TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE:
Agosto - Diciembre 2025

CARRERA:
Ingeniería en Sistemas Computacionales

MATERIA:
Patrones de Diseño de Software

TÍTULO ACTIVIDAD:
Examen unidad 2

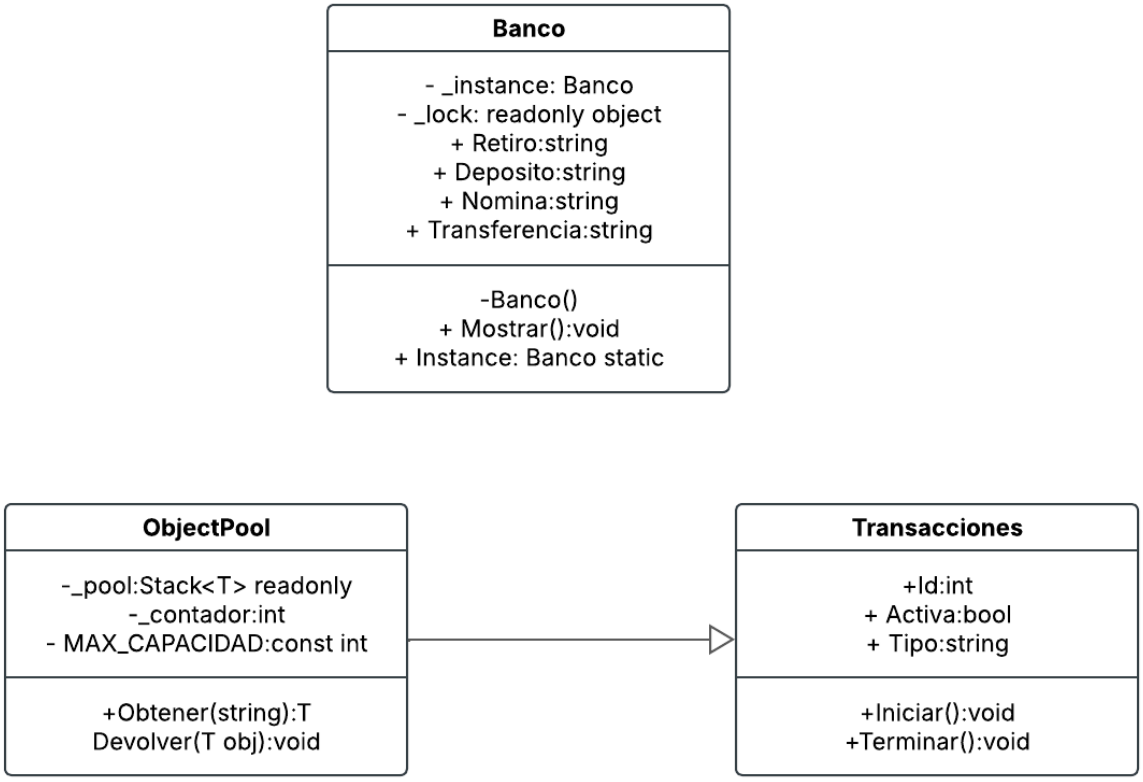
Alumno:
Hernandez Vazquez Cristian 21211964

NOMBRE DEL MAESTRO(A):
Maribel Guerrero Luis

Fecha

16/10/2025

UML


Banco
- _instance: Banco
- _lock: readonly object
+ Retiro:string
+ Deposito:string
+ Nomina:string
+ Transferencia:string
-Banco()
+ Mostrar():void
+ Instance: Banco static
ObjectPool
-_pool:Stack<T> readonly
-_contador:int
- MAX_CAPACIDAD:const int
+Obtener(string):T
Devolver(T obj):void
Transacciones
+Id:int
+ Activa:bool
+ Tipo:string
+Iniciar():void
+Terminar():void

Código

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ExamenPatrones1
{
    internal class Program
    {
        public class Banco
        {
            private static Banco _instance;
            private static readonly object _lock = new object();
            //Tipos de tarnsacciones que ofrece el banco
            public string Retiro { get; private set; } // Tipo de transacción: Retiro
            public string Deposito { get; private set; } // Tipo de transacción: Deposito
            public string Nomina { get; private set; } // Tipo de transacción: Nomina
            public string Transferencia { get; private set; } // Tipo de transacción:
Transferencia

            private Banco()
            {
                Retiro = "Retiro";
                Deposito = "Deposito";
                Nomina = "Nomina";
                Transferencia = "Transferencia";
            }
```

```
        public void Mostrar()
        {
            Console.WriteLine($"Tipos de
transferencias:\n1.-{Retiro}\n2.-{Deposito}\n3.-{Nomina}\n4.-{Transferencia}");


        }


        public static Banco Instance
        {
            get
            {
                if (_instance == null)
                {
                    lock (_lock)
                    {
                        if (_instance == null)
                            _instance = new Banco();
                    }
                }
                return _instance;
            }
        }
    }
    public class Transacciones
    {
        public int Id { get; set; }
        public bool Activa { get; set; }
        public string Tipo { get; set; }
```

```
    public void Iniciar()
    {
        if (Activa)
        {
            Console.WriteLine("La transacción no ha terminado o está en uso");
        }
        else
        {
            Activa = true;
            Console.WriteLine($"Transacción con: {Id}. ¡Activa!");
        }
    }
    public void Terminar()
    {
        Activa = false;
        Console.WriteLine($"La transacción con el Id: {Id}. Ha terminado");
    }
}

// Implementación genérica del Object Pool
public class ObjectPool<T> where T : Transacciones, new()
{
    private readonly Stack<T> _pool = new Stack<T>();
    private int _contador = 0;
    private const int MAX_CAPACIDAD = 10;

    public T Obtener(string tipo)
    {
```

```
            if (_pool.Count > 0)
            {
                var obj = _pool.Pop();
                Console.WriteLine($"Reutilizando transacción con Id: {obj.Id}");
                obj.Activa = true;
                obj.Tipo = tipo;
                return obj;
            }

            // Si no hay objetos disponibles en el pool
            if (_contador < MAX_CAPACIDAD)
            {
                var nuevo = new T { Id = ++_contador, Activa = true, Tipo = tipo };
                Console.WriteLine($"Creando nueva transacción con Id: {nuevo.Id}");
                return nuevo;
            }

            // Límite alcanzado y ningún objeto libre
            Console.WriteLine("No hay recursos disponibles. Por favor, termine una
transacción antes de iniciar otra.");
            return null;
        }

        public void Devolver(T obj)
        {
            if (obj.Activa)
            {
                Console.WriteLine("La transacción sigue activa. No puede devolverse.");
            }
```

```
            else
            {
                Console.WriteLine($"Transacción con Id: {obj.Id} devuelta al pool.");
                _pool.Push(obj);
            }
        }

        public int Disponible => _pool.Count;
    }




    static void Main(string[] args)
    {
        // Singleton: Banco
        Banco banco1 = Banco.Instance;
        Banco banco2 = Banco.Instance;
        Console.WriteLine("Revisando que los bancos sean iguales: " +
ReferenceEquals(banco1, banco2));

        // Object Pool con capacidad finita
        var pool = new ObjectPool<Transacciones>();

        List<Transacciones> usuarioTransacciones = new List<Transacciones>();
        char continuarProceso;
        do
        {
            char continuar = ' ';
            int opcion = 0;
```

```
//=== Parte 1: Crear nuevas transacciones usando Object Pool ===
do
{
    banco1.Mostrar();
    Console.Write("¿Cuál opción va a escoger? ");
    if (!int.TryParse(Console.ReadLine(), out opcion))
    {
        Console.WriteLine("Opción inválida. Intente de nuevo.");
        continue;
    }
    string tipo;
    switch (opcion)
    {
        case 1: tipo = "Retiro"; break;
        case 2: tipo = "Deposito"; break;
        case 3: tipo = "Nomina"; break;
        case 4: tipo = "Transferencia"; break;
        default: tipo = null; break;
    }

    if (tipo == null)
    {
        Console.WriteLine("No se maneja este tipo de transacción.");
    }
    else
    {
        var transaccion = pool.Obtener(tipo);
```

```
        if (transaccion != null)
        {
            usuarioTransacciones.Add(transaccion);
        }
        else
        {
            //Si el Pool está vacío pasar a la sección de finalizar transacciones
            break;
        }
    }

    Console.WriteLine($"\nPool disponible: {pool.Disponible}");
    Console.WriteLine("¿Va a realizar más transacciones? s/n");
    continuar = char.Parse(Console.ReadLine().ToLower());
    Console.WriteLine();

} while (continuar != 'n');

//=== Parte 2: Finalizar transacciones activas ===
char continuarFinalizar = ' ';
do
{
    var activas = usuarioTransacciones.Where(t => t.Activa).ToList();

    if (activas.Count == 0)
    {
        Console.WriteLine("No hay transacciones por finalizar.");
        break;
```

```
            }

            Console.WriteLine("Transacciones activas:");
            foreach (var t in activas)
            {
                Console.WriteLine($"Id: {t.Id} - Tipo: {t.Tipo}");
            }

            Console.Write("Ingrese Id de la transacción a finalizar: ");

            if (int.TryParse(Console.ReadLine(), out int idFinalizar)) //Obliga a que sea
un número
            {
                var trans = usuarioTransacciones.FirstOrDefault(t => t.Id == idFinalizar
&& t.Activa);
                if (trans != null)
                {
                    trans.Activa = false;
                    Console.WriteLine($"Transacción Id {trans.Id} finalizada.");
                }
                else
                {
                    Console.WriteLine("Id inválido o transacción ya finalizada.");
                }
            }

            Console.WriteLine("¿Desea finalizar otra transacción? s/n");
            continuarFinalizar = char.Parse(Console.ReadLine().ToLower());
            Console.WriteLine();
```

```
            } while (continuarFinalizar != 'n');

            //=== Parte 3: Devolver transacciones terminadas al pool ===
            Console.WriteLine("Devolviendo transacciones finalizadas al pool...");
            for (int i = usuarioTransacciones.Count - 1; i >= 0; i--)
            {
                if (!usuarioTransacciones[i].Activa)
                {
                    pool.Devolver(usuarioTransacciones[i]);
                    usuarioTransacciones.RemoveAt(i);
                }
            }

            Console.WriteLine($"\nPool disponible: {pool.Disponible}");
            Console.WriteLine("¿Desea repetir el proceso? s/n");
            continuarProceso = char.Parse(Console.ReadLine().ToLower());
            Console.WriteLine();

        } while (continuarProceso != 'n');

        Console.WriteLine("Programa finalizado.");
    }
  }
}
```

Capturas de pantalla

```
8 referencias
public class Banco
{
    private static Banco _instance;
    private static readonly object _lock = new object();
    //Tipos de tarnsacciones que ofrece el banco
    2 referencias
    public string Retiro { get; private set; } // Tipo de transacción: Retiro
    2 referencias
    public string Deposito { get; private set; } // Tipo de transacción: Deposito
    2 referencias
    public string Nomina { get; private set; } // Tipo de transacción: Nomina
    2 referencias
    public string Transferencia { get; private set; } // Tipo de transacción: Transferencia

    1 referencia
    private Banco()
    {
        Retiro = "Retiro";
        Deposito = "Deposito";
        Nomina = "Nomina";
        Transferencia = "Transferencia";
    }
    1 referencia
    public void Mostrar()
    {
        Console.WriteLine($"Tipos de transferencias:\n1.-{Retiro}\n2.-{Deposito}\n3.-{Nomina}\n4.-{Transferencia}");

    }
```

```
2 referencias
public static Banco Instance
{
    get
    {
        if (_instance == null)
        {
            lock (_lock)
            {
                if (_instance == null)
                    _instance = new Banco();
            }
        }
        return _instance;
    }
}
```

```
public class Transacciones
{
    9 referencias
    public int Id { get; set; }
    10 referencias
    public bool Activa { get; set; }
    3 referencias
    public string Tipo { get; set; }

    0 referencias
    public void Iniciar()
    {
        if (Activa)
        {
            Console.WriteLine("La transacción no ha terminado o está en uso");
        }
        else
        {
            Activa = true;
            Console.WriteLine($"Transacción con: {Id}. ¡Activa!");
        }
    }
    0 referencias
    public void Terminar()
    {
        Activa = false;
        Console.WriteLine($"La transacción con el Id: {Id}. Ha terminado");
    }
}
```

```
// Implementación genérica del Object Pool
1 referencia
public class ObjectPool<T> where T : Transacciones, new()
{
    private readonly Stack<T> _pool = new Stack<T>();
    private int _contador = 0;
    private const int MAX_CAPACIDAD = 10;

    1 referencia
    public T Obtener(string tipo)
    {
        if (_pool.Count > 0)
        {
            var obj = _pool.Pop();
            Console.WriteLine($"Reutilizando transacción con Id: {obj.Id}");
            obj.Activa = true;
            obj.Tipo = tipo;
            return obj;
        }

        // Si no hay objetos disponibles en el pool
        if (_contador < MAX_CAPACIDAD)
        {
            var nuevo = new T { Id = ++_contador, Activa = true, Tipo = tipo };
            Console.WriteLine($"Creando nueva transacción con Id: {nuevo.Id}");
            return nuevo;
        }

        // Límite alcanzado y ningún objeto libre
        Console.WriteLine("No hay recursos disponibles. Por favor, termine una transa
        return null;
    }

    1 referencia
```

```
    1 referencia
    public void Devolver(T obj)
    {
        if (obj.Activa)
        {
            Console.WriteLine("La transacción sigue activa. No puede devolverse.");
        }
        else
        {
            Console.WriteLine($"Transacción con Id: {obj.Id} devuelta al pool.");
            _pool.Push(obj);
        }
    }

    2 referencias
    public int Disponible => _pool.Count;
}
```

```
0 referencias
static void Main(string[] args)
{
    // Singleton: Banco
    Banco banco1 = Banco.Instance;
    Banco banco2 = Banco.Instance;
    Console.WriteLine("Revisando que los bancos sean iguales: " + ReferenceEquals(banco

    // Object Pool con capacidad finita
    var pool = new ObjectPool<Transacciones>();

    List<Transacciones> usuarioTransacciones = new List<Transacciones>();
    char continuarProceso;
    do
    {
        char continuar = ' ';
        int opcion = 0;

        //=== Parte 1: Crear nuevas transacciones usando Object Pool ===
        do
        {
            banco1.Mostrar();
            Console.Write("¿Cuál opción va a escoger? ");
            if (!int.TryParse(Console.ReadLine(), out opcion))
            {
                Console.WriteLine("Opción inválida. Intente de nuevo.");
                continue;
            }
            string tipo;
            switch (opcion)
            {
                case 1: tipo = "Retiro"; break;
                case 2: tipo = "Deposito"; break;
                case 3: tipo = "Nomina"; break;
                case 4: tipo = "Transferencia"; break;
                default: tipo = null; break;
            }

            if (tipo == null)
```

```
            if (tipo == null)
            {
                Console.WriteLine("No se maneja este tipo de transacción.");
            }
            else
            {
                var transaccion = pool.Obtener(tipo);
                if (transaccion != null)
                {
                    usuarioTransacciones.Add(transaccion);
                }
                else
                {
                    //Si el Pool está vacío pasar a la sección de finalizar transacciones
                    break;
                }
            }

            Console.WriteLine($"\nPool disponible: {pool.Disponible}");
            Console.WriteLine("¿Va a realizar más transacciones? s/n");
            continuar = char.Parse(Console.ReadLine().ToLower());
            Console.WriteLine();

        } while (continuar != 'n');

        //=== Parte 2: Finalizar transacciones activas ===
        char continuarFinalizar = ' ';
        do
        {
            var activas = usuarioTransacciones.Where(t => t.Activa).ToList();

            if (activas.Count == 0)
            {
                Console.WriteLine("No hay transacciones por finalizar.");
                break;
            }
```

```
        Console.WriteLine("Transacciones activas:");
        foreach (var t in activas)
        {
            Console.WriteLine($"Id: {t.Id} - Tipo: {t.Tipo}");
        }

        Console.Write("Ingrese Id de la transacción a finalizar: ");

        if (int.TryParse(Console.ReadLine(), out int idFinalizar)) //Obliga a que sea un
        {
            var trans = usuarioTransacciones.FirstOrDefault(t => t.Id == idFinalizar && t
            if (trans != null)
            {
                trans.Activa = false;
                Console.WriteLine($"Transacción Id {trans.Id} finalizada.");
            }
            else
            {
                Console.WriteLine("Id inválido o transacción ya finalizada.");
            }
        }

        Console.WriteLine("¿Desea finalizar otra transacción? s/n");
        continuarFinalizar = char.Parse(Console.ReadLine().ToLower());
        Console.WriteLine();

    } while (continuarFinalizar != 'n');
```

```
            //=== Parte 3: Devolver transacciones terminadas al pool ===
            Console.WriteLine("Devolviendo transacciones finalizadas al pool...");
            for (int i = usuarioTransacciones.Count - 1; i >= 0; i--)
            {
                if (!usuarioTransacciones[i].Activa)
                {
                    pool.Devolver(usuarioTransacciones[i]);
                    usuarioTransacciones.RemoveAt(i);
                }
            }

            Console.WriteLine($"\nPool disponible: {pool.Disponible}");
            Console.WriteLine("¿Desea repetir el proceso? s/n");
            continuarProceso = char.Parse(Console.ReadLine().ToLower());
            Console.WriteLine();

        } while (continuarProceso != 'n');

        Console.WriteLine("Programa finalizado.");
    }
}
}
```

Código ejecutándose

```
Revisando que los bancos sean iguales: True
Tipos de transferencias:
1.-Retiro
2.-Deposito
3.-Nomina
4.-Transferencia
¿Cuál opción va a escoger? 1
Creando nueva transacción con Id: 1

Pool disponible: 0
¿Va a realizar más transacciones? s/n
s

Tipos de transferencias:
1.-Retiro
2.-Deposito
3.-Nomina
4.-Transferencia
¿Cuál opción va a escoger? 1
Creando nueva transacción con Id: 2

Pool disponible: 0
¿Va a realizar más transacciones? s/n
s
```

```
1.-Retiro
2.-Deposito
3.-Nomina
4.-Transferencia
¿Cuál opción va a escoger? 1
Creando nueva transacción con Id: 10

Pool disponible: 0
¿Va a realizar más transacciones? s/n
s

Tipos de transferencias:
1.-Retiro
2.-Deposito
3.-Nomina
4.-Transferencia
¿Cuál opción va a escoger? 1
No hay recursos disponibles. Por favor, termine una transacción antes de iniciar otra.
Transacciones activas:
Id: 1 - Tipo: Retiro
Id: 2 - Tipo: Retiro
Id: 3 - Tipo: Retiro
Id: 4 - Tipo: Retiro
Id: 5 - Tipo: Retiro
Id: 6 - Tipo: Retiro
Id: 7 - Tipo: Retiro
Id: 8 - Tipo: Retiro
Id: 9 - Tipo: Retiro
Id: 10 - Tipo: Retiro
Ingrese Id de la transacción a finalizar:
```

```
Transacciones activas:
Id: 10 - Tipo: Retiro
Ingrese Id de la transacción a finalizar: 10
Transacción Id 10 finalizada.
¿Desea finalizar otra transacción? s/n
s

No hay transacciones por finalizar.
Devolviendo transacciones finalizadas al pool...
Transacción con Id: 10 devuelta al pool.
Transacción con Id: 9 devuelta al pool.
Transacción con Id: 8 devuelta al pool.
Transacción con Id: 7 devuelta al pool.
Transacción con Id: 6 devuelta al pool.
Transacción con Id: 5 devuelta al pool.
Transacción con Id: 4 devuelta al pool.
Transacción con Id: 3 devuelta al pool.
Transacción con Id: 2 devuelta al pool.
Transacción con Id: 1 devuelta al pool.

Pool disponible: 10
¿Desea repetir el proceso? s/n
```

Conclusión

A través de este programa se puede observar la implementación de los patrones Singleton y Object Pool, y cómo cada uno aporta ventajas claras en la creación de código. Por ejemplo, el Object Pool permite reutilizar objetos en lugar de crear constantemente nuevos, lo que reduce el uso de recursos, mejora la eficiencia, la escalabilidad, la mantenibilidad y la coherencia del programa. Por su parte, el Singleton asegura que exista una única instancia de una clase, manteniendo el código más ordenado y evitando duplicaciones innecesarias. Además, el uso de readonly en algunas variables refuerza la seguridad y la inmutabilidad de datos críticos, evitando modificaciones accidentales. En conjunto, estos patrones permiten desarrollar programas con un diseño más sólido, eficiente y fácilmente escalable.