# Guia de Estudio: Patron de Creación Object Pool

## 1. Introducción al Patrón Object Pool

**a) Definición**

El patrón **Object Pool** es un patrón de diseño creacional que permite la reutilización de objetos en lugar de crear nuevas instancias cada vez que se necesitan. Se usa cuando la creación de objetos es costosa en términos de rendimiento y recursos.

Este patrón mantiene un conjunto de objetos inicializados disponibles para su uso, y cuando un cliente solicita un objeto, en lugar de crear uno nuevo, se obtiene de la piscina (pool). Una vez que el objeto ya no es necesario, se devuelve al pool para ser reutilizado.

**b) ¿Cuándo utilizarlo?**

* Cuando la creación de objetos es costosa en términos de tiempo o recursos (por ejemplo, conexiones a bases de datos, conexiones de red, hilos de ejecución, etc.).
* Cuando los objetos son reutilizados frecuentemente en la aplicación.
* Cuando se necesita controlar la cantidad de instancias activas de un objeto para evitar sobrecarga en la memoria.

**c) ¿Cuándo NO utilizarlo?**

* Cuando los objetos son livianos y su creación no genera un impacto en el rendimiento.
* Cuando cada objeto necesita mantenerse en un estado único y personalizado, lo que podría hacer que su reutilización genere inconsistencias.
* Cuando se requiere un alto nivel de concurrencia sin mecanismos de sincronización bien definidos.

## 2. Caso de Uso: Conexiones a Bases de Datos en una Aplicación Financiera

**Historia de Usuario**

Como desarrollador de una aplicación bancaria, necesito un sistema que administre conexiones a la base de datos de manera eficiente, permitiendo la reutilización de conexiones activas en lugar de crear nuevas instancias cada vez que se realiza una consulta, para mejorar el rendimiento y reducir la carga en el servidor.

**Código en Java:**

import java.util.Queue;

import java.util.LinkedList;

// Clase que representa una conexión a la base de datos

class ConexionDB {

private String id;

public ConexionDB(String id) {

this.id = id;

System.out.println("Nueva conexión creada: " + id);

}

public void ejecutarConsulta(String consulta) {

System.out.println("Ejecutando consulta en " + id + ": " + consulta);

}

}

// Implementación del Object Pool

class PoolConexiones {

private Queue<ConexionDB> pool;

private int limite;

public PoolConexiones(int limite) {

this.limite = limite;

this.pool = new LinkedList<>();

for (int i = 0; i < limite; i++) {

pool.add(new ConexionDB("Conexion-" + (i + 1)));

}

}

public ConexionDB obtenerConexion() {

if (!pool.isEmpty()) {

return pool.poll();

} else {

System.out.println("No hay conexiones disponibles, espere...");

return null;

}

}

public void liberarConexion(ConexionDB conexion) {

pool.offer(conexion);

}

}

public class Main {

public static void main(String[] args) {

PoolConexiones pool = new PoolConexiones(2);

ConexionDB c1 = pool.obtenerConexion();

ConexionDB c2 = pool.obtenerConexion();

ConexionDB c3 = pool.obtenerConexion(); // No hay conexiones disponibles

c1.ejecutarConsulta("SELECT \* FROM usuarios");

pool.liberarConexion(c1);

ConexionDB c4 = pool.obtenerConexion(); // Ahora sí hay conexión disponible

c4.ejecutarConsulta("INSERT INTO transacciones VALUES (1, 100)");

}

}

**Salida esperada:**

Nueva conexión creada: Conexion-1

Nueva conexión creada: Conexion-2

Ejecutando consulta en Conexion-1: SELECT \* FROM usuarios

No hay conexiones disponibles, espere...

Ejecutando consulta en Conexion-1: INSERT INTO transacciones VALUES (1, 100)

**Explicación:** Se crean solo dos conexiones en el pool, y cuando una conexión se libera, puede reutilizarse en otra consulta en lugar de crear una nueva.

## 3. Identificación de Escenarios Donde Object Pool es Necesario

* **Gestión de conexiones a bases de datos.**
* **Gestión de hilos en aplicaciones concurrentes.**
* **Reutilización de objetos costosos en videojuegos (por ejemplo, balas o partículas en un motor gráfico).**
* **Administración de conexiones de red en servidores de alto tráfico.**

## 4. Antipatrones Relacionados con Object Pool

**a) No liberar los objetos al Pool (Fuga de Memoria)**

Si un objeto no es devuelto al pool después de su uso, el sistema terminará creando más instancias de las necesarias, causando **fugas de memoria** y consumo innecesario de recursos.

**Solución:** Asegurarse de que cada objeto utilizado sea devuelto al pool después de su uso.

**b) Sincronización Deficiente en Entornos Concurrentes**

Si múltiples hilos acceden al pool sin mecanismos de sincronización, pueden ocurrir **condiciones de carrera** y corrupción de datos.

**Solución:** Usar estructuras de datos seguras para concurrencia como ConcurrentLinkedQueue en Java o bloquear el acceso con synchronized.

**c) Crear un Pool Demasiado Grande**

Si el tamaño del pool es mayor al necesario, se consumirá más memoria de la necesaria y se podrían mantener objetos inactivos sin ser utilizados.

**Solución:** Analizar el número óptimo de objetos a mantener en el pool según las necesidades del sistema.

## 5. Conclusión

El patrón **Object Pool** es una estrategia efectiva para gestionar objetos costosos y mejorar el rendimiento de una aplicación. Sin embargo, debe implementarse con cuidado para evitar fugas de memoria, problemas de sincronización y uso excesivo de recursos.

# Taller Ejercicios Prácticos

## Implementación del Patrón Object Pool en Turismo, Recreación y Medicina

### Caso 1: Sistema de Gestión de Transporte en Turismo

#### Historia de Usuario

Como operador de una agencia de turismo, necesito un sistema que administre un conjunto de vehículos turísticos para que puedan ser asignados a clientes cuando estén disponibles y devueltos al sistema una vez finalizado el recorrido, optimizando el uso de los recursos sin necesidad de crear nuevas instancias de vehículos.

**Código en Java:**

import java.util.Queue;

import java.util.LinkedList;

class VehiculoTuristico {

private String id;

public VehiculoTuristico(String id) {

this.id = id;

System.out.println("Nuevo vehículo disponible: " + id);

}

public void asignarRecorrido(String recorrido) {

System.out.println("Vehículo " + id + " asignado a " + recorrido);

}

}

class PoolVehiculos {

private Queue<VehiculoTuristico> pool;

private int limite;

public PoolVehiculos(int limite) {

this.limite = limite;

this.pool = new LinkedList<>();

for (int i = 0; i < limite; i++) {

pool.add(new VehiculoTuristico("Vehiculo-" + (i + 1)));

}

}

public VehiculoTuristico obtenerVehiculo() {

if (!pool.isEmpty()) {

return pool.poll();

} else {

System.out.println("No hay vehículos disponibles, espere...");

return null;

}

}

public void liberarVehiculo(VehiculoTuristico vehiculo) {

pool.offer(vehiculo);

}

}

public class Main {

public static void main(String[] args) {

PoolVehiculos pool = new PoolVehiculos(2);

VehiculoTuristico v1 = pool.obtenerVehiculo();

VehiculoTuristico v2 = pool.obtenerVehiculo();

VehiculoTuristico v3 = pool.obtenerVehiculo(); // No hay vehículos disponibles

v1.asignarRecorrido("Tour por la ciudad");

pool.liberarVehiculo(v1);

VehiculoTuristico v4 = pool.obtenerVehiculo(); // Ahora hay uno disponible

v4.asignarRecorrido("Visita a las montañas");

}

}

### Caso 2: Gestión de Equipamiento en un Parque de Diversiones

#### Historia de Usuario

Como administrador de un parque de diversiones, necesito un sistema para administrar los equipos de realidad virtual (VR), de manera que los visitantes puedan usarlos cuando estén disponibles y devolverlos para su reutilización sin necesidad de comprar más dispositivos de los necesarios.

**Código en Java:**

import java.util.Queue;

import java.util.LinkedList;

class EquipoVR {

private String id;

public EquipoVR(String id) {

this.id = id;

System.out.println("Nuevo equipo VR disponible: " + id);

}

public void asignarUsuario(String usuario) {

System.out.println("Equipo VR " + id + " asignado a " + usuario);

}

}

class PoolEquiposVR {

private Queue<EquipoVR> pool;

private int limite;

public PoolEquiposVR(int limite) {

this.limite = limite;

this.pool = new LinkedList<>();

for (int i = 0; i < limite; i++) {

pool.add(new EquipoVR("VR-" + (i + 1)));

}

}

public EquipoVR obtenerEquipo() {

if (!pool.isEmpty()) {

return pool.poll();

} else {

System.out.println("No hay equipos VR disponibles, espere...");

return null;

}

}

public void liberarEquipo(EquipoVR equipo) {

pool.offer(equipo);

}

}

public class Main {

public static void main(String[] args) {

PoolEquiposVR pool = new PoolEquiposVR(3);

EquipoVR e1 = pool.obtenerEquipo();

EquipoVR e2 = pool.obtenerEquipo();

e1.asignarUsuario("Carlos");

pool.liberarEquipo(e1);

EquipoVR e3 = pool.obtenerEquipo();

e3.asignarUsuario("Ana");

}

}

### Caso 3: Gestión de Camillas en un Hospital

#### Historia de Usuario

Como director de un hospital, necesito administrar las camillas de urgencias para que puedan ser asignadas a pacientes cuando sea necesario y liberadas una vez finalizada la atención, evitando la sobreasignación de camillas y asegurando su reutilización eficiente.

**Código en Java:**

import java.util.Queue;

import java.util.LinkedList;

class Camilla {

private String id;

public Camilla(String id) {

this.id = id;

System.out.println("Nueva camilla disponible: " + id);

}

public void asignarPaciente(String paciente) {

System.out.println("Camilla " + id + " asignada a paciente " + paciente);

}

}

class PoolCamillas {

private Queue<Camilla> pool;

private int limite;

public PoolCamillas(int limite) {

this.limite = limite;

this.pool = new LinkedList<>();

for (int i = 0; i < limite; i++) {

pool.add(new Camilla("Camilla-" + (i + 1)));

}

}

public Camilla obtenerCamilla() {

if (!pool.isEmpty()) {

return pool.poll();

} else {

System.out.println("No hay camillas disponibles, espere...");

return null;

}

}

public void liberarCamilla(Camilla camilla) {

pool.offer(camilla);

}

}

public class Main {

public static void main(String[] args) {

PoolCamillas pool = new PoolCamillas(3);

Camilla c1 = pool.obtenerCamilla();

Camilla c2 = pool.obtenerCamilla();

c1.asignarPaciente("María");

pool.liberarCamilla(c1);

Camilla c3 = pool.obtenerCamilla();

c3.asignarPaciente("Pedro");

}

}