UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°2:

Programación Orientada a Objetos Avanzada

Material Teórico

1° Año – 2° Cuatrimestre

Índice

PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

Introducción

Los pilares fundamentales de la POO son: Abstracción, Encapsulamiento, Herencia y Polimorfismo. Con frecuencia, la implementación real de estos principios se ve reflejada en la construcción de soluciones estándar a problemas comunes de diseño en el desarrollo de software. Es decir, como programadores seguramente nos encontramos en muchas ocasiones resolviendo un mismo problema de maneras diferentes. Con el paso del tiempo nos gustará implementar directamente la solución más escalable, testeable y reutilizable posible, entonces centraremos nuestro esfuerzo en buscar **un patrón de diseño** adecuado.

El objetivo de esta unidad es reforzar los conceptos de Herencia y Polimorfismo visto en Programación I e incorporar el uso de interfaces en C#. A modo introductorio se analizarán los patrones: Singleton, Factory y DAO, para rediseñar la solución del caso de estudio propuesto en la unidad anterior llevando nuestro desarrollo hacia un modelo en capas.

REPASO DE POLIMORFISMO

El Polimorfismo es uno de los principios fundamentales de la POO y está muy ligado con el concepto de Herencia. En términos prácticos, el polimorfismo es el mecanismo que permite (en tiempo de ejecución) que los objetos puedan responder a un mismo mensaje de diferentes maneras, dependiendo su tipo específico.

Podemos identificar tres formas diferentes de polimorfismo en la práctica, dos de los cuales fueron vistos en Programación I. Supongamos la siguiente clase:

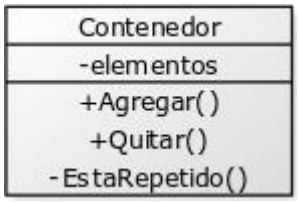

Imagen 1: Elaboración propia

Su interfaz está formada por los métodos: Quitar y Agregar. El método **EstaRepetido()** no forma parte de la interfaz de dicha clase, porque es privado. En POO decimos que la interfaz de una clase define el comportamiento de dicha clase, ya que define qué podemos y qué no podemos hacer con objetos de dicha clase.

Ahora imagine una segunda clase:

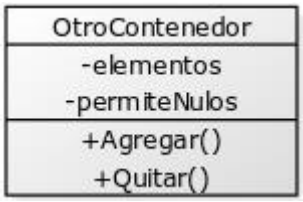

Imagen 2: Elaboración propia

¿Qué puede decir sobre ambas clases? Ambas tienen la misma interfaz. Piense que ambas clases forman parte de una solución donde existe un método en una clase cualquiera de la forma:

```
public void Actualizar (Contenedor c, Contenido elemento)
{
   int i = c.Quitar(); // quita el primer elemento no nulo
   c.Agregar(elemento); // agregar el elemento en la primer posición libre
}
```

El método recibe un *Contenedor* y opera con él. Ahora dado que las interfaces de *Contenedor* y *OtroContenedor* son iguales, se podría esperar que lo siguiente funcionase:

```
OtroContenedor oOtroContenedor = new OtroContenedor(20, false);
this.Actualizar (oOtroContenedor, new Contenido()); // suponiendo que la clase
//Contenido existe y modela los datos de los elementos incluidos
```

Pero esto no compila ya que, aunque se pueda comparar la interfaz de ambas clases, el compilador no puede hacerlo. Para el compilador *Contenedor* y *OtroContenedor* son dos clases totalmente distintas sin ninguna relación. Es decir son clases con interfaces similares pero **no son intercambiables.**

Pero se puede pensar que si ambas clases tienen una interfaz similar pueden estar perteneciendo a una jerarquía de clases con cierto comportamiento común. Entonces si aplica el concepto de Herencia:

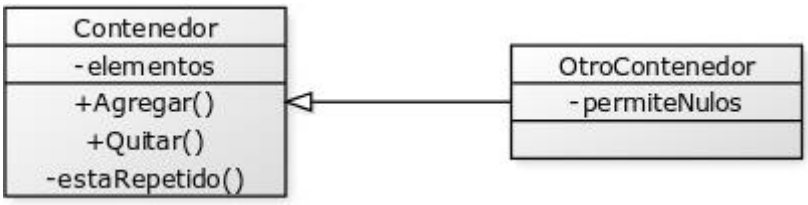

Imagen 3: Elaboración propia

Con el código resultante:

```
class Contenedor
{
	private Contenido [] elementos;
	public Contenedor (int tam)
	{
		elementos = new Contenido[tam];
	}
	public virtual void Agregar(Contenido elemento){
		if(!estaRepetido(elemento)){// valida repetidos
			for (int i = 0; i < elementos.Length; i++){
				if (elementos [i] == null){
					elementos [i] = elemento;
					break; //corta el ciclo
				}
			}
		}
	}
```

```
        public void Quitar(){
            for (int i = 0; i < elementos.Length; i++){
                if (elementos [i] != null){
                    elementos [i] = null;
                    break; //corta el ciclo
                }
            }
        }
    }

    class OtroContenedor : Contenedor
    {
        private bool permiteNulos;
        public OtroContenedor (int tam, bool permiteNulos): base(tam)
        {
            this.permiteNulos = permiteNulos;
        }
        public override void Agregar(Contenido elemento){
            if(permiteNulos){ //Si permite nulos agrega
                              //directamente.
                base.Agregar(elemento);
            }else{
                if(elemento != null){
                    base.Agregar(elemento);
                }
            }
        }
    }
```

Ahora si se escribe el mismo código anterior:

```
OtroContenedor oOtroContenedor = new OtroContenedor(20,false);
this.Actualizar (oOtroContenedor, new Contenido());
```

Si compilaría sin errores. Esto es lo que se conoce como **Polimorfismo por Herencia.** Suponga ahora que el comportamiento para quitar un elemento al contenedor no puede quedar definido en la clase base y debe necesariamente definirse en cada implementación específica (clase hija) de Contenedor. El código resulta:

```
class abstract Contenedor
{
    private Contenido [] elementos;
    public Contenedor (int tam)
    {
        elementos = new Contenido[tam];
    }
```

```
        public virtual void Agregar(Contenido elemento){
            if(!estaRepetido(elemento)){// valida repetidos
                for (int i = 0; i < elementos.Length; i++){
                    if (elementos [i] == null){
                        elementos [i] = elemento;
                        break; //corta el ciclo
                    }
                }
            }
        }
        public abstract void Quitar();
}
```

Al indicar que el método **Quitar()** es **abstract** entonces la clase Contenedor también pasa a serlo. Por consiguiente la clase OtroContenido resulta:

```
class OtroContenedor : Contenedor
{
        private bool permiteNulos;
        public OtroContenedor (int tam, bool permiteNulos): base(tam)
        {
            this.permiteNulos = permiteNulos;
        }
        public override void Agregar(Contenido elemento){
            if(permiteNulos){ //Si permite nulos agrega directamente.
                base.Agregar(elemento);
            }else{
                if(elemento != null){
                    base.Agregar(elemento);
                }
            }
        }
        public override void Quitar(){
            for (int i = 0; i < elementos.Length; i++){
                if (elementos [i] != null){
                    elementos [i] = null;
                    break; //corta el ciclo
                }
            }
        }
}
```

Cuando se sobrescribe un método abstracto de una clase base, entonces se habla de: **Polimorfismo por abstracción**. Ambos tipos de polimorfismos fueron estudiados en Programación I y son la manera más sencilla de implementarlo.

Existe un tercer tipo para el caso en que las clases tienen una interfaz similar pero no necesariamente forman parte de la misma jerarquía de clases llamado**: Polimorfismo por Interface**. Primero se verá el concepto de **interface en C#.**

INTERFACES EN C#

Las interfaces son una abstracción estupenda que ofrecen la mayor parte de los lenguajes de programación orientados a objetos. Básicamente permiten definir un **contrato** sobre el que se puede estar seguro de que, las clases que las implementen, lo van a cumplir.

El ejemplo anterior ejemplifica un caso muy común: tener dos clases que hacen lo mismo pero de diferente manera. Por ejemplo pensemos que *Contenedor* está implementado usando un arreglo fijo en memoria (como se mencionó) y *OtroContenedor* está implementado usando un fichero en disco. La funcionalidad (la interfaz) es la misma, lo que varía es la implementación. Es por ello que en POO se dice que las interfaces son funcionalidades (o comportamientos) y las clases representen implementaciones.

Ahora bien, si dos clases representan dos implementaciones distintas de la misma funcionalidad, suele ser muy útil tener la posibilidad de intercambiar las implementaciones aun cuando no forman parte de la misma jerarquía de clases. Para que dicho intercambio sea posible C# permite explicitar la **interfaz**, es decir *separar la declaración de la interfaz de su implementación* (de su clase). Para ello se usa la palabra clave **interface**:

```
interface IContenedor
{
   void Quitar();
   void Agregar(Contenido elemento);
}
```

Este código define una interfaz **IContenedor** que declara los métodos Quitar() y Agregar(). Notar que los métodos no se declaran como **public** (en una interfaz la visibilidad no tiene sentido, ya que todo es público) y que no se implementan los métodos ni se marcan como **abstract**, por defecto ya lo son.

Las interfaces son un concepto más teórico que real. No se pueden crear instancias de interfaces. El siguiente código *no compila*:

```
IContenedor c = new IContenedor();
```

Es decir las interfaces son similares a las clases abstractas pero a diferencia de éstas solo pueden contener definiciones de métodos sin implementación. Volviendo al ejemplo se puede indicar explícitamente que una clase implementa una interfaz, es decir proporciona implementación (código) a todos y cada uno de los métodos (y propiedades) declarados en la interfaz:

```
class Contenedor : IContenedor // notar que es como indicar una herencia
{
   public void Quitar() { ... }
   public void Agregar(Contendido e) { ... }
}
```

Donde la representación UML se muestra en el siguiente diagrama:

Imagen 4: Elaboración propia

La clase *Contenedor* declara explícitamente que implementa la interfaz **IContenedor**. Así pues la clase debe proporcionar implementación para todos los métodos de la interfaz. Es por esto que en POO se dice que las interfaces son **contratos** entre clases. Por último si dos clases implementan la misma **interface**, ambas clases son **intercambiables.**

Interfaces y clases abstractas

Ambos conceptos son similares y dan soporte al concepto de polimorfismo que se viene analizando. A si mismo existen ciertas diferencias que se pueden enumerar a continuación:

1. Una clase abstracta puede heredar o extender cualquier clase (independientemente de que esta sea abstracta o no), mientras que una interfaz solamente puede extender o implementar otras interfaces.
2. Una clase abstracta puede heredar de una sola clase (abstracta o no) mientras que una interfaz puede extender varias interfaces a la vez.
3. Una clase abstracta puede tener métodos que sean abstractos o que no lo sean, mientras que las interfaces sólo y exclusivamente pueden definir métodos abstractos.

4. En una clase abstracta, los métodos abstractos pueden ser public o protected. Todos los miembros de la interfaz son implícitamente públicos. No se les puede dar ningún modificador (ni siquiera public).
5. Las interfaces no pueden tener definiciones de campos, operadores, constructores, destructores o miembros estáticos.
6. *Importante*: si una clase no implementa todos los métodos definidos por una interfaz la clase puede marcarse como abstracta.

ENUMERACIONES

Al desarrollar una aplicación suele ser común encontrarse con datos que pueden tener valores específicos, es decir no van a cambiar con el tiempo. Estos pueden ser por ejemplo: el género, estado civil, los roles de usuario, opciones a ser ingresados en un menú, etc. Para estos casos existen los **enum o enumeraciones**.

Una enumeración es una clase especial que representa un grupo de constantes (variables inmutables / de solo lectura). Para crearla, se utiliza palabra clave **enum** (en lugar de class o interface) y se separan los elementos con una coma. Por ejemplo:

```
enum Nivel
{
  Bajo,
  Medio,
  Alto
}
```

Para acceder a elementos de una enumeración se utiliza la *sintaxis de punto*:

```
Nivel oNivel = Nivel.Medio;
```

Una enumeración puede ser definida dentro de otra clase, tal como se muestra en el siguiente ejemplo:

```
class Program
{
        enum Nivel
        {
                Bajo,
                Medio,
                Alto
        }
        static void Main(string[] args)
        {
```

```
                Nivel oNivel = Nivel.Bajo;
                Console.WriteLine(oNivel);
        }
}
```

De forma predeterminada, el primer elemento de una enumeración tiene el valor 0. El segundo tiene el valor 1, y así sucesivamente. Para obtener el valor entero de un elemento, debe convertir explícitamente el elemento en un **int**. Supongamos que necesitamos modelar el primer semestre del año:

```
num Meses
{
        Enero,      // 0
        Febrero,    // 1
        Marzo,      // 2
        Abril,      // 3
        Mayo,       // 4
        Junio,      // 5
        Julio       // 6
}
```

El siguiente Main() mostraría el número de ordinal asociado a un mes específico (en este caso 3-Abril):

```
static void Main(string[] args)
{
        int ordinal = (int) Meses.Abril;
        Console.WriteLine(ordinal);
}
```

También es posible asignar una valor de ordinal diferente:

```
enum Nivel
    {
        Bajo,
        Medio = 5,
        Alto
    }
```

En este caso a "Bajo" se le asignará el número **0**, a Medio se le ha asignado el número 5, por consiguiente Alto tendrá el número 6, es decir el siguiente entero de la lista.

Por último es posible utilizar las enumeraciones como casos de una estructura condicional **switch**, tal como se muestra en el siguiente programa:

```
class Program
{
        enum Nivel
        {
                Bajo,
                Medio,
                Alto
        }
        static void Main(string[] args)
        {
                Nivel val = Nivel.Medio;
                switch(val)
                {
                        case Nivel.Bajo:
                                Console.WriteLine("Nivel bajo");
                                break;
                        case Nivel.Medio:
                                Console.WriteLine("Nivel medio");
                                break;
                        case Nivel.Alto:
                                Console.WriteLine("Nivel alto");
                                break;
                }
        }
}
```

PATRONES DE DISEÑO

Como se mencionó en la introducción de la unidad, los **patrones de diseño** son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en nuestro código. En la definición de los patrones se ponen en práctica los conceptos principales de POO como Herencia, polimorfismo y Composición entre clases.

Hay una gran variedad de patrones de diseño *orientados a clases y objetos*. Los más populares son los del libro Design patterns, Elements of Reusable Object-Oriented Software, escrita por los Gang of Four (GOF), en este libro se presentan 23 patrones de diseño, divididos en las siguientes categorías:

- **Creacionales:** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente. Éstos son:
  - **Abstract Factory**
  - Builder
  - Factory Method
  - Prototype
  - **Singleton**
- **Estructurales:** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura. Éstos son:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- **De comportamiento:** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos. Éstos son:
  - Intercepter
  - Template Method
  - Chain of Responsability
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor

Otra categoría en la que suelen agruparse los patrones son aquellos *orientados a sistemas o compontes*. Estos definen estructuras de componentes y sus relaciones, por ejemplo: patrones de diseño orientado al acceso a datos, dominio o presentación. Sobre esta categoría solo se mencionará uno de los

patrones más conocidos utilizados para acceder a datos: DAO (Data Access Object) que se implementará sobre el caso de estudio de la unidad anterior.

Cabe aclarar que el objetivo es solo abordar de manera introductoria el uso de ciertos patrones para modelar una abstracción de acceso a datos sobre el caso de estudio analizado en la unidad anterior, aunque es altamente recomendable como actividad de auto-aprendizaje el análisis en profundidad de los patrones antes mencionados. El foco ahora será refactorizar la solución propuesta para el caso de estudio enfatizando el esfuerzo en la división correcta de responsabilidades entre las clases mediante un *Modelo Programación en Capas*.

MODELO DE PROGRAMACIÓN EN CAPAS

La Programación por Capas se refiere a un estilo de programación que tiene como objetivo separar responsabilidades entre las clases de tal manera que cada capa cumpla una función específica y diferente a las demás. Dentro de este estilo se destaca el *desarrollo de software a tres capas***,** el cual es una técnica ampliamente usada en el desarrollo de sistemas de información que involucren conexiones a bases de datos.

Entre las ventajas que se pueden destacar sobre el desarrollo de software en tres capas se puede citar:

- La posibilidad de reutilizar código fácilmente
- La separación de roles en tres capas hace más sencillo reemplazar o modificar a una, sin afectar a las demás.
- Poder cambiar la presentación de la aplicación sin afectar a la lógica de ni a la Base de datos.
- La capacidad de poder cambiar el motor de Base de Datos sin grandes impactos al resto del proyecto.

Esquemáticamente:

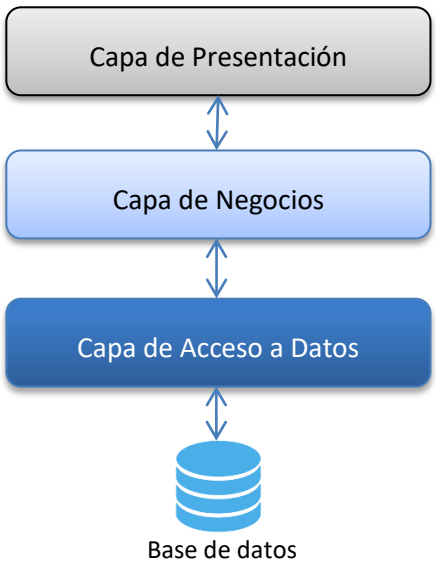

Imagen 5: Elaboración propia

La programación en tres capas está fundamentada en la POO y divide los componentes de la aplicación en las capas de:

- **Presentación:** Recibe órdenes e información del usuario, solicita servicios a la capa de lógica de negocio, y presenta los resultados de nuevo al usuario.
- **Lógica de negocio:** Es donde residen los procesos propiamente dichos de la aplicación: el manejo de las transacciones, generación listados y consultas, etc.
- **Acceso a datos:** Provee de objetos único que los programas pueden aprovechar para manipular cualquier base de datos, sin importar de qué tecnología sea o quien sea su proveedor.

Caso de estudio: Presupuestos de Carpintería en capas

Según lo expuesto en el párrafo anterior, lo primero que debiéramos revisar son las responsabilidades de las clases de la solución. *Frm_Alta_Presupuesto* es el principal componente de la capa de presentación, sin embargo si se analiza el comportamiento privado: void ProximoPresupuesto()**:**

```
private void ProximoPresupuesto()
{
        try
        {
                SqlConnection cnn = new SqlConnection();
```

```
                    cnn.ConnectionString = @"Data
                    Source=.\SQLEXPRESS;Initial
                    Catalog=carpinteria_db;Integrated Security=True";
                    cnn.Open();
                    SqlCommand cmd = new
                    SqlCommand("SP_PROXIMO_NRO_PRESUPUESTO", cnn);
                    cmd.CommandType = CommandType.StoredProcedure;
                    SqlParameter param = new SqlParameter("@next",
                    SqlDbType.Int);
                    param.Direction = ParameterDirection.Output;
                    cmd.Parameters.Add(param);
                    cmd.ExecuteNonQuery();
                    int next = Convert.ToInt32(param.Value);
                    lblNroPresupuesto.Text = "Presupuesto Nº: " +
                    next.ToString();
                    cnn.Close();
            }
        catch (Exception)
        {
                    MessageBox.Show("Error de datos", "Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
```

En este comportamiento el formulario está colaborando con los objetos de ADO.NET para poder ejecutar un procedimiento almacenado y obtener el número del próximo presupuesto. ¿Qué sucedería si por ejemplo cambia la base de datos o si cambia la lógica de obtención de este dato? Evidentemente hay que modificar el código de este método. Para evitarlo, se necesita delegar la tarea de obtener el próximo número de presupuesto a otra clase: *GestorPresupuesto*, que será el principal componente de la capa de servicios o de capa de negocio.

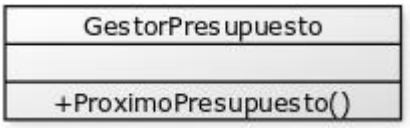

Imagen 6: Elaboración propia

Entonces si se refactoriza el código del formulario para obtener el próximo número de presupuesto, resulta:

```
private void ProximoPresupuesto()
{
        GestorPresupuesto gestor = new GestorPresupuesto();
```

```
            int next = gestor.ProximoPresupuesto();
            lblNroPresupuesto.Text = "Presupuesto Nº: " +
            next.ToString();
}
```

Donde se puede ver que el formulario solo consume los servicios proporcionados por el *objeto gestor* y muestra información relevante para el usuario.

Continuando con este criterio la clase GestorPresupuesto resulta:

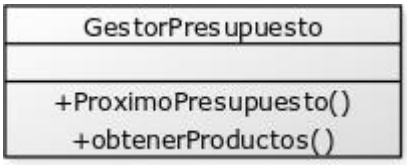

Imagen 7: Elaboración propia

Con el siguiente código:

```
class GestorPresupuesto
{
            public int ProximoPresupuesto() {
                        int nro = 0;
                        try
                        {
                                    SqlConnection cnn = new SqlConnection();
                                    cnn.ConnectionString = @"Data
                                    Source=.\SQLEXPRESS;Initial
                                     Catalog=carpinteria_db;Integrated
                                    Security=True";
                                    cnn.Open();
                                    SqlCommand cmd = new
                                    SqlCommand("SP_PROXIMO_NRO_PRESUPUESTO", cnn);
                                    cmd.CommandType = CommandType.StoredProcedure;
                                    SqlParameter param = new SqlParameter("@next",
                                    SqlDbType.Int);
                                    param.Direction = ParameterDirection.Output;
                                    cmd.Parameters.Add(param);
                                    cmd.ExecuteNonQuery();
                                    nro = Convert.ToInt32(param.Value);
                                    cnn.Close();
                        }
                        catch (Exception)
                        {
                                    //si ocurre un error se devuelve -1:
                                    nro = -1;
```

```
                }
                return nro;
        }
        public DataTable obtenerProductos()
        {
                DataTable table;
                try
                {
                        SqlConnection cnn = new SqlConnection();
                        cnn.ConnectionString = @"Data
                        Source=.\SQLEXPRESS;Initial
                        Catalog=carpinteria_db;Integrated
                        Security=True";
                        cnn.Open();
                        SqlCommand cmd = new
                        SqlCommand("SP_CONSULTAR_PRODUCTOS", cnn);
                        cmd.CommandType = CommandType.StoredProcedure;
                        table = new DataTable();
                        table.Load(cmd.ExecuteReader());
                        cnn.Close();
                }
                catch (Exception)
                {
                        //si ocurre un error se devuelve null:
                        table = null;
                }
                return table;
        }
    }
```

Solo queda refactorizar el método CargarProductos() utilizado en el constructor del formulario:

```
private void CargarProductos()
{
        GestorPresupuesto gestor = new GestorPresupuesto();
        DataTable table = gestor.obtenerProductos();
        if (table != null)
        {
                cboProductos.DataSource = table;
                cboProductos.DisplayMember = "n_producto";
                cboProductos.ValueMember = "id_producto";
         }
}
```

El formulario realiza su trabajo sin necesidad de conocer ningún detalle de implementación referido al acceso a datos, es decir, se logra desacoplar la capa de presentación de los mecanismos utilizados para guardar y obtener datos de la base.

Notar que en ambos métodos auxiliares del formulario es necesario crear un objeto GestorPresupuesto generando una dependencia a dicha clase. En general si todos los servicios requeridos por el formulario serán implementados por un objeto gestor, conviene definirlo como atributo e iniciarlo en el constructor de la ventana:

```
private GestorPresupuesto gestor;
public Frm_Alta_Presupuesto()
{
        //…
        gestor = new GestorPresupuesto();

}
```

Ahora si se analiza la clase Presupuesto, más precisamente en su comportamiento **Confirmar():**

```
public bool Confirmar()
{
        bool resultado = true;
        SqlConnection cnn = new SqlConnection();
        SqlTransaction t = null;
        try
        {
                cnn.ConnectionString = @"Data
                Source=.\SQLEXPRESS;Initial
                Catalog=carpinteria_db;Integrated Security=True";
                cnn.Open();
                t = cnn.BeginTransaction();
                SqlCommand cmd = new SqlCommand("SP_INSERTAR_MAESTRO",
                cnn, t);
                cmd.CommandType = CommandType.StoredProcedure;
          //...

        catch (Exception ex)
        {
                t.Rollback();
                resultado = false;
        }
        finally
        {
                if (cnn != null && cnn.State == ConnectionState.Open)
```

```
                cnn.Close();
            }
            return resultado;
        }
```

Nuevamente se encuentra con el problema recurrente de tener dependencias en el código a los mecanismos de persistencia provistos por .NET. Es momento entonces de pensar en alguna solución estándar: un *patrón de Diseño DAO.*

Implementando Patrón DAO

El patrón DAO (Data Access Object) o patrón de Objeto de Acceso a Datos se usa al crear una aplicación que debe mantener información (persistirla).

Permite separar el dominio de los problemas de persistencia. Generalmente usa una interfaz para definir los métodos usados para la persistencia

Esquemáticamente:

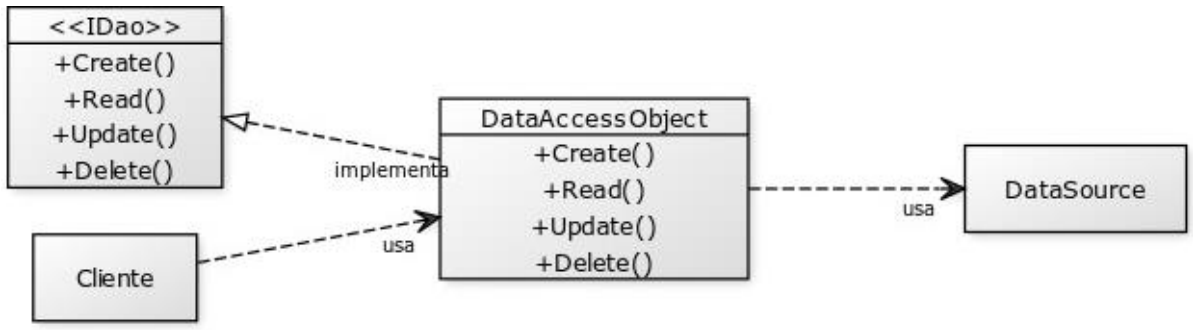

Imagen 8: Elaboración propia

Los métodos de la interfaz IDao generalmente corresponden a las funciones básicas en base de datos para: **insertar, leer, actualizar y/o borrar** una entidad en una tabla (operaciones *CRUD*). La clase **Cliente** puede ser cualquier clase que requiera de los servicios de persistencia provistos por el DAO. Generalmente se corresponde a cualquier clase de la capa de negocio.

Si se aplica este patrón a la clase Presupuesto resulta:

```
class Presupuesto
{
        public int PresupuestoNro { get; set; }
        public DateTime Fecha { get; set; }
```

```
        public string Cliente { get; set; }
        public double CostoMO { get; set; }
        public double Descuento { get; set; }
        public DateTime fechaBaja { get; set; }
        public List<DetallePresupuesto> Detalles { get; }
        public Presupuesto()
        {
                Detalles = new List<DetallePresupuesto>();
        }
        public void AgregarDetalle(DetallePresupuesto detalle)
        {
                Detalles.Add(detalle);
        }
        public void QuitarDetalle(int indice) {
                Detalles.RemoveAt(indice);
        }
        public double calcularTotal() {
                double total = 0;
                foreach (DetallePresupuesto item in Detalles)
                        total += item.CalcularSubTotal();
                return total;
        }
  }
```

Notar que como clase de dominio que es, no tiene ninguna dependencia a objetos de persistencia. Algo similar ocurre en la clase *GestorPresupuesto*:

```
class GestorPresupuesto
{
        private PresupuestoDao dao;
        public GestorPresupuesto()
        {
                dao = new PresupuestoDao();
        }
        public int ProximoPresupuesto()
        {
                return dao.ObtenerProximoNumero();
        }
        public DataTable ObtenerProductos()
        {
                return dao.ListarProductos();
        }
        public bool ConfirmarPresupuesto(Presupuesto oPresupuesto)
        {
                return dao.Crear(oPresupuesto);
        }
```

```
}
```

Como se observa esta clase de negocio tampoco necesita conocer ningún mecanismo para persistir u obtener los datos de la base, esto será responsabilidad pura y exclusiva del objeto de acceso a datos, tal como se muestra:

```
class PresupuestoDao : IPresupuestoDao
  {
      public bool Crear(Presupuesto oPresupuesto)
      {
          bool resultado = true;
          SqlConnection cnn = new SqlConnection();
          SqlTransaction t = null;

          try
          {
              cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
              Catalog=carpinteria_db;Integrated Security=True";
              cnn.Open();
              t = cnn.BeginTransaction();
              SqlCommand cmd = new SqlCommand("SP_INSERTAR_MAESTRO", cnn,
              t);
              cmd.CommandType = CommandType.StoredProcedure;
              cmd.Parameters.AddWithValue("@cliente",
              oPresupuesto.Cliente);
              cmd.Parameters.AddWithValue("@dto",
              oPresupuesto.Descuento);
              cmd.Parameters.AddWithValue("@total",
              oPresupuesto.CalcularTotal() - oPresupuesto.Descuento);
              SqlParameter param = new SqlParameter("@presupuesto_nro",
              SqlDbType.Int);
              param.Direction = ParameterDirection.Output;
              cmd.Parameters.Add(param);
              cmd.ExecuteNonQuery();
              int presupuestoNro = Convert.ToInt32(param.Value);
              int cDetalles = 1;
              foreach (DetallePresupuesto det in oPresupuesto.Detalles)
              {
                  SqlCommand cmdDet = new
                  SqlCommand("SP_INSERTAR_DETALLE", cnn, t);
                  cmdDet.CommandType = CommandType.StoredProcedure;
                  cmdDet.Parameters.AddWithValue("@presupuesto_nro",
                  presupuestoNro);
                  cmdDet.Parameters.AddWithValue("@detalle", cDetalles);
                  cmdDet.Parameters.AddWithValue("@id_producto",
                  det.Producto.ProductoNro);
```

```
                cmdDet.Parameters.AddWithValue("@cantidad",
                det.Cantidad);
                cmdDet.ExecuteNonQuery();
                cDetalles++;
            }
            t.Commit();
        }
        catch (Exception)
        {
            t.Rollback();
            resultado = false;
        }
        finally
        {
            if (cnn != null && cnn.State == ConnectionState.Open)
                cnn.Close();
        }
        return resultado;
    }

    public int ObtenerProximoNumero()
    {
        int nro = 0;
        try
        {
            SqlConnection cnn =
            ConexionGlobal.ObtenerInstancia().ObtenerConexion();
            cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
            Catalog=carpinteria_db;Integrated Security=True";
            cnn.Open();
            SqlCommand cmd = new
            SqlCommand("SP_PROXIMO_NRO_PRESUPUESTO", cnn);
            cmd.CommandType = CommandType.StoredProcedure;
            SqlParameter param = new SqlParameter("@next",
            SqlDbType.Int);
            param.Direction = ParameterDirection.Output;
            cmd.Parameters.Add(param);
            cmd.ExecuteNonQuery();
            nro = Convert.ToInt32(param.Value);
            cnn.Close();
        }
        catch (Exception e)
        {
            //si ocurre un error se devuelve -1:
            nro = -1;
```

```
            }
            return nro;
        }

        public DataTable ListarProductos()
        {
            DataTable table;
            try
            {
                SqlConnection cnn = new SqlConnection();
                cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
                Catalog=carpinteria_db;Integrated Security=True";
                cnn.Open();
                SqlCommand cmd = new SqlCommand("SP_CONSULTAR_PRODUCTOS",
                cnn);
                cmd.CommandType = CommandType.StoredProcedure;
                table = new DataTable();
                table.Load(cmd.ExecuteReader());
                cnn.Close();
            }
            catch (Exception)
            {
                table = null;
            }
            return table;
        }
    }
}
```

Un detalle no menos importante surge a preguntarse qué sucedería si por ejemplo necesitamos una implementación diferente del objeto DAO. Al crear una instancia específica de la clase PresupuestoDao en el constructor, la clase de negocio queda acoplada con esta implementación. Para resolver esta situación de creación específica de una instancia se recurre a otro patrón: Factory (la fábrica).

Implementando Patrón Factory

El patrón *Factory Method* o simplemente *Factory* define un método que debe utilizarse para crear objetos, en lugar de una llamada directa al constructor (operador new). Las subclases pueden sobrescribir este método para cambiar las clases de los objetos que se crearán.

En clases:

Imagen 9: Elaboración propia

Siendo *Creador* una clase abstracta con un método abstracto que luego se redefine en la clase hija (*CreadorConcreto*) con la creación específica del objeto a instanciar.

Resultando:

```
abstract class AbstractDaoFactory
{
        public abstract IPresupuestoDao CrearPresupuestoDao();
}
class DaoFactory : AbstractDaoFactory
{
        public override IPresupuestoDao CrearPresupuestoDao()
        {
                return new PresupuestoDao();
        }
}
```

Finalmente si se refactoriza el constructor de la clase *GestorPresupuesto*, que es quien necesita crear un objeto Dao específico, resulta:

```
public GestorPresupuesto(AbstractDaoFactory factory)
{
        dao = factory.CrearPresupuestoDao();
}
```

Notar que ahora se recibe como argumento un objeto **factory abstracto**. Se elimina la dependencia derivada de la creación del objeto DAO específico. Solo con recibir un objeto **factory concreto** es posible crear un DAO específico. Esto hace que el código sea más flexible y fácil de mantener antes futuros cambios.

Otra alternativa sería recibir directamente una interface *IDaoFactory*, en lugar de una clase abstracta. La diferencia en este punto radica en que se puede tener más comportamiento definido en la clase abstracta que no necesariamente sea abstracto.

Implementando Patrón Singleton

Por último se analiazará el siguiente fragmento de código presente en los métodos de la clase PresupuestoDao:

```
SqlConnection cnn = new SqlConnection();
cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
Catalog=carpinteria_db;Integrated Security=True";
cnn.Open();
//crear el comando, ejecutarlo y obtener los resultados:
//…
Cnn.Close();
```

En la implementación de cada método del objeto Dao siempre es necesario ejecutar los siguientes pasos:

1. Crear primero una conexión a la base de datos
2. Abrir la conexión
3. Ejecutar un comando SQL
4. Procesar la respuesta.
5. Cerrar la conexión

Una pregunta interesante es: ¿Los objetos DAO podrían delegar esta responsabilidad a un **único objeto** encargado de ejecutar los comandos? Se puede pensar entonces en una clase auxiliar que conozca los parámetros para conectarse a la fuente de datos (que generalmente son los mismos) y que ejecute los pasos antes mencionados cada vez que el objeto DAO necesita realizar una operación con la base de datos. Notar que si de esta clase solo es necesario tener una instancia, entonces se puede pensar en otro patrón creacional muy utilizado en la ingeniería de software: *patrón Singleton***.**

El *Singleton* es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer **privado el constructor** por defecto para evitar que otros objetos utilicen el operador **new** con la clase Singleton.
- Crear un **método de creación estático** que actúe como constructor. Este método invoca al constructor privado para crear un objeto y lo guarda en un

campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si el código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto. En clases:

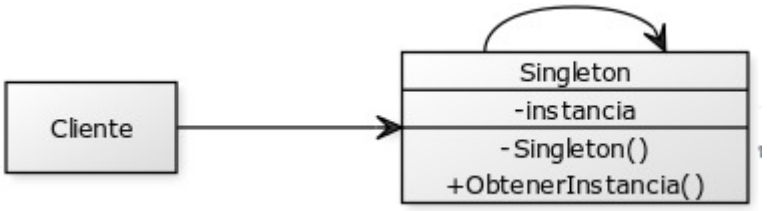

Imagen 10: Elaboración propia

Ejemplificando para el dominio analizado, se define la clase **HelperDAO** con las siguientes líneas de código:

```
class HelperDao
{
        private static HelperDao instancia;
        private string connectionString;
        private HelperDao () {
                connectionString = @"Data
                Source=.\SQLEXPRESS;Initial
                Catalog=carpinteria_db;Integrated Security=True";
        }
        public static HelperDao ObtenerInstancia()
        {
                if (instancia == null) {
                        instancia = new HelperDao();
                }
                return instancia;
        }
        public DataTable ConsultaSQL(string storeName) {
                SqlConnection cnn = new SqlConnection();
                SqlCommand cmd = new SqlCommand();
                DataTable tabla = new DataTable();
                try{
                        cnn.ConnectionString = string_conexion;
                        cnn.Open();
                        cmd.Connection = cnn;
                        cmd.CommandType = CommandType.StoredProcedure;
                        cmd.CommandText = storeName;
                        tabla.Load(cmd.ExecuteReader());
                        return tabla;
```

```
                }catch(SqlException ex){
                        throw(ex);
                }finally{
                        this.CloseConnection(cnn); //método privado que
                        //cierra la conexión siempre que no sea nula y
                        //se encuentre abierta.
                }
        }
}
```

Entonces podemos utilizarla desde el objeto Dao de la siguiente manera:

```
public DataTable ListarProductos()
{
        HelperDAO helper = HelperDAO.ObtenerInstancia();
        return helper.ConsultaSQL("SP_CONSULTAR_PRODUCTOS");
}
```

De esta forma se garantiza que existe un único objeto HelperDao con el que se accede a la base de datos para ejecutar sentencias SQL. Solo resta completar la clase HelperDao con los métodos para ejecutar procedimientos con parámetros de entrada y salida. Es altamente recomendable usarlo como actividad de auto-aprendizaje por parte de los estudiantes.

Por último tenga presente evitar siempre cadenas constantes en el código, como el caso de la cadena de conexión a la base, ya que si por alguna razón cambia es necesario compilar nuevamente la solución. Para evitarlo podríamos utilizar un el archivo de configuración de la aplicación y crear un recurso con el valor de la cadena de conexión para luego tomar ese valor en tiempo de ejecución.

BIBLIOGRAFÍA

Gamma E., Helm R., Johnson R. y Vlissides J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

Microsoft. Guía de programación de C#. Recuperado de: https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/

Refactoring.Guru. Patrones de diseño. El catálogo de ejemplos en C#. Recuperado de: https://refactoring.guru/es/design-patterns/csharp