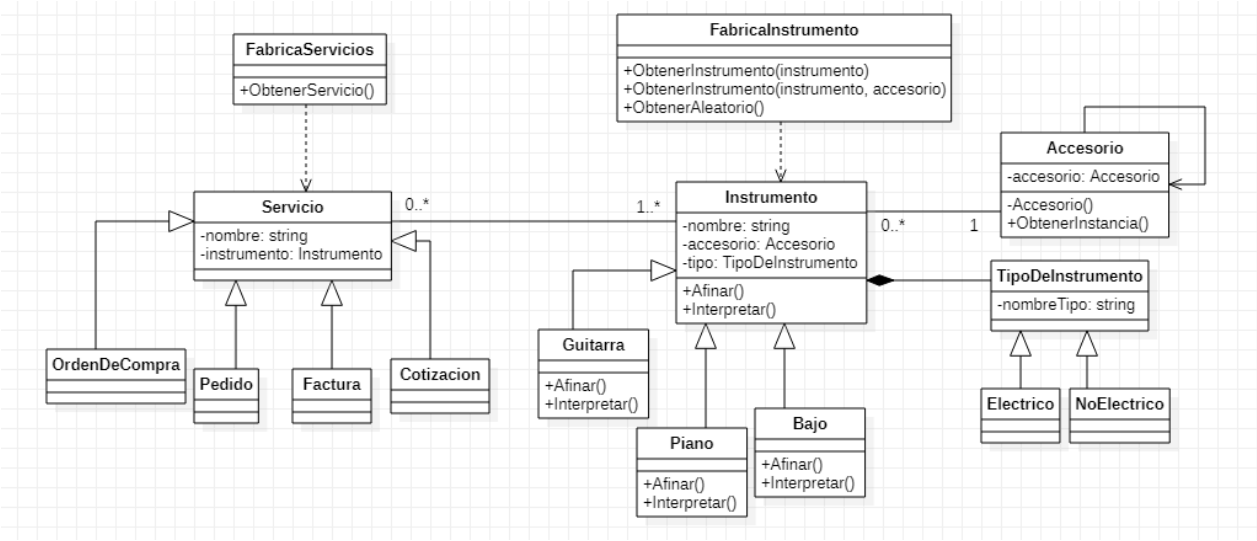
```
public class Camion : Vehiculo
{
    2 referencias
    private string? _carga { get; set; }
    1 referencia
    public string? Carga
    {
        get => _carga;
        set { _carga = value; }
    }

    1 referencia
    public override double CalcularTarifa()
    {
        double tarifa = 0;

        switch (this.Carga)
        {
            case "LIVIANA":
                tarifa = 12000;
                break;
            case "MEDIANA":
                tarifa = 18000;
                break;
            case "PESADA":
                tarifa = 25000;
                break;
            default:
                break;
        }

        return tarifa;
    }
}
```

2. Diagrama de clases de la solución:



3. **Describa tres situaciones del mundo real donde puede aplicar el patrón de diseño de software Singleton.**

   El patrón de diseño singleton se puede usar en:

   - **Conexiones a bases de datos,** singleton nos permite crear una sola conexión a una base de datos, esto ayuda a ahorrar recursos del sistema evitando sobrecargarlo con demasiadas conexiones.
   - **Colas de impresora,** si un usuario realiza una solicitud a la impresora y ya hay una instancia del objeto, quiere decir que ya esta está siendo usada, por lo que la solicitud se agrega a la cola, si no hay instancia creada, singleton procede a crearla.
   - **Registro de logs,** singleton nos ayuda a controlar que exista un único objeto capaz de registrar en el archivo de log los eventos realizados por el usuario.