¿Qué es el Patrón Singleton con Fiabilidad de Hilos?

El **patrón Singleton** es un patrón de diseño que garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. La **fiabilidad de hilos** (thread safety) asegura que este patrón funcione correctamente en un entorno multihilo, evitando condiciones de carrera y garantizando que solo se cree una instancia incluso cuando múltiples hilos intenten acceder simultáneamente.

Implementación en el Código

El patrón Singleton con fiabilidad de hilos está implementado en la clase FileLogger:

FileLogger.java

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  private FileLogger() {
  }
  public static synchronized FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt",msg, true, true);
  }
}
```

Elementos Clave de la Implementación:

1. **Variable estática privada**: private static FileLogger logger;
   - Almacena la única instancia de la clase
   - Es estática para que sea compartida por todos los hilos
2. **Constructor privado**: private FileLogger()
   - Impide la creación de instancias desde fuera de la clase
   - Solo permite la creación interna
3. **Método getter sincronizado**: public static synchronized FileLogger getFileLogger()
   - El modificador synchronized garantiza que solo un hilo pueda ejecutar este método a la vez
   - Implementa la lógica de "lazy initialization" (inicialización perezosa)
   - Crea la instancia solo cuando se solicita por primera vez
4. **Método log sincronizado**: public synchronized void log(String msg)
   - Garantiza que las operaciones de escritura sean thread-safe

- Evita que múltiples hilos escriban simultáneamente al archivo

Funcionamiento en el Programa

En ClientManager.java, se crean 4 hilos que compiten por acceder al mismo logger:

ClientManager.java

```
FileProcess proceso1 = new FileProcess("Thread 1 is writting");
proceso1.start();
FileProcess proceso2 = new FileProcess("Thread 2 is writting");
proceso2.start();
FileProcess proceso3 = new FileProcess("Thread 3 is writting");
proceso3.start();
FileProcess proceso4 = new FileProcess("Thread 4 is writting");
proceso4.start();
```

Cada hilo:

1. Obtiene la misma instancia de FileLogger mediante FileLogger.getFileLogger()
2. Escribe 100 mensajes al archivo log.txt
3. Mide los tiempos de ejecución para análisis de rendimiento

Ventajas de esta Implementación:

- **Thread Safety**: El uso de synchronized previene condiciones de carrera
- **Lazy Initialization**: La instancia se crea solo cuando es necesaria
- **Acceso Global**: Todos los hilos comparten la misma instancia del logger
- **Consistencia**: Garantiza que todos los mensajes se escriban al mismo archivo

Este patrón es especialmente útil en este contexto porque permite que múltiples hilos escriban de forma segura a un recurso compartido (el archivo de log) sin corromper los datos.

¿Qué es el Patrón de Inicialización Temprana?

El patrón de Inicialización Temprana es una variante del patrón Singleton que garantiza que la instancia única se cree en el momento de la carga de la clase, antes de que cualquier hilo pueda acceder a ella. Esto elimina completamente la posibilidad de condiciones de carrera durante la inicialización.

Cómo funciona en este programa

1. Implementación en FileLogger.java

FileLogger.java

```
public synchronized void log(String msg) {
```

El patrón se implementa en la clase FileLogger de la siguiente manera:

FileLogger.java

```
private static FileLogger logger = new FileLogger();
```

Características clave:

1. **Inicialización estática: La instancia logger se declara como static y se inicializa inmediatamente al cargar la clase**
2. **Constructor privado: El constructor es privado para evitar instanciación externa**
3. **Método de acceso estático: getFileLogger() proporciona acceso a la instancia única**
4. **Thread-safe por diseño: Al crearse durante la carga de la clase, no hay posibilidad de condiciones de carrera**

2. Uso en el programa

En ClientManager.java, múltiples hilos acceden simultáneamente al logger:

ClientManager.java

```
FileProcess proceso1 = new FileProcess("Thread 1 is writting");
FileProcess proceso2 = new FileProcess("Thread 2 is writting");
FileProcess proceso3 = new FileProcess("Thread 3 is writting");
FileProcess proceso4 = new FileProcess("Thread 4 is writting");
```

Cada hilo obtiene la misma instancia del logger:

ClientManager.java

```
Logger fileLogger = FileLogger.getFileLogger();
```

Y la utiliza para escribir mensajes de forma sincronizada:

ClientManager.java

```
for (int i = 0; i < 100; i++) {
   fileLogger.log(msgLog);
}
```

Ventajas de esta implementación

1. **Thread-safety garantizada: La instancia se crea antes de que cualquier hilo pueda acceder**
2. **Simplicidad: No requiere sincronización adicional en el método getFileLogger()**
3. **Rendimiento: Acceso directo sin verificación de sincronización**
4. **Determinismo: La inicialización siempre ocurre en el mismo momento**

Medición de rendimiento

El programa incluye mediciones de tiempo para evaluar el rendimiento:

ClientManager.java

```
long tiFL = System.nanoTime(); // Antes de Obtener el FileLogger
Logger fileLogger = FileLogger.getFileLogger();
```

ClientManager.java

```
long tfFL = System.nanoTime(); // Tras Obtener el FileLogger
long tTotalFL = tfFL - tiFL; // Tiempo total para obtener el FileLogger
```

Esto permite comparar el tiempo de acceso al logger con el tiempo de escritura de mensajes, demostrando la eficiencia del patrón de inicialización temprana.

Conclusión

El patrón de Inicialización Temprana en este programa garantiza que todos los hilos compartan la misma instancia de FileLogger de forma segura, eliminando cualquier posibilidad de crear múltiples instancias o condiciones de carrera durante la inicialización. El método log() está sincronizado para garantizar que las escrituras al archivo sean thread-safe.

¿Qué es el Patrón Suspensión Condicionada?

El patrón Suspensión Condicionada es un patrón de concurrencia que permite que un hilo se suspenda hasta que se cumpla una condición específica. El hilo espera de forma segura hasta que la condición se vuelve verdadera.

Implementación en el Programa

1. Condición de Guarda

GSTest.java

```
public synchronized void park(String member) {
  while (totalParkedCars >= MAX_CAPACITY) {
   try {
    System.out.println(" The parking lot is full " +
       member + " has to wait ");
    wait();
```

La condición de guarda es totalParkedCars >= MAX_CAPACITY. Cuando el estacionamiento está lleno (4 coches), los hilos que intentan estacionar deben esperar.

2. Mecanismo de Espera

GSTest.java

```
     wait();
   } catch (InterruptedException e) {
    //
   }
  }
```

- wait(): Suspende el hilo actual hasta que otro hilo llame a notify() o notifyAll()
- El hilo se despierta automáticamente cuando se libera un espacio

3. Notificación de Liberación

GSTest.java

```
public synchronized void leave(String member) {
  totalParkedCars = totalParkedCars - 1;
  System.out.println(member +
```

" has left, notify a waiting member");

notify();

Cuando un miembro sale del estacionamiento:

- Se decrementa el contador de coches
- Se llama a notify() para despertar a un hilo en espera

4. Bucle While para Re-verificación

GSTest.java

while (totalParkedCars >= MAX_CAPACITY) {

El uso de while en lugar de if es crucial porque:

- Garantiza que la condición se verifique nuevamente después de despertar
- Previene condiciones de carrera donde múltiples hilos podrían despertar simultáneamente

¿Por qué se usa este patrón aquí?

Problema que resuelve:

1. **Recurso limitado**: Solo hay 4 espacios de estacionamiento
2. **Acceso concurrente**: Múltiples hilos (miembros) intentan estacionar simultáneamente
3. **Coordinación**: Necesitamos que los hilos esperen cuando no hay espacio disponible

Ventajas de la implementación:

1. **Thread-safe**: El uso de synchronized garantiza acceso exclusivo
2. **Eficiencia**: Los hilos no consumen CPU mientras esperan
3. **Justicia**: Los hilos se despiertan en orden FIFO (First In, First Out)
4. **Simplicidad**: El código es fácil de entender y mantener

Flujo de ejecución:

1. Un miembro intenta estacionar
2. Si hay espacio disponible → estaciona inmediatamente
3. Si no hay espacio → se suspende con wait()
4. Cuando otro miembro sale → llama notify() para despertar a un hilo en espera
5. El hilo despierto verifica nuevamente la condición y procede si hay espacio

Este patrón es ideal para este escenario porque proporciona una solución elegante para la sincronización de recursos compartidos limitados, evitando tanto el polling ineficiente como las condiciones de carrera.

Análisis del Patrón Read-Write Lock

Este programa implementa el **patrón de concurrencia "Bloqueo de Lectura-Escritura" (Read-Write Lock)**, que es una técnica de sincronización que permite múltiples lectores simultáneos, pero solo un escritor a la vez.

¿Cómo funciona el patrón?

El patrón Read-Write Lock se basa en estos principios:

1. **Múltiples lectores pueden acceder simultáneamente** - Cuando un thread obtiene un read lock, otros threads pueden obtener read locks también
2. **Solo un escritor a la vez** - Cuando un thread obtiene un write lock, ningún otro thread puede obtener ningún tipo de lock
3. **Los escritores tienen prioridad** - Si hay threads esperando para escribir, los nuevos lectores deben esperar

Implementación en el código

1. Clase ReadWriteLock (líneas 1-86)

Esta es la implementación central del patrón:

ReadWriteLock.java

```
private Object lockObj;

private int totalReadLocksGiven;

private boolean writeLockIssued;

private int threadsWaitingForWriteLock;
```

- totalReadLocksGiven: Contador de locks de lectura activos
- writeLockIssued: Indica si hay un lock de escritura activo
- threadsWaitingForWriteLock: Contador de threads esperando para escribir

2. Método getReadLock() (líneas 25-38)

ReadWriteLock.java

```
public void getReadLock() {

synchronized (lockObj) {

while ((writeLockIssued) ||

(threadsWaitingForWriteLock != 0)) {

try {

lockObj.wait();

} catch (InterruptedException e) {

//

}

}

//System.out.println(" Read Lock Issued");

totalReadLocksGiven++;

}

}
```

Punto clave: Un read lock se puede obtener solo si:

- No hay write lock activo (!writeLockIssued)

- No hay threads esperando para escribir (threadsWaitingForWriteLock == 0)

3. Método getWriteLock() (líneas 45-62)

ReadWriteLock.java

```
public void getWriteLock() {
synchronized (lockObj) {
threadsWaitingForWriteLock++;
while ((totalReadLocksGiven != 0) ||
(writeLockIssued)) {
try {
lockObj.wait();
} catch (InterruptedException e) {
//
}
}
//System.out.println(" Write Lock Issued");
threadsWaitingForWriteLock--;
writeLockIssued = true;
}
}
```

Punto clave: Un write lock se puede obtener solo si:

- No hay read locks activos (totalReadLocksGiven == 0)
- No hay write lock activo (!writeLockIssued)

Aplicación en el contexto del programa

4. Clase Item (líneas 1-38)

Item.java

```
public void checkOut(String member) {
rwLock.getWriteLock();
status = "Y";
System.out.println(member +
" has been issued a write lock-ChkOut");
rwLock.done();
}
```

Item.java

```
public String getStatus(String member) {
```

```
rwLock.getReadLock();
System.out.println(member +
" has been issued a read lock");
rwLock.done();
return status;
}
```

Puntos clave de aplicación:

1. **Operaciones de lectura** (getStatus): Usan getReadLock() porque múltiples miembros pueden consultar el estado simultáneamente
2. **Operaciones de escritura** (checkOut, checkIn): Usan getWriteLock() porque solo un miembro puede cambiar el estado a la vez

5. Clase MemberTransaction (líneas 1-32)

MemberTransaction.java

```
public void run() {
//all members first read the status
item.getStatus(name);
if (operation.equals("CheckOut")) {
System.out.println("\n" + name +
" is ready to checkout the item.");
item.checkOut(name);
try {
sleep(1);
} catch (InterruptedException e) {
//
}
item.checkIn(name);
}
}
```

Escenario de ejecución

En RWTest.java, se crean 6 transacciones:

- 3 miembros hacen "StatusCheck" (lectura)
- 3 miembros hacen "CheckOut" (escritura)

Comportamiento esperado:

1. Los 3 "StatusCheck" pueden ejecutarse simultáneamente (múltiples read locks)
2. Los "CheckOut" deben esperar a que no haya read locks activos

3. Solo un "CheckOut" puede ejecutarse a la vez (exclusividad de write lock)

Ventajas del patrón

1. **Mayor concurrencia**: Múltiples lectores pueden acceder simultáneamente
2. **Integridad de datos**: Los escritores tienen acceso exclusivo
3. **Prevención de starvation**: Los escritores tienen prioridad sobre nuevos lectores

Este patrón es especialmente útil en sistemas donde las operaciones de lectura son más frecuentes que las de escritura, como en este caso de gestión de biblioteca donde consultar el estado es más común que hacer checkout/checkin.

Análisis del Patrón "Orden de Bloqueo - Cierre Consistente"

Este proyecto demuestra la aplicación del patrón de concurrencia **"Orden de Bloqueo - Cierre Consistente"** (Lock Ordering - Consistent Locking), que es una solución al problema clásico del **deadlock** en sistemas concurrentes.

Problema que Resuelve:

El código muestra dos versiones:

- **Pre/**: Versión problemática que puede causar deadlock
- **Post/**: Versión corregida que implementa el patrón

Puntos Críticos donde se Aplica el Patrón:

1. Problema en la Versión Pre/ (líneas 4-8 en FileSysUtil.java):

java

Apply to ClientTest.j...

```
public void moveContents(Directory src, Directory dest) {
 synchronized (src) {
  synchronized (dest) {
   System.out.println("Contents Moved Successfully");
  }
 }
}
```

Problema: Siempre adquiere los locks en el mismo orden (src primero, luego dest), pero cuando dos threads ejecutan:

- Thread 1: moveContents(dir_1, dir_2) → lock dir_1, espera dir_2
- Thread 2: moveContents(dir_2, dir_1) → lock dir_2, espera dir_1

Resultado: **DEADLOCK** - ambos threads esperan indefinidamente.

2. Solución en la Versión Post/ (líneas 3-17 en FileSysUtil_Rev.java):

java

Apply to ClientTest.j...

```
public void moveContents(Directory src, Directory dest) {
 if (src.hashCode() > dest.hashCode()) {
  synchronized (src) {
   synchronized (dest) {
    System.out.println("Contents Moved Successfully");
   }
  }
 } else {
  synchronized (dest) {
   synchronized (src) {
    System.out.println("Contents Moved Successfully");
   }
  }
 }
}
```

Solución: **Orden de Bloqueo Consistente** basado en hashCode():

- Si src.hashCode() > dest.hashCode(): lock src primero, luego dest
- Si src.hashCode() <= dest.hashCode(): lock dest primero, luego src

Cómo Funciona el Patrón:

1. **Ordenamiento Determinístico**: Usa hashCode() para establecer un orden consistente entre objetos
2. **Prevención de Deadlock**: Garantiza que todos los threads adquieran locks en el mismo orden global
3. **Cierre Consistente**: Los locks se liberan en orden inverso (LIFO - Last In, First Out)

Escenario de Prueba (ClientTest.java):

java

Apply to ClientTest.j...

```
Directory dir_1 = new Directory("Directorio 1");
Directory dir_2 = new Directory("Directorio 2");
// Thread 1: moveContents(dir_1, dir_2)
// Thread 2: moveContents(dir_2, dir_1)
```

Resultado con el patrón aplicado: Ambos threads adquieren locks en el mismo orden basado en hashCode(), evitando el deadlock.

Ventajas del Patrón:

1. **Elimina Deadlocks**: Garantiza orden consistente de adquisición de locks
2. **Simplicidad**: Fácil de implementar y entender

3. **Eficiencia**: No requiere timeout o detección de deadlock
4. **Escalabilidad**: Funciona con cualquier número de threads

Este patrón es fundamental en sistemas concurrentes donde múltiples recursos deben ser protegidos simultáneamente.