UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°1:

Programación Orientada a Objetos Avanzada

Material Teórico

1° Año – 2° Cuatrimestre

Índice

PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

Introducción

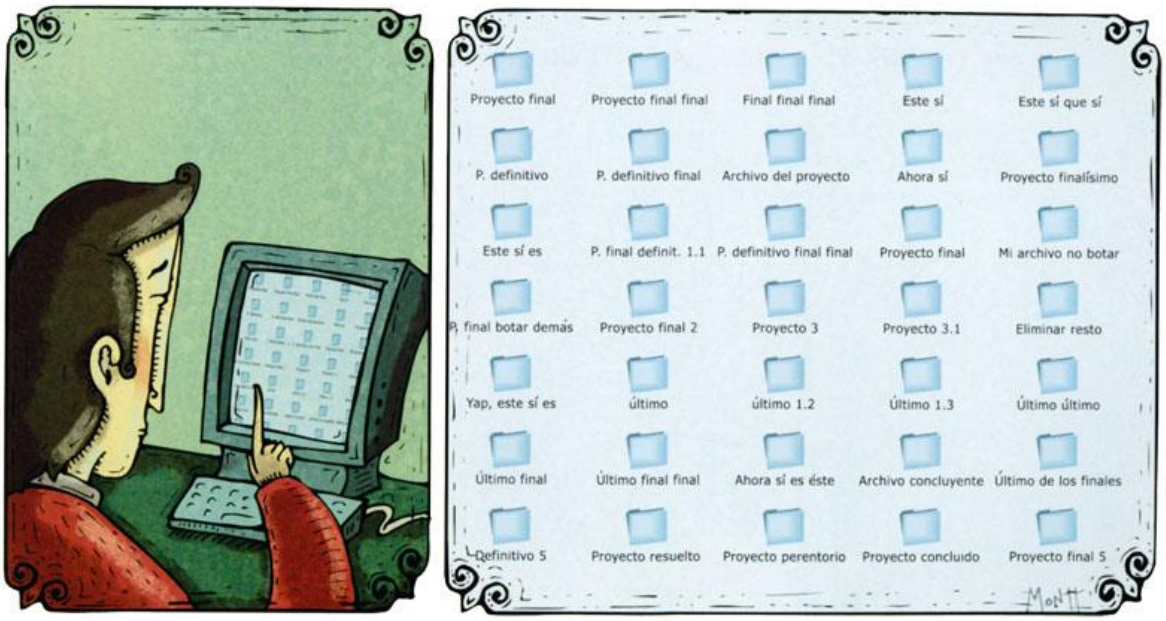
Los pilares fundamentales de la POO son: Abstracción, Encapsulamiento, Herencia y Polimorfismo. Con frecuencia, la implementación real de estos principios se ve reflejada en la construcción de soluciones estándar a problemas comunes de diseño en el desarrollo de software. Es decir, como programadores 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 Repository, para construir una abstracción de datos diseñada con estos patrones.

Antes de adentrar en el desarrollo de esta unidad, es fundamental comprender el concepto de versionado de código a través de Git. Este sistema de control de versiones se ha convertido en una herramienta indispensable para gestionar cambios en el código de manera eficiente y colaborativa. A través de Git, los desarrolladores pueden registrar y rastrear modificaciones, facilitando la colaboración en proyectos de software y asegurando un flujo de trabajo organizado y controlado.

INTRODUCCIÓN AL CONTROL DE VERSIONES

En cualquier proyecto de software colaborativo, integrar cambios realizados por múltiples programadores puede ser un desafío considerable. La práctica común de copiar y pegar código enviado por otros a menudo conduce a proyectos rotos y caos de versiones, como ilustra en la siguiente imagen.



La solución a este problema es la utilización de un sistema de control de versiones, **VCS** por sus siglas en inglés (Version Control System). La idea central de un VCS es almacenar todos los cambios que se realizaron sobre un conjunto de archivos, generando así un historial de cambios, donde se puede ver cada modificación realizada en los archivos, y llegado el caso volver a una versión previa.

Hay una infinidad de VCS diferentes, con diferentes características que los diferencian, por ejemplo, por nombrar algunos: SVN, Mercurial, Team Foundation o **Git** (que será la herramienta que se usará en esta asignatura). En todo sistema de versionamiento hay varios conceptos básicos que son comunes:

- **Commit**: Detalle de un cambio específico que se realizó a uno o más archivos bajo control de versiones. Tiene un autor asociado, una fecha y un comentario donde generalmente se describe que se cambió.
- **Repositorio**: Lugar donde se almacenan todo el historial de cambios. Al trabajar con varios desarrolladores generalmente hay un repositorio compartido donde los diferentes programadores van subiendo sus cambios.

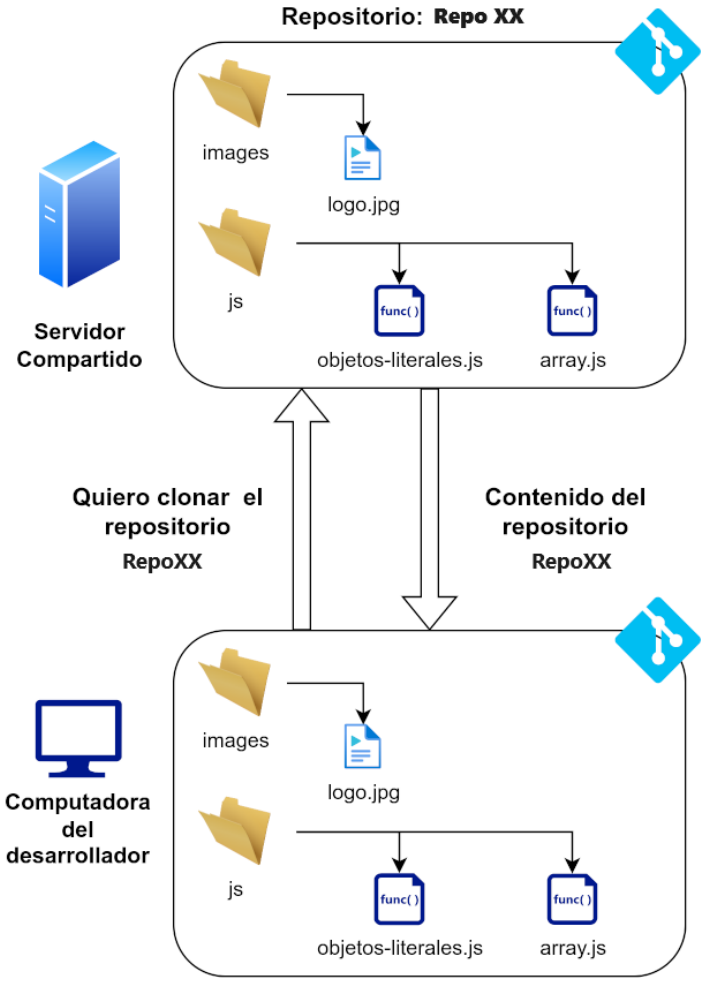
- **Copia de trabajo**: Copia de los archivos bajo control de versiones, generalmente una carpeta, donde un programador hace los cambios en los archivos del proyecto y desde donde se generan los parches que van evolucionando el proyecto.

Git

Git es un proyecto de código abierto maduro y con un mantenimiento activo.

Como se mencionó anteriormente, un VCS (en este caso Git) permite almacenar todos los cambios realizados sobre un conjunto de archivos manteniendo su historial y permitiendo ver (o restaurar) las versiones previas de estos archivos. No existen restricciones respecto al tipo de archivos con los que Git puede trabajar.

Teniendo en mente cual es el rol que cumple Git en el trabajo diario de un programador el próximo paso será comprender algunos conceptos claves para su utilización. Típicamente, cuando un programador participe en un proyecto de desarrollo de software descargará (*clonará*) desde el servidor compartido el repositorio con los archivos del mismo, generando una copia local sobre la que trabajará. Este escenario se describe en la siguiente figura:



Una vez que el programador haya descargado (*clonado*) el repositorio trabajará sobre esta copia local agregando, modificando y eliminando archivos. Es de importancia aclarar una vez más que estas modificaciones el programador las estará realizando sobre su copia local.

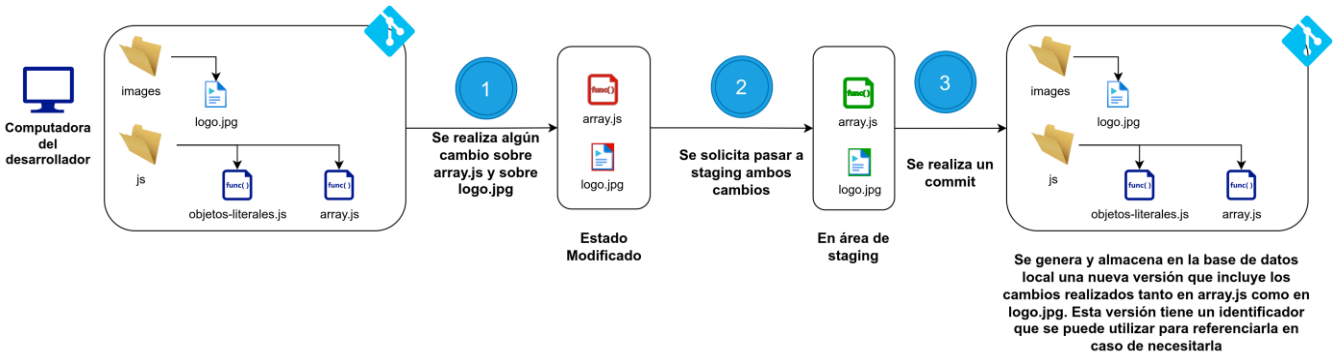
¿Cómo funcionará, entonces, el control de versiones en Git?, ¿Cada vez que se modifique un archivo (por mínimo que sea el cambio) ¿Se generará una nueva versión?, ¿Cuando se genere una nueva versión, se comunicará automáticamente el cambio al servidor compartido?. Para responder estas preguntas debe entenderse que para Git los archivos estarán en uno de tres estados posibles: *modificado*, *en staging (staged)*, *confirmado (commited)*.

Modificado: El archivo estará en este estado si, luego de haber sido descargado, se modificó. Por ejemplo, se clonó el proyecto *ProyectoDDS*, se abrió con un editor *array.js* y se modificó una línea de código y esta modificación se guardó. Hasta este momento, para Git el archivo está en estado *modificado*, no se confirmó todavía este cambio, no se lo asoció a un autor, ni se guardó este cambio en el historial.

Staged: El archivo estará en este estado si el programador decidió agregar sus modificaciones para que sean confirmadas en el próximo *commit*. Por ejemplo, el programador decide que la modificación al archivo *array.js* es correcta e indicó a Git (de una manera que se detallará más adelante) que este cambio formará parte del próximo *commit* que realice. Debe notarse que si se realizó un cambio sobre algún archivo y este cambio no fué pasado al área de staging el mismo **no** se incluirá en el próximo commit.

Commited: El archivo estará en este estado si el programador confirmó el cambio sobre el mismo (realizó un *commit*). Cuando un archivo pasa a este estado Git creará en su base de datos la nueva versión del archivo, registrará el autor del cambio y el comentario que el mismo ingresó, además de otras varias acciones. Nuevamente, estas acciones ocurren sobre la **copia local** del repositorio y **no** se comunican al servidor compartido. Para enviar los cambios realizados en un *commit* al servidor compartido, debe realizarse un paso adicional que se analizará más adelante.

La siguiente figura muestra un ejemplo de transición entre estos estados:



Paso a paso por línea de comandos

A continuación, se enumeran los pasos necesarios para empezar a trabajar con un repositorio de código remoto utilizando **Git** como herramienta de versionado

Paso 0: Instalación y configuración inicial:

1. **Instalación de Git:** Si Git no está instalado en tu sistema, puedes descargarlo e instalarlo desde su sitio web oficial: https://git-scm.com/downloads.
2. **Crear una cuenta en GitHub (o GitLab):** Si aún no tienes una cuenta, regístrate en GitHub en *github.com* o en GitLab en *gitlab.com*.

1. **Configuración inicial de Git:** Antes de comenzar a trabajar con Git, es recomendable configurar tus credenciales (nombre de usuario y correo electrónico) para que los commits reflejen correctamente la autoría de los cambios. Esto se realiza una sola vez después de instalar Git:

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu_correo@example.com"
```

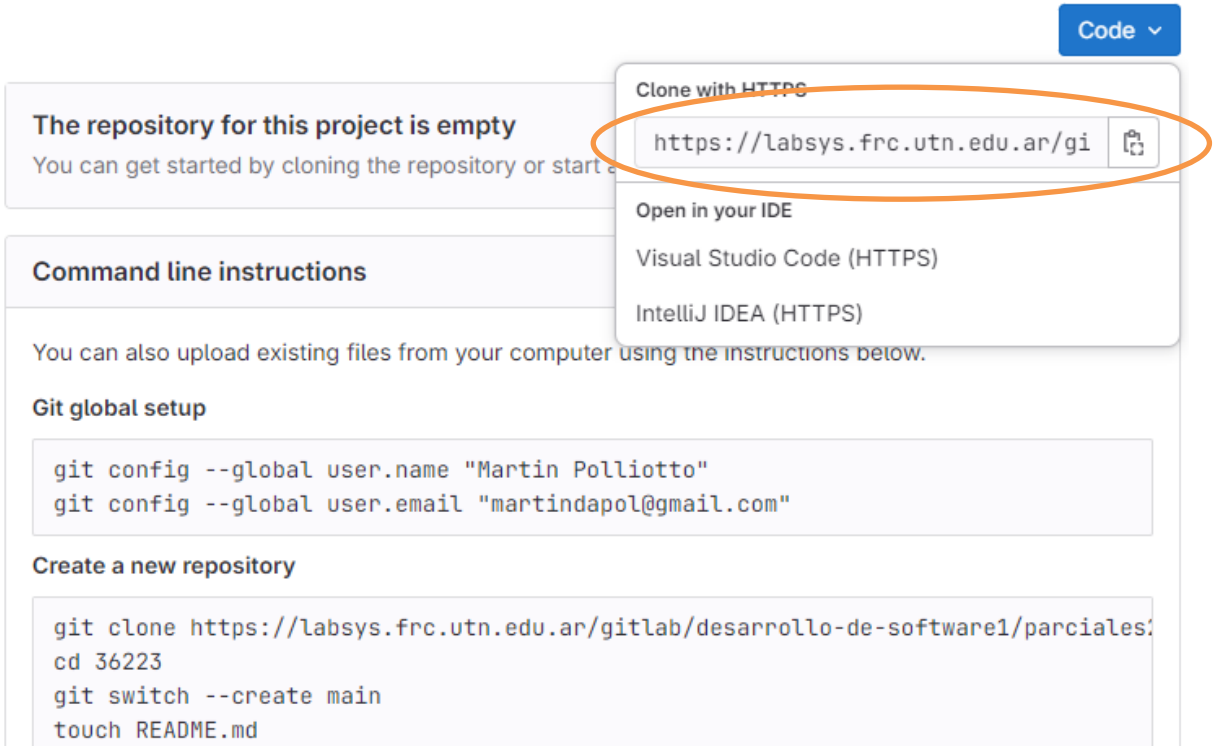
Reemplaza "Tu Nombre" con tu nombre y "tu_correo@example.com" con tu dirección de correo electrónico asociada a tu cuenta de GitHub o GitLab.

Paso 1: Clonar un repositorio:

Para comenzar a trabajar con un repositorio Git existente (por ejemplo, llamado RepoXX), se clona localmente a tu máquina usando el siguiente comando en la consola:

```
git clone <url_del_repositorio>
```

Donde <url_del_repositorio> es la URL del repositorio que deseas clonar. Para obtener la URL es necesario navegar en GitHub/Gitlab el repositorio remoto, tal como se indica en la siguiente figura:



Generalmente la URL está formada por https://<nombre_usuario_o_dominio>/<nombre_repositorio >.git

Paso 2: Realizar cambios y confirmarlos:

Después de clonar el repositorio, realiza cambios en los archivos del proyecto. Para ver el estado de los archivos modificados y/o nuevos, utiliza:

```
git status
```

A continuación, agrega los archivos modificados al área de preparación para ser confirmados:

```
git add nombre_del_archivo.txt
```

O para agregar todos los archivos modificados:

```
git add . (punto)
```

Paso 3: Confirmar los cambios:

Después de agregar los archivos al área de preparación, confirma los cambios con un mensaje descriptivo:

```
git commit -m "Mensaje descriptivo de los cambios realizados"
```

Reemplaza "Mensaje descriptivo de los cambios realizados" con un mensaje que describa los cambios realizados de manera clara y concisa.

Paso 4: Hacer push de los cambios:

Finalmente, para enviar los cambios confirmados al repositorio remoto (por ejemplo, origin), utiliza el comando:

```
git push origin master
```

Donde **origin** es el nombre del control remoto y **master** es la rama principal donde se están empujando los cambios. Asegúrate de tener los permisos adecuados para realizar esta operación en el repositorio remoto.

Git en sí es una utilidad de consola, que se utiliza escribiendo comandos a través de una línea de comandos (cmd o git bash en Windows, una terminal en Mac o Linux). Si bien hay una gran cantidad de herramientas gráficas que permiten manejar un repositorio, o integración de Git en los propios IDE, en última instancia estas herramientas terminan ejecutando los comandos de Git por detrás. Se puede mencionar como ejemplo GitHub Desktop (disponibles para Windows o Mac). Otra alternativa es utilizar las herramientas gráficas que se integran con el IDE de programación.

¿Qué es gitignore?

Cuando se trabaja en un repositorio local, Git observa cada archivo y lo considera de tres maneras:

- Seguimiento: ya ha preparado o confirmado el archivo.
- Sin seguimiento: no ha organizado ni comprometido.
- Ignorado: Le has dicho explícitamente a Git que ignore los archivos.

El archivo .gitignore le dice a Git qué archivos ignorar al enviar su proyecto al repositorio remoto. Se encuentra en el directorio raíz del repositorio.

El archivo .gitignore en sí es un documento de texto sin formato. Aquí hay un ejemplo de archivo .gitignore:

```
# Ignorar archivos de configuración específicos para el entorno local
config.ini

# Ignorar archivos de logs y bases de datos
*.log
*.sqlite

# Ignorar archivos compilados y binarios
*.exe
*.dll
*.so
*.o

# Ignorar directorios generados automáticamente
node_modules/
vendor/

# Ignorar archivos de compilación y dependencias específicos de IDEs
.idea/
.vscode/

# Ignorar archivos de configuración sensibles con contraseñas o tokens
.env

# Ignorar archivos temporales o de backup
*.bak
*.tmp
```

Ramas y conflictos

Git es una herramienta poderosa para el control de versiones que permite a los equipos de desarrollo gestionar cambios de manera eficiente. Dos conceptos clave en Git son el manejo de ramas y la resolución de conflictos. Las ramas permiten trabajar en paralelo en diferentes funcionalidades o versiones del proyecto, mientras que la resolución de conflictos es necesaria cuando los cambios en una rama entran en conflicto con los cambios en otra. A continuación, se presenta un paso a paso avanzado que cubre estas operaciones:

Paso 1: Clonar un repositorio y crear una nueva rama

Clonar un repositorio: Clona un repositorio Git existente a tu máquina local:

```
git clone <url_del_repositorio>
cd nombre_del_repositorio
```

Donde <url_del_repositorio> es la URL del repositorio que deseas clonar.

Crear y cambiar a una nueva rama: Crea una nueva rama para trabajar en una nueva funcionalidad o arreglo de bugs:

```
git checkout -b nueva-rama
```

Esto crea una nueva rama llamada nueva-rama y automáticamente te cambia a ella.

Paso 2: Realizar cambios y confirmarlos en la nueva rama

Realizar cambios: Realiza cambios en los archivos del proyecto para implementar la nueva funcionalidad o arreglo de bugs.

Agregar y confirmar cambios: Agrega los archivos modificados al área de preparación y confirma los cambios en la nueva rama:

```
git add .
git commit -m "Implementar nueva funcionalidad X"
```

Paso 3: Cambiar de rama y realizar más cambios

Cambiar a otra rama: Cambia a otra rama existente para realizar otro conjunto de cambios:

```
git checkout otra-rama
```

Donde otra-rama es el nombre de la rama a la que deseas cambiar.

Realizar cambios y confirmarlos: Realiza cambios en esta nueva rama y confírmalos como se hizo en el Paso 2.

Paso 4: Fusionar ramas y resolver conflictos

Fusionar cambios: Fusiona los cambios de una rama en otra (por ejemplo, fusionar nueva-rama en otra-rama):

```
git checkout otra-rama
git merge nueva-rama
```

Esto combinará los cambios de nueva-rama en otra-rama.

Si hay conflictos entre los cambios realizados en ambas ramas, Git te pedirá que los resuelvas manualmente. Abre los archivos con conflictos en un editor de texto y modifica las secciones conflictivas marcadas por Git. Una vez resueltos los conflictos, marca los archivos como resueltos:

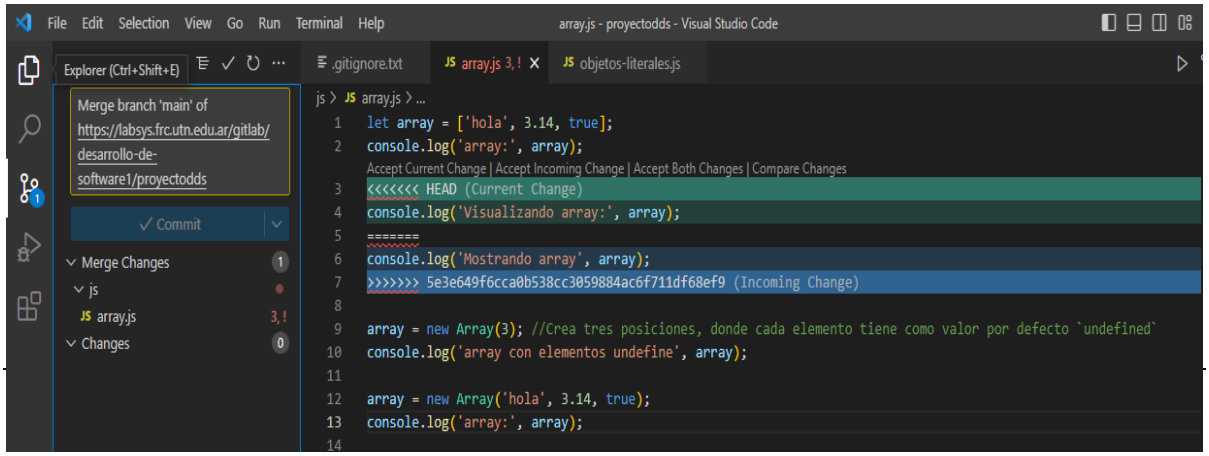
```
git add archivo_conflictivo.txt
```

Continúa con el proceso de fusión con:

```
git commit
```

Esto finaliza la fusión de las ramas y completa el proceso de resolución de conflictos.

Para ejemplificar dos miembros del repositorio modificaron la misma línea en el



archivo array.js.

Un usuario escribió "Mostrando array" y otro "Visualizando array". Git no sabe qué cambio dejar y por eso se genera un conflicto. El marcador "<<<<<<<<<<< HEAD >>>>>>>>>>>" indica donde comienza y termina el conflicto.

Para corregirlo es preciso que te comuniques con el otro colaborador y decidan qué linea dejar. Manualmente se quita el marcador "<<<<<<<<<<< HEAD >>>>>>>>>>>" y se deja la línea con el cambio final.

Para que este cambio sea efectivo se debe ejecutar: git pull (traerse cambios) editar array.js (quitar marcas y dejar la línea definitiva) git add "array.js" git commmit -m "resolviendo conflicto" git push (enviar cambios)

Paso 5: Enviar cambios a repositorio remoto

Enviar cambios a repositorio remoto: Finalmente, envía los cambios confirmados a tu repositorio remoto:

```
git push origin otra-rama
```

Donde otra-rama es la rama donde has realizado los cambios y deseas enviarlos al repositorio remoto (origin).

Este proceso avanzado de Git no solo facilita el trabajo colaborativo en proyectos, sino que también permite manejar eficazmente diferentes versiones y funcionalidades mediante el uso de ramas y la habilidad para resolver conflictos cuando surgen cambios simultáneos en el código.

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:

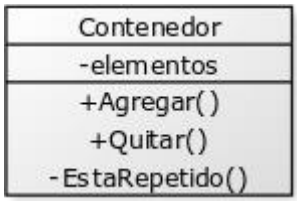


Imagen 1: Elaboración propia

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:

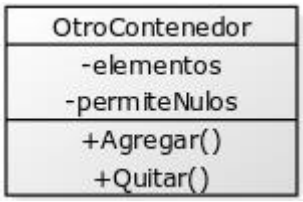


Imagen 2: Elaboración propia

¿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:

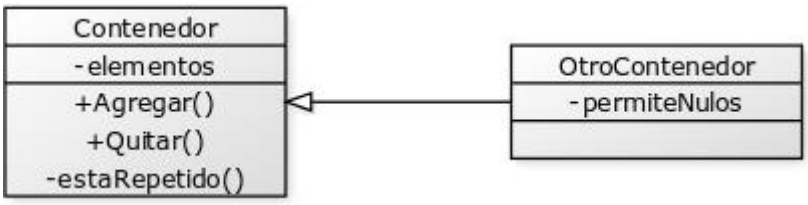


Imagen 3: Elaboración propia

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;
                }
        }
}
```

CLASES ESTÁTICAS

En C#, la palabra reservada **static** se utiliza para declarar clases, métodos o atributos que pertenecen a la clase en sí misma en lugar de a una instancia específica de una clase (objeto).

Clases estáticas

Una clase estática solo puede contener miembros estáticos y no se pueden instanciar. Se utilizan principalmente para agrupar métodos y atributos relacionados que no dependen de la instancia de la clase para su funcionamiento. Por ejemplo:

```
public static class Utilidades
{
    public static void MostrarMensaje(string mensaje)
```

```
        {
            Console.WriteLine($"Mensaje: {mensaje}");
        }
    }
```

En este ejemplo, Utilidades es una clase estática que contiene el método estático MostrarMensaje. Lo más interesante es que este método se puede llamar sin crear una instancia de Utilidades, como `Utilidades.MostrarMensaje("Hola mundo");`.

Métodos estáticos

Los métodos estáticos pertenecen a la clase en lugar de a las instancias individuales de la clase. Estos métodos se pueden invocar directamente a través del nombre de la clase sin necesidad de crear un objeto. Por ejemplo:

```
public class Calculadora
{
    public static int Sumar(int a, int b)
    {
        return a + b;
    }
}
```

En este caso, **Sumar** es un método estático de la clase Calculadora. Puedes usarlo así: `int resultado = Calculadora.Sumar(5, 3);`.

Atributos estáticos

Los atributos estáticos pertenecen a la clase en lugar de a las instancias individuales. Esto significa que el valor de un atributo estático es compartido entre todas las instancias de la clase. Por ejemplo:

```
public class Contador

{
    public static int contador = 0;

    public Contador()
    {
        contador++;
    }
}
```

En este ejemplo, **contador** es un atributo estático de la clase Contador. Cada vez que se crea una nueva instancia de Contador, el valor de contador se incrementa en uno. Por lo que es un atributo común a todas las instancias de la clase Contador.

Métodos estáticos en clases no estáticas

Cuando defines un método como static dentro de una clase que no es estática, significa que el método pertenece a la clase en sí misma y no a las instancias individuales de esa clase. Algunas restricciones y consideraciones son:

1. *Acceso a miembros no estáticos*: Dentro de un método estático no se puede acceder directamente a miembros no estáticos de la clase. Esto se debe a que un método estático se puede llamar sin una instancia de la clase, por lo tanto, no tiene acceso al estado de ninguna instancia específica.

```
public class Persona
{
    public string Nombre { get; set; }
    public static void Saludar()
    {
        // No se puede acceder a this.Nombre porque Nombre no es estático
        Console.WriteLine("¡Hola!");
    }
}
```

2. *Llamadas a métodos estáticos:* Desde métodos no estáticos de la misma clase, puedes llamar a métodos estáticos directamente usando el nombre de la clase.

```
public class Persona
{
    public static void Saludar()
    {
```

```
            Console.WriteLine("¡Hola!");
        }
        public void Presentarse()
        {
            // Llamada a un método estático desde un método no estático
            Persona.Saludar();
        }
    }
```

3. *Herencia y polimorfismo:* Los métodos estáticos no se pueden sobrescribir ni implementar en clases derivadas usando la palabra clave override. Se pueden ocultar utilizando la misma palabra clave static en la clase derivada, pero esto no es un comportamiento polimórfico.

En resumen, al usar la palabra clave **static** en C#, tener en cuenta las restricciones y comportamientos específicos que afectan el acceso a miembros no estáticos, la compartición de datos entre instancias y la capacidad de herencia y polimorfismo en el caso de métodos estáticos en clases no estáticas.

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:

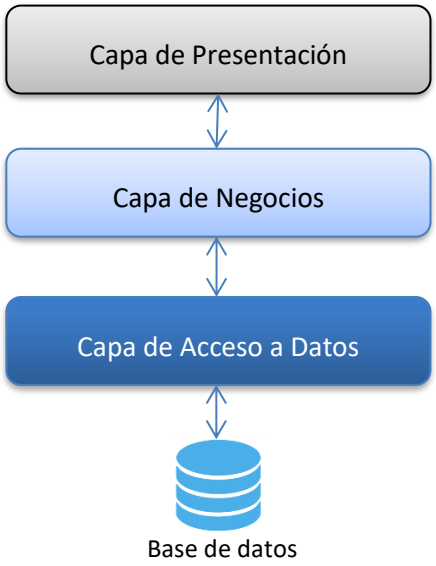


Imagen 5: Elaboración propia

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.

Ejemplo de aplicación en capas

Suponer la siguiente situación: existe cierta aplicación informática que permite registrar presupuestos de una carpintería metálica. En la solución *Frm_Alta_Presupuesto* es el principal componente de la capa de presentación, si se analiza el comportamiento del siguiente método privado:

```
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);
        }
    }
```

Se observa que 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.

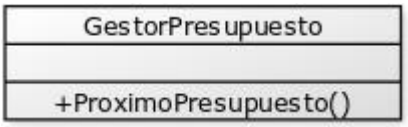


Imagen 6: Elaboración propia

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:

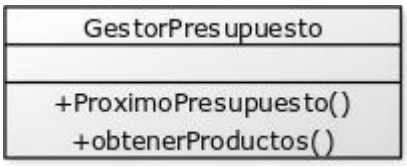


Imagen 7: Elaboración propia

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 además se cuenta con la clase Presupuesto y un comportamiento **Confirmar()**, tal como se muestra a continuación**:**

```
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 Repository.*

PATRÓN REPOSITORY

Definición: El patrón Repository es un patrón de diseño que se utiliza comúnmente en el desarrollo de software para abstraer y encapsular la lógica de acceso a objetos de dominio. Proporciona una interfaz uniforme para acceder a los datos almacenados en diferentes fuentes de datos, como bases de datos (relacionales o no relacionales), servicios web, APIs, archivos, etc.

Problemática: En aplicaciones sin una estructura adecuada, el acceso directo a los datos desde múltiples partes del código puede llevar a un acoplamiento fuerte y dificultar el mantenimiento y la evolución del sistema. Además, las operaciones de acceso a datos (como consultas SQL) suelen estar dispersas por el código, dificultando la gestión y la optimización de las consultas.

Solución: El patrón Repository aborda estas preocupaciones proporcionando un punto centralizado de acceso a datos. Define una capa intermedia entre la lógica de negocio de la aplicación y los detalles de la implementación de almacenamiento de datos. Esto permite desacoplar la lógica de negocio de los detalles de implementación específicos de la fuente de datos, mejorando así la modularidad y la mantenibilidad del código.

Diferencia con el patrón DAO:

- DAO (Data Access Object): Este patrón se centra en proporcionar una interfaz abstracta para acceder a los datos sin exponer los detalles específicos de la fuente de datos. En DAO, cada entidad (como una tabla de base de datos) tiene su propio DAO que proporciona métodos CRUD (Crear, Leer, Actualizar, Eliminar).
- Repository: Aunque ambos patrones tienen similitudes en su objetivo de abstracción del acceso a datos, el patrón Repository generalmente se enfoca en proporcionar operaciones de alto nivel relacionadas con agregados o conjuntos de datos más complejos. Además, Repository a menudo se asocia con conceptos de persistencia y recuperación de entidades de negocio completas, mientras que DAO está más centrado en operaciones directas en entidades individuales.

Ejemplo de implementación en C#:

Considere que la aplicación del ejemplo anterior permite realizar operaciones CRUD sobre la entidad Producto correspondiente a una tabla con el mismo nombre en la base de datos. Para definir un mecanismo de abstracción que permita recuperar y grabar dichas entidades se necesita:

1. **Definición de la interfaz** `IProductoRepository`**:**

```
public interface IProductoRepository
{
   Producto? GetById(int id); //Símbolo ? indica que el método puede
                              //retornar un valor nulo.
   IEnumerable<Producto> GetAll();
   void Save(Producto entity);
   void Delete(Producto entity);
}
```

2. **Implementación de la interfaz** `IProductoRepository`

```
public class ProductoRepository : IProductoRepository
{
    private string _connString;
    public ProductoRepository()
    {
        _connString = Properties.Resources.CnnString;
    }
```

```
public IEnumerable<Producto> GetAll()
{
    List<Producto> lst = new List<Producto>();
    using (var cnn = new SqlConnection(_connString))
    {
        cnn.Open();
        var query = $"SELECT * FROM Productos";
        SqlCommand cmd = new SqlCommand(query, cnn);
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            Producto aux = Mapper(reader);
            lst.Add(aux);
        }

    }
    return lst;
}

public Producto? GetById(int id)
{
    Producto? aux = null;
    using (var cnn = new SqlConnection(_connString))
    {
        cnn.Open();
        var query = $"SELECT * FROM Productos WHERE Id = @Id";
        SqlCommand cmd = new SqlCommand(query, cnn);
        cmd.Parameters.AddWithValue("@Id", id);
        SqlDataReader reader = cmd.ExecuteReader();
        if (reader.Read())
            aux = Mapper(reader);
    }
    return aux;
}

public void Save(Producto entity)
{
    if (entity != null)
    {
        string query = string.Empty;
        if (entity.Id == 0)
        {
            //Entidad nueva
```

```
                query = "INSERT INTO Productos (Nombre, Stock, Precio)
                VALUES(@Nombre, @Stock, @Precio)";
            }
            else
            {
                query = "UPDATE Productos SET Nombre = @Nombre, Stock =
                @Stock, Precio = @Precio WHERE Id = @Id";
            }

            using (var cnn = new SqlConnection(_connString))
            {
                cnn.Open();
                SqlCommand cmd = new SqlCommand(query, cnn);
                cmd.Parameters.AddWithValue("@Id", entity.Id);
                cmd.Parameters.AddWithValue("@Nombre", entity.Nombre);
                cmd.Parameters.AddWithValue("@Stock", entity.Stock);
                cmd.Parameters.AddWithValue("@Precio", entity.Precio);
                cmd.ExecuteNonQuery();
            }
        }
    }

    public void Delete(Producto entity)
    {
        if (entity != null && entity.Id != 0)
        {
            using (var cnn = new SqlConnection(_connString))
            {
                cnn.Open();
                string query = "DELETE Productos WHERE Id = @Id";
                SqlCommand cmd = new SqlCommand(query, cnn);
                cmd.Parameters.AddWithValue("@Id", entity.Id);
                cmd.ExecuteNonQuery();
            }
        }
    }

    private Producto Mapper(SqlDataReader reader)
    {
        Producto aux = new Producto();
        aux.Id = reader.GetInt32(0);
        aux.Nombre = reader.GetString(1);
        aux.Stock = reader.GetInt32(2);
        aux.Precio = reader.GetInt32(3);
        return aux;
```

```
        }
    }
```

Algunas consideraciones importantes:

- Notar que, a diferencia de un mecanismo estándar de persistencia, la interfaz define un método `Save()` en vez de Create/Update. El método `Save()` en un Repositorio puede manejar tanto la inserción como la actualización de entidades, dependiendo del contexto y del estado actual del objeto que se pasa como parámetro. Esto puede ser una elección válida y coherente en el contexto del patrón Repository, especialmente cuando se desea simplificar la interfaz y mejorar la semántica de las operaciones de persistencia de datos.
- Uso de **bloque using(){}**: la estructura using en C# se utiliza para garantizar que los recursos que implementan la interfaz **IDisposable** se liberen correctamente al finalizar su uso, incluso si ocurren excepciones durante la ejecución del código. En el contexto del ejemplo:

  ```
  using (var cnn = new SqlConnection(_connString))
  {//…}
  ```

  La palabra clave **using** se utiliza para crear un bloque dentro del cual se instancia y utiliza el objeto **SqlConnection**. Aquí están los aspectos clave de cómo y por qué se utiliza using en este caso:

  1. Manejo automático de recursos: SqlConnection es una clase que maneja recursos externos como conexiones a bases de datos. Estos recursos deben ser liberados explícitamente cuando ya no son necesarios para evitar posibles problemas de memoria o de recursos.
  2. Implementación de IDisposable: SqlConnection implementa la interfaz IDisposable, lo que significa que tiene un método Dispose() que libera los recursos no administrados asociados a la conexión. Al usar using, el compilador genera automáticamente un bloque try-finally que asegura que Dispose() se llame al finalizar el bloque using, incluso si ocurre una excepción durante la ejecución del código dentro del bloque.
  3. Garantía de liberación de recursos: En el ejemplo, SqlConnection se abre dentro del bloque using. Después de completar las operaciones dentro del bloque (en este caso, ejecutar una consulta SQL para eliminar un producto), el flujo de ejecución automáticamente llama a Dispose() en

SqlConnection. Esto cierra la conexión a la base de datos y libera los recursos asociados.

- Auxiliarmente la clase ProductoRepository utiliza el método `Mapper(reader)` que permite corresponder una fila de la tabla de Productos con una entidad homónima del dominio del problema.
- En la implementación de la interfaz es posible utilizar los objetos de ADO.NET para ejecutar directamente **procedimientos almacenados**, eliminando así el código SQL de la aplicación, lo que redunda en un desacoplamiento aún mayor con los mecanismos de persistencia. La creación y uso de procedimientos queda como anexo de la presente unidad.
- En una aplicación divida en capas, el Repository proporciona una abstracción entre la lógica de negocio de la aplicación y los detalles específicos de cómo se acceden y persisten los objetos de dominio. Esto promueve un diseño más limpio y desacoplado, donde la lógica de negocio no está directamente vinculada a la infraestructura de almacenamiento o acceso a datos.
- El patrón se puede combinar con una implementación de un ORM (Modelo Objeto-Relacional), como es Entity Framework Core (que se verá más adelante), para ofrecer una abstracción más clara y eficiente del mecanismo de persistencia.
- Comúnmente el patrón Repository suele combinarse con otro patrón muy difundido en la gestión de persistencia de datos, el **Unit of Work** (Unidad de Trabajo). Este último resuelve el problema de administrar **transacciones** que involucren múltiples operaciones de actualización agrupando dichas operaciones y asegurando que todas se completen con éxito o se deshagan en caso de error.

MANEJO DE TRANSACCIONES

Las transacciones se usan cuando se desea realizar una secuencia de operaciones (o sentencias SQL), de manera que son ejecutadas como una única **unidad de trabajo**. Por ejemplo, queremos una aplicación que realice dos acciones en nuestra base de datos, la primera acción inserta un registro en nuestra tabla MAESTRO y la otra acción inserta un segundo registro en la tabla de DETALLE.

Con el manejo de transacciones de ADO.NET, no importa que cualquiera de las dos acciones falle, ya que los cambios no se realizarán en la base de datos a menos que todas las operaciones sean **confirmadas de manera exitosa.**

Las propiedades de las transacciones se las conoce como **ACID**: **A**tomicidad, **C**oherencia, A**i**slamiento y **D**urabilidad.

- *Atomicidad*: Una transacción debe ser una unidad atómica de trabajo, o se hace todo o no se hace nada.
- *Coherencia*: Debe dejar los datos en un estado coherente luego de realizada la transacción.
- *Aislamiento*: Las modificaciones realizadas por transacciones son tratadas en forma independiente, como si fueran un solo y único usuario de la base de datos.
- *Durabilidad*: Una vez concluida la transacción sus efectos son permanentes y no hay forma de deshacerlos.

ADO.NET soporta el concepto de transacción mediante el uso transacciones iniciadas por código. Para llevar a cabo una transacción de este tipo son necesarios los siguientes pasos:

1. Llamar al método **BeginTransaction** del objeto **SqlConnection** para marcar el comienzo de la transacción. El método BeginTransaction devuelve una referencia a la transacción. Esta referencia se asigna a los objetos **SqlCommand** que están inscritos en la transacción.
2. Asigne el objeto Transaction a la propiedad Transaction del objeto SqlCommand que se va a ejecutar. Si el comando se ejecuta en una conexión con una transacción activa y el objeto Transaction no se ha asignado a la propiedad Transaction del objeto Command, se inicia una excepción.
3. Ejecute los comandos necesarios.
4. Llame al método Commit del objeto SqlTransaction para completar la transacción, o al método Rollback para finalizarla. Si la conexión se cierra o elimina antes de que se hayan ejecutado los métodos Commit o Rollback, la transacción se revierte.
5. El siguiente ejemplo muestra cómo manejar una transacción con los objetos provistos por ADO.NET. Continuando con caso de los presupuestos resulta:

```
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;
              cmd.Parameters.AddWithValue("@cliente", this.Cliente);
              cmd.Parameters.AddWithValue("@dto", this.Descuento);
              cmd.Parameters.AddWithValue("@total", this.calcularTotal()
              - this.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 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 ex)
          {
              t.Rollback();
              resultado = false;
```

```
            }
            finally
            {
                if (cnn != null && cnn.State == ConnectionState.Open)
                    cnn.Close();
            }
            return resultado;
        }
```

Implementando Unit of Work

El patrón Unit of Work se utiliza para administrar transacciones a través de múltiples operaciones de Repository. Agrupa operaciones relacionadas en una única transacción, asegurando que todas se completen con éxito o se reviertan si alguna falla.

Ejemplo de Implementación en C# con ADO.NET

Retomando la implementación del repositorio de productos del ejemplo anterior la implementación del patrón **Unit of Work** resulta:

```
public class UnitOfWork: IDisposable
{
    private readonly SqlConnection _connection;
    private SqlTransaction _transaction;
    private IProductoRepository _productoRepository;

    public UnitOfWork(string cnnString)
    {
        _connection = new SqlConnection(cnnString);
        _connection.Open();
        _transaction = _connection.BeginTransaction();
    }

    public IProductoRepository ProductoRepository
    {
        get
        {
            if (_productoRepository == null)
            {
                _productoRepository= new productoRepository(_connection);
            }
            return _productoRepository;
```

```
            }
        }

        public void SaveChanges()
        {
            try
            {
                _transaction.Commit();
            }
            catch (Exception ex)
            {
                _transaction.Rollback();
                throw new Exception("Error al guardar cambios en la base de
                datos.", ex);
            }
        }

        public void Dispose()
        {
            if (_transaction != null)
            {
                _transaction.Dispose();
            }
            if (_connection != null)
            {
                _connection.Close();
                _connection.Dispose();
            }
        }
    }
```

Uso en la capa de negocio

```
    public class GestorProducto
    {
        private readonly UnitOfWork _unitOfWork;

        public GestorProducto (UnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public void NuevoProducto(Producto producto)
        {
            _unitOfWork.ProductoRepository.Save(producto);
```

```
    }

    public void ModificarProducto(Producto producto)
    {
        _unitOfWork.ProductoRepository.Save(producto);
    }

    public void BorrarProducto(Producto producto)
    {
        _unitOfWork.ProductoRepository.Delete(producto);
        //No se llama a SaveChanges aquí
    }

    public IEnumerable<Producto> ObtenerTodos()
    {
        return _unitOfWork.ProductoRepository.GetAll();
    }

    public void GuardarCambios()
    {
        _unitOfWork.SaveChanges();//Se llama a SaveChanges al final,
                                  //después de todas las operaciones.
    }
}
```

Consideraciones finales

1. El objetivo principal del Unit of Work es agrupar varias operaciones relacionadas en una única transacción y confirmar o revertir dicha transacción de manera coherente. Cuando utilizas el Unit of Work junto con el patrón Repository, normalmente sigues estos pasos:

    - **Iniciar una Transacción:** Cuando se crea una instancia del UnitOfWork, se inicia una transacción en la conexión de base de datos.
    - **Realizar Operaciones CRUD:** Utilizas los métodos del Repository a través del Unit of Work para realizar operaciones de inserción (Insert), actualización (Update), eliminación (Delete), etc.
    - **Confirmar Cambios:** Una vez que todas las operaciones necesarias se han completado satisfactoriamente, llamas al método SaveChanges del Unit of Work para confirmar la transacción. Este método generalmente

realiza un Commit en la transacción activa, lo que persiste los cambios en la base de datos.

- **Manejo de Errores:** Si ocurre algún error durante la ejecución de las operaciones dentro del Unit of Work, el SaveChanges puede realizar un Rollback para deshacer todas las operaciones y mantener la integridad de los datos.

2. No es Necesario Llamar SaveChanges en Cada Operación. Dado que el patrón ya maneja la transacción y la confirmación de cambios, no es necesario llamar a SaveChanges después de cada operación individual (inserción, actualización, etc.). El SaveChanges se llama una vez al final, después de que todas las operaciones relacionadas hayan sido ejecutadas satisfactoriamente.

Implementando 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:

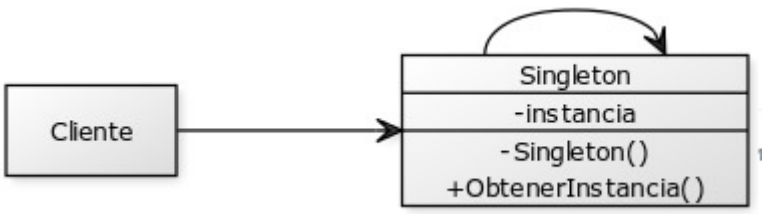


Imagen 10: Elaboración propia

Ejemplificando para el dominio analizado en el caso de estudio, se define la clase **HelperRepository** con las siguientes líneas de código:

```
class HelperRepository
{
        private static HelperRepository _instancia;
        private string connString;
        private HelperRepository () {
                _connString = Properties.Resources.CnnString;
        }
        public static HelperRepository ObtenerInstancia()
        {
                if (instancia == null) {
                        instancia = new HelperRepository();
                }
                return instancia;
        }
        public DataTable ConsultarSQL(string query) {
           DataTable tabla = new DataTable();
           try{
                  using (var cnn = new SqlConnection(_connString))
                  {
                         cnn.Open();
                         SqlCommand cmd = new SqlCommand(query, cnn);
                         tabla.Load(cmd.ExecuteReader());

                  }
                  return tabla;
           }catch(SqlException ex){
                  return null;
           }

        }
}
```

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

```
public IEnumerable<Producto> GetAll()
    {
        List<Producto> lst = new List<Producto>();
        HelperRepository helper = HelperRepository.GetInstance();
        DataTable tabla = helper.ConsultarSQL("SELECT * FROM Producto");
        // mapear las filas del DataTable y cargar el objeto lst.
        //…
        return lst;
    }
```

Consideraciones finales

- De esta forma se garantiza que existe un único objeto HelperRepository con el que se accede a la base de datos para ejecutar sentencias SQL. Solo resta completar la clase HelperRepository con los métodos para ejecutar sentencias DML y procedimientos con parámetros de entrada/salida. Es altamente recomendable usarlo como actividad de auto-aprendizaje por parte de los estudiantes.
- Permite generalizar y reutilizar todo el código necesario para crear y comunicar los objetos de ADO.NET cuando se ejecutan sentencias SQL.
- Cabe mencionar que esta implementación es válida para ver la fisionomía del patrón, aunque es posible encontrar alternativas más eficientes que contemplen toda la problemática subyacente a la ejecución de sentencias SQL contra una base de datos.

ANEXO

Procedimientos almacenados con ADO.NET

Un procedimiento almacenado (Store Procedure o SP en inglés) es un conjunto de sentencias SQL que aceptan y/o retornan cero o más variables, comúnmente llamados parámetros. Para nuestro propósito, los SP nos permitirán escribir todas las sentencias SQL necesarias para registrar, actualizar, eliminar y/o consultar las entidades gestionadas por la aplicación, eliminando todo código SQL de nuestro proyecto. Esta forma de incluir la lógica de la aplicación en la base de datos utilizando procedimientos almacenados tiene como principal ventaja la simplificación del mantenimiento de nuestros programas.

Para consumir un SP desde C# utilizando los objetos de ADO necesitamos crear un objeto **Command** (en nuestro caso SQLCommand) y configurarlo para ejecutar correctamente el procedimiento. Para ello es necesario:

1. Obtener una conexión mediante un objeto **SQLConnection (SqlConnection cnn)**
2. Crear un objeto **SQLCommand** y asociarlo con el objeto conexión (**SQLCommand cmd**)
3. Asignar las propiedades Text y Type al comando:
    a. **cmd**.Connection = cnn;
    b. **cmd**.CommandText = “NOMBRE_DEL_PROCEDIMIENTO”;
    c. **cmd**.CommandType = CommandType.StoredProcedure;
    d. Establecer el/los parámetros del procedimiento (opcional):

    cmd.Parameters.AddWithValue("@param1",valor1); siendo valor1 una variable de nuestro programa y que necesitamos asignar al parámetro @param1 definido en el cuerpo del SP. La propiedad Parameters es una colección que almacena los objetos SQLParameter que serán enviados al ejecutarlo. El método AddWithValue(key, value) permite establecer para un parámetro su valor sin necesidad de definir un tipo específico.

Será nuestra responsabilidad enviar los valores correctos según la definición del procedimiento en la base de datos.

Otra forma es crear un objeto SQLParameter, setearle sus propiedades y luego agregarlo a la colección **Parameters.** Tal como lo muestra el siguiente método:

```
private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter();
    parameter.ParameterName = "@Description";
    parameter.IsNullable = true;
    parameter.SqlDbType = SqlDbType.VarChar;
    command.Parameters.Add(parameter);
}
```

e. Executar el comando:

cmd.ExecuteNonQuery();

Parámetros de salida

Otro aspecto a considerar son los parámetros de retorno (o salida) que pudiera devolver el procedimiento. En la mayoría de los casos o no devuelve ningún valor o devuelve el resultado de una consulta. Por ejemplo, dado el siguiente Store Procedure escrito en SQLServer:

```
CREATE PROCEDURE [dbo].[SP_CONSULTAR_MODELOS]
      @nombre varchar(40)
AS
BEGIN
      SELECT * FROM T_MODELOS t WHERE t.nombre = @nombre;
END
```

Para ejecutarlo se desarrolla el siguiente método:

```
public void EjecutarSP(string nombreSP, Connection cnn)
    {
        try
        {
```

```
                    SqlCommand cmd = new SqlCommand(nombreSP, cnn);
                     cmd.CommandType = CommandType.StoredProcedure;
                     cmd.Parameters.AddWithValue("@nombre","marca 1");
                     DataTable table = new DataTable();
                     table.Load(cmd.ExecuteReader());
               //procesar el resultado obtenido
                }
                    catch (Exception ex)
              {
               throw new Exception(" Error al ejecutar procedimiento almacenado
", ex);
              }
          }
```

Suponiendo que tenemos una conexión creada y abierta a la base de datos, entonces se puede llamar al método anterior:

```
EjecutarSP ("SP_CONSULTAR_MODELOS",cnn);
```

Como vemos permite obtener como resultado la consulta de todos los modelos de la tabla T_MODELOS filtrada por nombre, siendo este último el único parámetro de entrada del procedimiento.

Veamos ahora un segundo ejemplo en el que insertamos una fila en la tabla T_MODELOS y obtenemos un parámetro de salida que representa el último ID generado para su clave primaria de tipo IDENTITY (Autoincremental):

```
CREATE PROCEDURE SP_INSERTAR_MODELO
    @nombre varchar(40),
    @NewId int OUTPUT
AS
BEGIN
    INSERT INTO T_MODELOS (nombre)
    VALUES (@nombre);
    --Asignar el valor del último ID autogenerado (obtenido --
    --mediante la función SCOPE_IDENTITY() de SQLServer)
    SET @NewId = SCOPE_IDENTITY();
END
```

Si se refactoriza el método anterior resulta:

```
public void EjecutarSP(string nombreSP, Connection cnn)
    {
        try
        {
                SqlCommand cmd = new SqlCommand(nombreSP, cnn);
                cmd.CommandType = CommandType.StoredProcedure;
                //parámetro de entrada:
                cmd.Parameters.AddWithValue("@nombre","marca 1");
                //parámetro de salida:
                SqlParameter param = new SqlParameter("@NewId", SqlDbType.Int);
                param.Direction = ParameterDirection.Output;
                cmd.Parameters.Add(param);
                //ejecutamos el comando
                cmd.ExecuteNonQuery();
                //Obtenemos el valor del parámetro de salida:
                int ultimoID = Convert.ToInt32(param.Value);
        }
        catch (Exception ex)
        {
         throw new Exception(" Error al ejecutar procedimiento almacenado ", ex);
        }
    }
//llamamos al método EjecutarSP:
EjecutarSP ("SP_INSERTAR_MODELO",cnn);
```

En este caso el procedimiento tiene 2 parámetros: uno de entrada y otro de salida con el que se obtiene el valor del último ID generado para la columna ID_MODELO de la tabla T_MODELOS luego de ejecutar la sentencia **Insert**.

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/

Documentación de GitHub. Get started. Recuperado de: https://docs.github.com/es/get-started

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