# INFORME DE LABORATORIO

| INFORMACIÓN BÁSICA | | | | | |
|---|---|---|---|---|---|
| **ASIGNATURA:** | Tecnologia de objetos | | | | |
| **TÍTULO DE LA PRÁCTICA:** | **Ejercicios -Singleton** | | | | |
| **NÚMERO DE PRÁCTICA:** | *07* | **AÑO LECTIVO:** | *2025* | **NRO. SEMESTRE:** | *VI* |
| **FECHA DE PRESENTACIÓN** | *15-11-25* | **HORA DE PRESENTACIÓN** | | | |
| **INTEGRANTE (s):**<br>Anco Aymara Jean Pierre<br>Suasaca Pacompia Alvaro Gustavo<br>Valdiviezo Tovar Alexander | | | | **NOTA:** | |
| **DOCENTE(s):**<br>*Edith Giovanna Cano Mamani* | | | | | |

## SOLUCIÓN Y RESULTADOS

### I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS

**Repositorio:** [https://github.com/Alexandervt8/to](https://github.com/Alexandervt8/to)

**Ejercicio 01 (Básico): Implementación directa del Singleton**

Crear una clase **Configuracion** que almacene las configuraciones generales del sistema (por ejemplo, idioma y zona horaria).

- Asegúrate de que solo exista una instancia de esta clase.
- Añade un método **mostrar_configuracion()** que imprima los valores actuales.
- Verifica en el main que aunque crees varios objetos, todos apunten al mismo.

```cpp
#include <iostream>
#include <string>

class Configuracion {
private:
    std::string idioma;
    std::string zonaHoraria;

    static Configuracion* instancia;
```

```
    Configuracion() : idioma("es-PE"), zonaHoraria("America/Lima") {}

    Configuracion(const Configuracion&) = delete;
    Configuracion& operator=(const Configuracion&) = delete;

public:
    static Configuracion* getInstancia() {
        if (instancia == nullptr) {
            instancia = new Configuracion();
        }
        return instancia;
    }

    void setIdioma(const std::string& i) { idioma = i; }
    void setZonaHoraria(const std::string& z) { zonaHoraria = z; }

    void mostrar_configuracion() const {
        std::cout << "Idioma: " << idioma << "\n";
        std::cout << "Zona horaria: " << zonaHoraria << "\n";
    }
};

Configuracion* Configuracion::instancia = nullptr;

int main() {
    Configuracion* c1 = Configuracion::getInstancia();
    Configuracion* c2 = Configuracion::getInstancia();

    c1->setIdioma("es-PE");
    c1->setZonaHoraria("America/Lima");

    std::cout << "Configuracion desde c1:\n";
    c1->mostrar_configuracion();

    std::cout << "\nConfiguracion desde c2 :\n";
    c2->mostrar_configuracion();

    std::cout << "\nDirecciones de memoria:\n";
    std::cout << "c1: " << c1 << "\n";
    std::cout << "c2: " << c2 << "\n";

    return 0;
}
```

**Salida:**

```
 *  Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

Configuracion desde c1:
Idioma: es-PE
Zona horaria: America/Lima

Configuracion desde c2 :
Idioma: es-PE
Zona horaria: America/Lima

Direcciones de memoria:
c1: 0x19968141a50
c2: 0x19968141a50
```

**Explicacion:**

El código implementa el patrón de diseño Singleton mediante la clase Configuracion, asegurando que solo exista una única instancia durante toda la ejecución del programa. Para lograrlo, el constructor es privado y se utiliza un puntero estático instancia que se inicializa solo la primera vez que se llama a getInstancia(). Además, se bloquea la copia de objetos deshabilitando el constructor de copia y el operador de asignación, lo que evita que se generen instancias duplicadas. La clase almacena dos valores: idioma y zona horaria, con métodos para modificarlos y mostrarlos.

En el main(), se solicitan dos punteros (c1 y c2) a la instancia única de la clase. Ambos reciben exactamente el mismo objeto, lo cual se demuestra al imprimir la configuración y las direcciones de memoria. Cualquier cambio realizado a través de c1 también se refleja en c2, ya que ambos apuntan al mismo espacio en memoria. Esto confirma correctamente el funcionamiento del patrón Singleton en este ejemplo.

**Ejercicio 02 (Intermedio): Singleton con recursos compartidos**

Implemente una clase **Logger** que registre mensajes de log en un archivo de texto (por ejemplo **bitacora.log**).

- Solo debe existir una instancia de Logger.
- Implementa un método log(mensaje) que agregue el texto al archivo con la hora actual.
- Prueba el logger desde distintos puntos del programa y confirma que todos usan el mismo archivo.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <ctime>
#include <iomanip>
```

```
class Logger {
private:
    static Logger* instancia;
    std::string nombreArchivo;

    Logger() : nombreArchivo("bitacora.log") {}

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    std::string obtenerHoraActual() {
        using namespace std::chrono;
        auto ahora = system_clock::now();
        std::time_t t = system_clock::to_time_t(ahora);
        std::tm tmLocal = *std::localtime(&t);

        std::ostringstream oss;
        oss << std::put_time(&tmLocal, "%Y-%m-%d %H:%M:%S");
        return oss.str();
    }

public:
    static Logger* getInstancia() {
        if (instancia == nullptr) {
            instancia = new Logger();
        }
        return instancia;
    }

    void log(const std::string& mensaje) {
        std::ofstream archivo(nombreArchivo, std::ios::app);
        if (archivo.is_open()) {
            archivo << "[" << obtenerHoraActual() << "] " << mensaje <<
"\n";
        }
    }
};

Logger* Logger::instancia = nullptr;

void moduloRed() {
    Logger::getInstancia()->log("Mensaje desde moduloRed");
}

void moduloUI() {
    Logger::getInstancia()->log("Mensaje desde moduloUI");
}

void moduloNegocio() {
    Logger::getInstancia()->log("Mensaje desde moduloNegocio");
}
```

```cpp
int main() {
    Logger* l1 = Logger::getInstancia();
    Logger* l2 = Logger::getInstancia();

    l1->log("Inicio del programa");
    moduloRed();
    moduloUI();
    moduloNegocio();
    l2->log("Fin del programa");

    std::cout << "Se han escrito mensajes en bitacora.log\n";
    std::cout << "Logger l1: " << l1 << "\n";
    std::cout << "Logger l2: " << l2 << "\n";

    return 0;
}
```

**Salida:**

```
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

Se han escrito mensajes en bitacora.log
Logger l1: 0x19cff401a50
Logger l2: 0x19cff401a50
```

**Explicacion:**

El programa implementa un Logger utilizando el patrón Singleton para asegurar que solo exista una única instancia encargada de escribir mensajes en un archivo de bitácora (bitacora.log). El constructor es privado, lo que evita la creación de instancias externas, y el acceso se realiza mediante el método estático getInstancia(). Este método verifica si la instancia ya existe; si no, la crea y luego la retorna. Cada vez que se llama al método log(), se genera una entrada en el archivo con la hora actual formateada y el mensaje correspondiente.

En el main, así como en los módulos moduloRed, moduloUI y moduloNegocio, todos utilizan la misma instancia del Logger para escribir mensajes, lo cual garantiza coherencia y centralización del registro. Finalmente, el programa imprime las direcciones de memoria de l1 y l2, demostrando que ambos son el mismo objeto, validando así el correcto funcionamiento del patrón Singleton.

**Ejercicio 03 (Aplicado): Conexión simulada a base de datos**

Crea una clase **ConexionBD** que simule la conexión a una base de datos.

- Solo debe haber una conexión activa (Singleton).
- Incluye métodos como conectar(), desconectar() y estado().
- Si alguien intenta crear otra conexión, debe devolverse la ya existente.

```cpp
#include <iostream>
#include <string>

class ConexionBD {
private:
    static ConexionBD* instancia;
    bool conectada;
    std::string nombreBD;

    ConexionBD() : conectada(false), nombreBD("mi_basedatos") {}

    ConexionBD(const ConexionBD&) = delete;
    ConexionBD& operator=(const ConexionBD&) = delete;

public:
    static ConexionBD* getInstancia() {
        if (instancia == nullptr) {
            instancia = new ConexionBD();
        }
        return instancia;
    }

    void conectar() {
        if (!conectada) {
            std::cout << "Conectando a la BD: " << nombreBD << "...\n";
            conectada = true;
            std::cout << "Conexion establecida.\n";
        } else {
            std::cout << "Ya existe una conexion activa.\n";
        }
    }

    void desconectar() {
        if (conectada) {
            std::cout << "Desconectando de la BD...\n";
            conectada = false;
            std::cout << "Conexion cerrada.\n";
        } else {
            std::cout << "No hay conexion activa para cerrar.\n";
        }
    }

    void estado() const {
        std::cout << "Estado de la conexion: "
                  << (conectada ? "Conectada" : "Desconectada") << "\n";
    }
};

ConexionBD* ConexionBD::instancia = nullptr;
```

```cpp
int main() {
    ConexionBD* c1 = ConexionBD::getInstancia();
    ConexionBD* c2 = ConexionBD::getInstancia();

    c1->estado();
    c1->conectar();
    c2->estado();
    std::cout << "Direcciones de memoria:\n";
    std::cout << "c1: " << c1 << "\n";
    std::cout << "c2: " << c2 << "\n";
    c2->desconectar();
    c1->estado();

    return 0;
}
```

**Salida:**

```
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

Estado de la conexion: Desconectada
Conectando a la BD: mi_basedatos...
Conexion establecida.
Estado de la conexion: Conectada
Direcciones de memoria:
c1: 0x1f53e871a50
c2: 0x1f53e871a50
Desconectando de la BD...
Conexion cerrada.
Estado de la conexion: Desconectada
```

**Explicacion:**

Este programa implementa una clase llamada ConexionBD, que administra una conexión simulada a una base de datos. Para asegurar que solo exista una única conexión activa durante toda la ejecución del software, la clase aplica el patrón Singleton: el constructor es privado, y el acceso se realiza mediante el método estático getInstancia(). Dentro de la clase se manejan operaciones como conectar, desconectar y verificar el estado, manteniendo una lógica segura y centralizada de control de conexión.

En la función main(), se observa cómo dos punteros (c1 y c2) solicitan la instancia mediante getInstancia(). Ambos reciben el mismo objeto en memoria, lo cual se confirma al imprimir sus direcciones. Cuando uno de los punteros establece o cierra la conexión, el otro refleja ese mismo estado, demostrando claramente que se trata de una única instancia compartida. Con esto, el código ejemplifica de manera sencilla y efectiva el funcionamiento del patrón Singleton aplicado a sistemas de conexión.

**Ejercicio 04 (Avanzado): Singleton en un juego**

Desarrolla una clase ControlJuego para manejar el estado global de un juego (nivel actual, puntaje, vidas).

• Usa Singleton para que todos los componentes (jugador, enemigos, interfaz) consulten y modifiquen el mismo estado.

• Simula en el main cómo distintos módulos acceden al mismo ControlJuego.

```cpp
#include <iostream>
#include <string>

class ControlJuego {
private:
    static ControlJuego* instancia;
    int nivel;
    int puntaje;
    int vidas;
    ControlJuego() : nivel(1), puntaje(0), vidas(3) {}
    ControlJuego(const ControlJuego&) = delete;
    ControlJuego& operator=(const ControlJuego&) = delete;

public:
    static ControlJuego* getInstancia() {
        if (instancia == nullptr) {
            instancia = new ControlJuego();
        }
        return instancia;
    }

    void aumentarPuntaje(int p) { puntaje += p; }
    void perderVida() { if (vidas > 0) vidas--; }
    void siguienteNivel() { nivel++; }

    void mostrarEstado() const {
        std::cout << "=== Estado del Juego ===\n";
        std::cout << "Nivel: " << nivel << "\n";
        std::cout << "Puntaje: " << puntaje << "\n";
        std::cout << "Vidas: " << vidas << "\n";
        std::cout << "========================\n";
    }
};

ControlJuego* ControlJuego::instancia = nullptr;
// ---------------- MODULOS DEL JUEGO ----------------
void moduloJugador() {
    ControlJuego::getInstancia()->aumentarPuntaje(50);
    std::cout << "[Jugador] Gana 50 puntos.\n";
}

void moduloEnemigo() {
    ControlJuego::getInstancia()->perderVida();
```

```
    std::cout << "[Enemigo] El jugador pierde una vida.\n";
}

void moduloInterfaz() {
    std::cout << "[UI] Mostrando estado en pantalla...\n";
    ControlJuego::getInstancia()->mostrarEstado();
}
// ---------------- MAIN ----------------
int main() {
    ControlJuego* c1 = ControlJuego::getInstancia();
    ControlJuego* c2 = ControlJuego::getInstancia();
    std::cout << "Instancia c1: " << c1 << "\n";
    std::cout << "Instancia c2: " << c2 << "\n\n";
    moduloJugador();
    moduloInterfaz();
    moduloEnemigo();
    moduloInterfaz();
    std::cout << "[Sistema] Pasando al siguiente nivel...\n";
    c1->siguienteNivel();
    moduloInterfaz();
    return 0;
}
```

**Salida:**

```
E:\to\lab07\eje01>g++ ejercicio4.cpp -o ejercicio4.exe

E:\to\lab07\eje01>ejercicio4.exe
Instancia c1: 0x10b79e8
Instancia c2: 0x10b79e8

[Jugador] Gana 50 puntos.
[UI] Mostrando estado en pantalla...
=== Estado del Juego ===
Nivel: 1
Puntaje: 50
Vidas: 3
========================
[Enemigo] El jugador pierde una vida.
[UI] Mostrando estado en pantalla...
=== Estado del Juego ===
Nivel: 1
Puntaje: 50
Vidas: 2
========================
[Sistema] Pasando al siguiente nivel...
[UI] Mostrando estado en pantalla...
=== Estado del Juego ===
Nivel: 2
Puntaje: 50
Vidas: 2
========================
```

**Explicacion:**

En este ejercicio, la clase ControlJuego utiliza el patrón Singleton para mantener un único estado global del juego. Contiene variables que representan el nivel, puntaje y número de vidas, junto con métodos para modificarlas. Como el constructor es privado, solo puede obtenerse la instancia mediante getInstancia(), asegurando que todos los módulos del juego compartan el mismo objeto.

En el main, se simulan tres módulos: jugador, enemigo e interfaz. Cada uno accede a ControlJuego y modifica el mismo estado. Esto permite que cualquier cambio (sumar puntos, perder vidas, avanzar de nivel) sea reflejado inmediatamente en todos los módulos, demostrando el uso correcto del patrón Singleton en el contexto de un juego.

**Ejercicio 05 (Desafío): Singleton con subprocesos (thread-safe)**

Modifica el ejercicio 3 (conexión a BD) o el 2 (Logger) para hacerlo seguro en entornos multihilo.

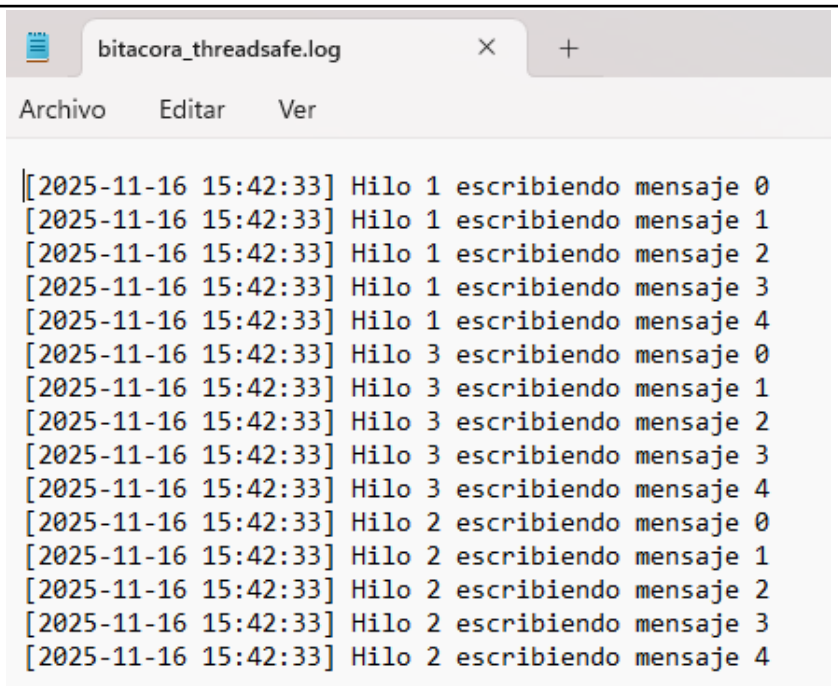
• Usa bloqueo o una verificación doble (double-checked locking) para evitar que se creen múltiples instancias en hilos distintos.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <ctime>
#include <iomanip>
#include <mutex>
#include <thread>

class Logger {
private:
    static Logger* instancia;
    static std::mutex mtx;
    std::string nombreArchivo;
    Logger() : nombreArchivo("bitacora_threadsafe.log") {}
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    std::string obtenerHoraActual() {
        using namespace std::chrono;
        auto ahora = system_clock::now();
        std::time_t t = system_clock::to_time_t(ahora);
        std::tm tmLocal = *std::localtime(&t);
        std::ostringstream oss;
        oss << std::put_time(&tmLocal, "%Y-%m-%d %H:%M:%S");
        return oss.str();
    }

public:
    static Logger* getInstancia() {
        if (instancia == nullptr) {
```

```
            std::lock_guard<std::mutex> lock(mtx);
            if (instancia == nullptr) {
                instancia = new Logger();
            }
        }
        return instancia;
    }

    void log(const std::string& mensaje) {
        std::lock_guard<std::mutex> lock(mtx);
        std::ofstream archivo(nombreArchivo, std::ios::app);
        if (archivo.is_open()) {
            archivo << "[" << obtenerHoraActual() << "] " << mensaje <<
"\n";
        }
    }
};

Logger* Logger::instancia = nullptr;
std::mutex Logger::mtx;
// --- Funcion usada por multiples hilos ---
void escribirDesdeHilo(const std::string& nombreHilo) {
    for (int i = 0; i < 5; i++) {
        Logger::getInstancia()->log(nombreHilo + " escribiendo mensaje " +
std::to_string(i));
    }
}

int main() {
    std::thread t1(escribirDesdeHilo, "Hilo 1");
    std::thread t2(escribirDesdeHilo, "Hilo 2");
    std::thread t3(escribirDesdeHilo, "Hilo 3");
    t1.join();
    t2.join();
    t3.join();
    std::cout << "Mensajes escritos en bitacora_threadsafe.log\n";
    return 0;
}
```

**Salida:**

```
E:\to\lab07\eje05>g++ ejercicio5.cpp -o ejercicio5.exe

E:\to\lab07\eje05>ejercicio.exe
"ejercicio.exe" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

E:\to\lab07\eje05>ejercicio5.exe
Mensajes escritos en bitacora_threadsafe.log
```




**Explicacion:**
En este ejercicio, la clase ControlJuego utiliza el patrón Singleton para mantener un único estado global del juego. Contiene variables que representan el nivel, puntaje y número de vidas, junto con métodos para modificarlas. Como el constructor es privado, solo puede obtenerse la instancia mediante getInstancia(), asegurando que todos los módulos del juego compartan el mismo objeto.

En el main, se simulan tres módulos: jugador, enemigo e interfaz. Cada uno accede a ControlJuego y modifica el mismo estado. Esto permite que cualquier cambio (sumar puntos, perder vidas, avanzar de nivel) sea reflejado inmediatamente en todos los módulos, demostrando el uso correcto del patrón Singleton en el contexto de un juego.

## II. SOLUCIÓN DEL CUESTIONARIO

**1. ¿Qué desventajas tiene el patrón Singleton en pruebas unitarias?**
El patrón Singleton puede dificultar las pruebas unitarias porque mantiene un estado global compartido entre todas las pruebas. Esto puede generar comportamientos inesperados si una prueba modifica el estado y otra depende de él, causando resultados inconsistentes. Además, es complicado sustituir un Singleton por un objeto simulado (mock), ya que su creación está controlada y no puede reemplazarse fácilmente durante la ejecución.

**2. ¿Cuándo no es recomendable usar Singleton?**
No es recomendable usar Singleton cuando se requiere flexibilidad, independencia entre módulos, o cuando la aplicación pueda necesitar múltiples instancias de una clase dependiendo del contexto.

Tampoco se recomienda en sistemas que deben evitar estados globales, ya que Singleton incrementa el acoplamiento y dificulta el mantenimiento, la escalabilidad y las pruebas. Si la clase cambia de estado con frecuencia, usar un Singleton puede provocar efectos secundarios difíciles de rastrear.

**3. ¿Cómo se diferencia de una clase estática?**

La principal diferencia es que un Singleton crea exactamente una instancia de la clase, mientras que una clase estática no crea instancias en absoluto. El Singleton puede implementar interfaces, soporta herencia y permite control sobre el ciclo de vida del objeto. En cambio, una clase estática solo ofrece métodos y atributos estáticos, no puede heredarse, no admite polimorfismo y su comportamiento es menos flexible. En resumen, el Singleton es un objeto real, y la clase estática no.

## III. CONCLUSIONES

- El desarrollo de los ejercicios permitió comprender cómo el patrón Singleton garantiza que solo exista una única instancia de una clase dentro de un programa. A través de ejemplos prácticos como configuraciones globales, conexiones a bases de datos y sistemas de registro (Logger), se evidenció que este patrón es útil para centralizar estados o recursos que deben ser compartidos por múltiples módulos. Su implementación básica consiste en un constructor privado, una instancia estática y un método de acceso controlado.

- Además, se observó que, aunque el Singleton es una solución conveniente en muchos casos, también implica riesgos cuando se trabaja en entornos complejos. Por ejemplo, en pruebas unitarias puede generar dependencias invisibles y efectos secundarios debido a su estado global persistente. Asimismo, su uso excesivo puede aumentar el acoplamiento entre componentes y dificultar la escalabilidad o la reutilización de código. Esto demuestra que debe aplicarse con criterio y solo cuando realmente existe la necesidad de una única instancia compartida.

- Finalmente, la versión multihilo del Singleton permitió comprender desafíos más avanzados, como la importancia del manejo de concurrencia mediante mutex y técnicas como el “double-checked locking”. Esta adaptación mostró que un diseño correcto evita la creación accidental de múltiples instancias cuando varias tareas acceden simultáneamente al recurso. En conjunto, todos los ejercicios reforzaron la comprensión de cómo aplicar patrones de diseño de manera segura, ordenada y adecuada al contexto del software.

## RETROALIMENTACIÓN GENERAL

## REFERENCIAS Y BIBLIOGRAFÍA

*Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.*

*Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.*

*Shalloway, A., & Trott, J. R. (2005). Design patterns explained: A new perspective on object-oriented design (2nd ed.). Addison-Wesley Professional.*