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

TÍTULO ACTIVIDAD :
Examen unidad 2

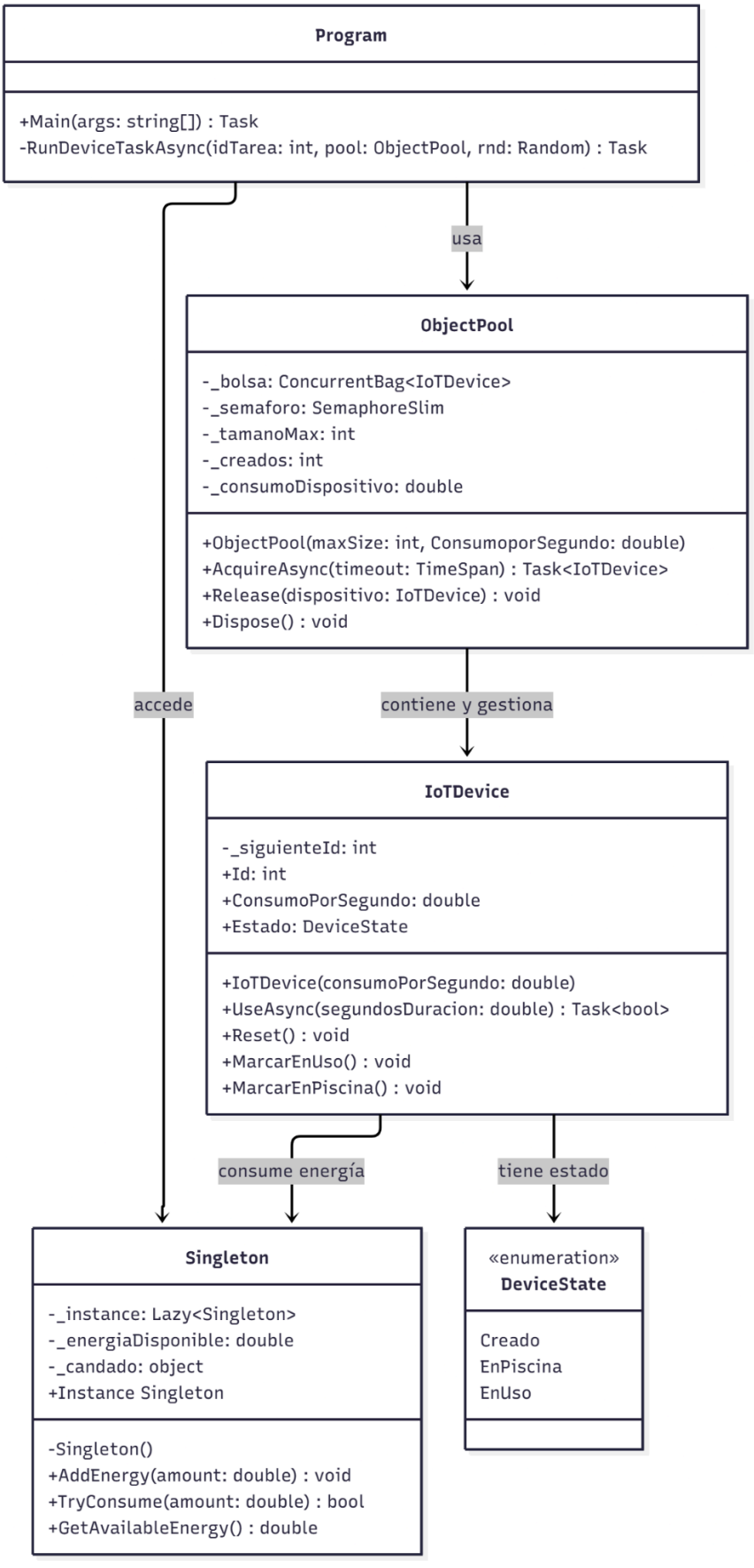
UNIDAD 2

NOMBRE Y NÚMERO DE CONTROL:
Ochoa Moran Victor Alejandro 22210329

NOMBRE DEL MAESTRO (A):

MARIBEL GUERRERO LUIS

Diagrama UML



Código

Program.cs

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Singleton (Central de Energía) + Object Pool
(Dispositivos IoT)");

        const int tamanoPool = 5; // máximo dispositivos simultáneos
reutilizables
        const double consumoPorSegundoDispositivo = 5.0; // unidades
por segundo
        using var pool = new ObjectPool(tamanoPool,
consumoPorSegundoDispositivo);

        // Inicializar la central de energía
        Singleton.Instance.AddEnergy(200.0);
        Console.WriteLine($"Energía inicial:
{Singleton.Instance.GetAvailableEnergy():F2}");

        // Simular peticiones de dispositivos
        var rnd = new Random();
        const int cantidadTareas = 5;
        var tareas = new Task[cantidadTareas];

        for (int i = 0; i < cantidadTareas; i++)
        {
            tareas[i] = RunDeviceTaskAsync(i, pool, rnd);
        }

        await Task.WhenAll(tareas);

        Console.WriteLine($"Simulación finalizada. Energía restante:
{Singleton.Instance.GetAvailableEnergy():F2}");
        Console.WriteLine("Fin.");
    }
```

```
    static async Task RunDeviceTaskAsync(int idTarea, ObjectPool pool,
Random rnd)
    {
        // pequeño retardo para aumentar concurrencia
        await Task.Delay(rnd.Next(0, 200));
        Console.WriteLine($"Tarea {idTarea}: solicitando
dispositivo...");

        var dispositivo = await
pool.AcquireAsync(TimeSpan.FromSeconds(5));
        if (dispositivo == null)
        {
            Console.WriteLine($"Tarea {idTarea}: timeout al adquirir
dispositivo.");
            return;
        }

        Console.WriteLine($"Tarea {idTarea}: adquirió dispositivo
#{dispositivo.Id} (consumo {dispositivo.ConsumoPorSegundo}/s)");
        var segundos = rnd.NextDouble() * 3.0 + 0.5;
        var exitoso = await dispositivo.UseAsync(segundos);
        if (exitoso)
        {
            Console.WriteLine($"Tarea {idTarea}: dispositivo
#{dispositivo.Id} usó {segundos:F2}s con éxito. Energía restante:
{Singleton.Instance.GetAvailableEnergy():F2}");
        }
        else
        {
            Console.WriteLine($"Tarea {idTarea}: dispositivo
#{dispositivo.Id} NO pudo consumir energía suficiente (requería
{dispositivo.ConsumoPorSegundo * segundos:F2}). Energía restante:
{Singleton.Instance.GetAvailableEnergy():F2}");
        }

        pool.Release(dispositivo);
        Console.WriteLine($"Tarea {idTarea}: liberó dispositivo
#{dispositivo.Id}");
    }
}
```

Object Pool.cs

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

public class ObjectPool : IDisposable
{
    private readonly ConcurrentBag<IoTDevice> _bolsa = new();
    // Semáforo: controla cuántas hay simultáneamente
    private readonly SemaphoreSlim _semaforo;
    // Tamaño máximo del pool
    private readonly int _tamanoMax;
    // Contador de objetos creados
    private int _creados = 0;
    // Consumo de los aparatos
    private readonly double _consumoDispositivo;

    public ObjectPool(int maxSize, double ConsumoporSegundo)
    {
        if (maxSize <= 0) throw new ArgumentException("",
nameof(maxSize));
        _tamanoMax = maxSize;
        _semaforo = new SemaphoreSlim(maxSize, maxSize);
        _consumoDispositivo = ConsumoporSegundo;
    }

    // aqui empieza la magia del Object Pool MÉTODO PRINCIPAL :D
    public async Task<IoTDevice?> AcquireAsync(TimeSpan timeout)
    {
        // Espera hasta tener permiso para continuar
        var acquired = await _semaforo.WaitAsync(timeout);
        if (!acquired) return null;

        // Reutilizar si hay uno disponible en la bolsa.
        if (_bolsa.TryTake(out var device))
        {
            device.MarcarEnUso();
            Console.WriteLine($"[Pool] Dispositivo #{device.Id} salió
de la piscina (reutilizado). Estado: {device.Estado}");
            return device;
        }
        // se ve si hay espacios libres
```

```
        var creadosAhora = Interlocked.Increment(ref _creados);
        if (creadosAhora <= _tamanoMax)
        {
            var nuevo = new IoTDevice(_consumoDispositivo);
            nuevo.MarcarEnUso();
            Console.WriteLine($"[Pool] Dispositivo #{nuevo.Id} creado y
sale de la piscina. Estado: {nuevo.Estado}");
            return nuevo;
        }


        // aqui se decrementa si se llego al limite y se espera hasta
que halla espacios nuevos
        Interlocked.Decrement(ref _creados);
        while (true)
        {
            if (_bolsa.TryTake(out device))
            {
                device.MarcarEnUso();
                Console.WriteLine($"[Pool] Dispositivo #{device.Id}
salió de la piscina (espera resuelta). Estado: {device.Estado}");
                return device;
            }
            await Task.Delay(20);
        }
    }

    // Devuelve un dispositivo al pool: resetea, marca y libera
    public void Release(IoTDevice dispositivo)
    {
        dispositivo.Reset();
        dispositivo.MarcarEnPiscina();
        _bolsa.Add(dispositivo);
        Console.WriteLine($"[Pool] Dispositivo #{dispositivo.Id}
devuelto a la piscina. Estado: {dispositivo.Estado}");
        _semaforo.Release();
    }

    //este metodo solo limpia el semaforo
    public void Dispose()
    {
        _semaforo?.Dispose();
    }
```

```
}
```

Singleton.cs

```
using System;

public sealed class Singleton
{
    // ESTA LÍNEA ES MAGIA: Garantiza que solo exista UN medidor
    // revisar por que se pone "=>" y poner en comentarios para que
sirve y por que lo puse que mo se te olvide !!!!!
    private static readonly Lazy<Singleton> _instance = new(() => new
Singleton());
    public static Singleton Instance => _instance.Value;

    // energía disponible en la central
    private double _energiaDisponible;
    // Candado para que no se mescle
    private readonly object _candado = new();

    // Constructor PRIVADO nadie más puede crear medidores
    private Singleton() => _energiaDisponible = 0;

    // Método para AGREGAR energía
    public void AddEnergy(double amount)
    {
        if (amount <= 0) return;
        lock (_candado)
        {
            _energiaDisponible += amount;
        }
    }

    // Intenta consumir energía y devuelve true si la operación tuvo
éxito.
    public bool TryConsume(double amount)
    {
        if (amount <= 0) return true;
        lock (_candado)
        {
            if (_energiaDisponible >= amount)
            {
                _energiaDisponible -= amount;
                return true;
            }
            return false;
```

```
        }
    }

    // lectura segura del estado de energía.
    public double GetAvailableEnergy()
    {
        lock (_candado)
        {
            return _energiaDisponible;
        }
    }
}
```

Dispositivos IoT.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;

public enum DeviceState
{
    Creado,
    EnPiscina,
    EnUso
}

// Dispositivo IoT reutilizable
public class IoTDevice
{
    // Contador de IDs
    private static int _siguienteId = 0;
    public int Id { get; }
    // Consumo por segundo del dispositivo
    public double ConsumoPorSegundo { get; }
    // Estado del dispositivo
    public DeviceState Estado { get; private set; }

    public IoTDevice(double consumoPorSegundo)
    {
        // Generar Id único
        Id = Interlocked.Increment(ref _siguienteId);
        ConsumoPorSegundo = consumoPorSegundo;
        Estado = DeviceState.Creado;
    }

    // Simula el uso del dispositivo durante 'segundosDuracion'
segundos
    // Devuelve true si la central pudo suministrar la energía
requerida
    public async Task<bool> UseAsync(double segundosDuracion)
    {
        if (segundosDuracion <= 0) return true;
        var energiaRequerida = ConsumoPorSegundo * segundosDuracion;

        await Task.Delay(TimeSpan.FromSeconds(segundosDuracion * 0.1));
        // Devuelve true si la central pudo restar la energía
        return Singleton.Instance.TryConsume(energiaRequerida);
```

```
    }

    public void Reset()
    {
        // Reseteo antes de ponerlo en la bolsa
        Estado = DeviceState.Creado;
    }

    public void MarcarEnUso()
    {
        // Marcar como en uso cuando el pool lo entrega a una tarea
        Estado = DeviceState.EnUso;
    }

    public void MarcarEnPiscina()
    {
        // Marcar como disponible en la piscina
        Estado = DeviceState.EnPiscina;
    }
}
```

Captura de pantalla de código en ejecución

```
Singleton (Central de Energía) + Object Pool (Dispositivos IoT)
Energía inicial: 200.00
Tarea 0: solicitando dispositivo...
Tarea 3: solicitando dispositivo...
[Pool] Dispositivo #2 creado y sale de la piscina. Estado: EnUso
[Pool] Dispositivo #1 creado y sale de la piscina. Estado: EnUso
Tarea 3: adquirió dispositivo #2 (consumo 5/s)
Tarea 0: adquirió dispositivo #1 (consumo 5/s)
Tarea 1: solicitando dispositivo...
[Pool] Dispositivo #3 creado y sale de la piscina. Estado: EnUso
Tarea 1: adquirió dispositivo #3 (consumo 5/s)
Tarea 4: solicitando dispositivo...
[Pool] Dispositivo #4 creado y sale de la piscina. Estado: EnUso
Tarea 4: adquirió dispositivo #4 (consumo 5/s)
Tarea 3: dispositivo #2 usó 0.57s con éxito. Energía restante: 197.15
[Pool] Dispositivo #2 devuelto a la piscina. Estado: EnPiscina
Tarea 3: liberó dispositivo #2
Tarea 2: solicitando dispositivo...
[Pool] Dispositivo #2 salió de la piscina (reutilizado). Estado: EnUso
Tarea 2: adquirió dispositivo #2 (consumo 5/s)
Tarea 4: dispositivo #4 usó 1.28s con éxito. Energía restante: 190.75
[Pool] Dispositivo #4 devuelto a la piscina. Estado: EnPiscina
Tarea 4: liberó dispositivo #4
Tarea 0: dispositivo #1 usó 2.35s con éxito. Energía restante: 179.01
[Pool] Dispositivo #1 devuelto a la piscina. Estado: EnPiscina
Tarea 0: liberó dispositivo #1
Tarea 1: dispositivo #3 usó 2.97s con éxito. Energía restante: 153.28
Tarea 2: dispositivo #2 usó 2.17s con éxito. Energía restante: 153.28
[Pool] Dispositivo #2 devuelto a la piscina. Estado: EnPiscina
Tarea 2: liberó dispositivo #2
[Pool] Dispositivo #3 devuelto a la piscina. Estado: EnPiscina
Tarea 1: liberó dispositivo #3
Simulación finalizada. Energía restante: 153.28
Fin.
```

Conclusión

La fusión de los patrones de diseño Singleton y Object Pool en este programa demuestra una aplicación sólida de buenas prácticas de arquitectura y concurrencia en C#.
El uso del Singleton garantiza un punto de acceso único y controlado para los recursos compartidos en este caso, la energía, evitando inconsistencias y duplicaciones de estado. Por su parte, el Object Pool optimiza el rendimiento al reutilizar objetos costosos de crear, reduciendo la carga de memoria y la sobrecarga del sistema.

Integrar ambos patrones dentro de un entorno asincrónico y concurrente, empleando bloqueos seguros (lock), estructuras concurrentes (ConcurrentBag) y control de acceso con SemaphoreSlim, refleja un manejo responsable de la sincronización y de los recursos del sistema.

En conjunto, esta implementación aplica principios esenciales de eficiencia, seguridad en hilos y reutilización de objetos, mostrando cómo la combinación adecuada de patrones de diseño puede mejorar la escalabilidad, estabilidad y mantenimiento de un sistema distribuido o multitarea.