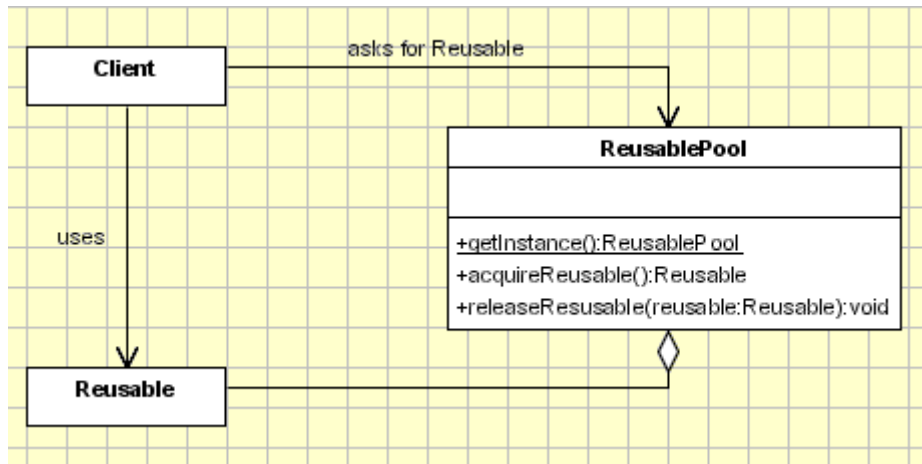


Object Pool

(*Resource Pool*)

Diagrama de clases e interfaces



Intención

El patrón Object Pool, también conocido como Pool de recursos, describe cómo se puede mejorar el rendimiento general de un sistema que debe tratar con computacionalmente costosas operaciones de adquisición y liberación de recursos. Para ello se basa en el reciclaje y cacheo de estos recursos.

Motivación

Consideremos el caso de una aplicación web que proporciona un catálogo que permite a los usuarios seleccionar y ordenar elementos del mismo.

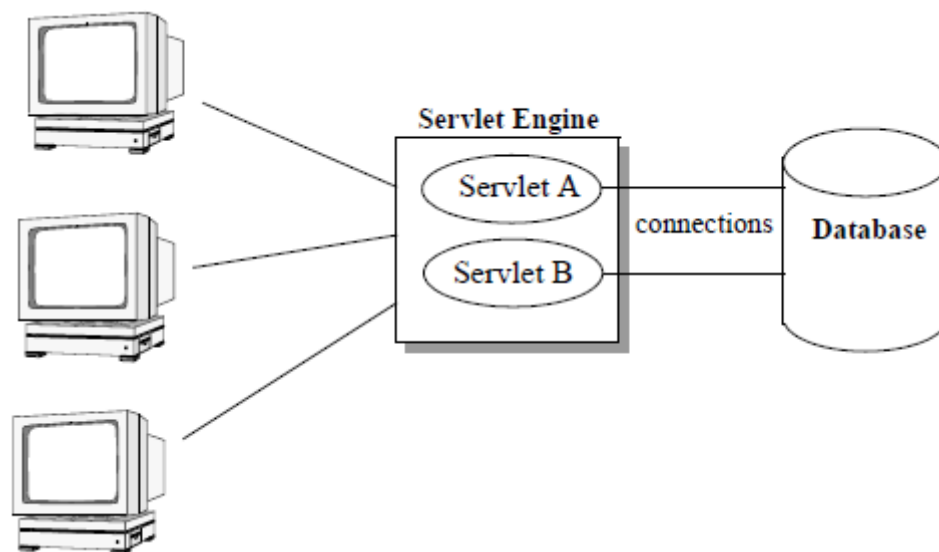
Supongamos que se ha utilizando una arquitectura de tres capas (presentación, negocio y datos). Los clientes son los navegadores web que se comunican con un contenedor de servlets, por ejemplo, Tomcat.

La **capa de presentación** (capa web) se lleva a cabo por uno o más servlets y páginas dinámicas (JSP) y estáticas.

Respecto a la **capa de negocio**, dado que la aplicación es muy sencilla y apenas hay lógica de negocio, se ha estimado oportuno que los propios servlets la implementen.

En cuanto a la **capa de datos**, son los mismos servlets los encargados de conectarse al RDBMS mediante JDBC.

La siguiente figura muestra una configuración de estas características, con dos servlets ejecutándose en el contenedor de servlets y estableciendo conexión a una base de datos.



La mayoría de las páginas web del catálogo que se generan de forma dinámica, dependiendo de los contenidos de la base de datos. Por ejemplo, para obtener una lista de elementos disponibles en el catálogo, junto con sus precios, uno de los servlets se conecta a la base de datos y ejecuta una consulta. El servlet utiliza los resultados de la consulta para generar la página HTML que se muestra al usuario. Del mismo modo, si un usuario realiza una compra y entra en los detalles de pago a través de un formulario HTML, uno de los servlets se conecta a la base de datos y la actualiza.

Según la arquitectura planteada, un servlet necesita obtener una conexión a la base de datos para poder ejecutar sentencias SQL. Una implementación trivial sería crear una conexión con la base de datos en cada petición realizada al servlet. Sin embargo, esta solución puede ser muy ineficiente, ya la creación de una conexión por cada petición del usuario conlleva un tiempo nada despreciable. Por otro lado, esta solución es costosa computacionalmente, ya que potencialmente se tendrían que crear miles

de conexiones en un periodo muy breve de tiempo (esto dependería del número de usuarios y los horarios de uso).

Una optimización de la solución anterior consistiría en almacenar una conexión en la sesión del usuario. Esto permitiría reutilizar conexiones por sesión, es decir, emplear la misma conexión para todas las peticiones de un mismo usuario. De todos modos, dado que un catálogo es un tipo de aplicación que potencialmente puede ser visitada simultáneamente por miles de usuarios, incluso con esta optimización se necesitaría un gran número de conexiones para cumplir con los requisitos funcionales (transacciones por segundo, tiempo de respuesta máximo para visualizar resultados, etc).

Contexto

Sistemas que tienen que ser altamente escalables, eficientes en los tiempos de respuesta y que continuamente deben adquirir y liberar recursos de un mismo tipo (o muy similares).

Problema

Muchos sistemas requieren un acceso rápido y fiable a los recursos, como conexiones de red, threads y memoria. Además, los sistemas también requieren que la solución escale correctamente al número de recursos utilizados, así como al número de usuarios de los recursos. Además, las diferentes peticiones que realice un usuario deben experimentar muy poca variación en su tiempo de acceso. Por lo tanto, el tiempo de adquisición del recurso r_0 no debe variar significativamente del tiempo de adquisición del recurso r_1 , donde r_0 y r_1 son recursos del mismo tipo.

Para solucionar los problemas mencionados es necesario resolver las siguientes características:

- Rendimiento (Performance): Se debe evitar desperdiciar ciclos de CPU en operaciones repetitivas de adquisición y liberación de recursos.
- Previsibilidad (Predictability): El tiempo que un usuario tarda en acceder a un recurso debe ser predecible.
- Simplicidad (Simplicity): La solución debe ser simple para minimizar la complejidad de la aplicación.

- Estabilidad (Stability): La repetitiva adquisición y liberación de recursos puede aumentar el riesgo de inestabilidad del sistema. Por ejemplo, la adquisición y liberación repetitiva de memoria puede conducir a la fragmentación de la misma. La solución debe minimizar la inestabilidad del sistema.
- Reutilización (Reuse): Los recursos no utilizados se deben reutilizar para evitar la sobrecarga que supone volver a obtenerlos.

Los patrones Adquisición Anticipada (Eager Acquisition) y Adquisición Perezosa o Bajo Demanda (Lazy Acquisition) resuelven sólo un subconjunto de las características anteriores. Por ejemplo, la Adquisición Anticipada, que cachea recursos cuando aún no se han solicitado, garantiza la obtención de recursos, ya que éstos se encuentran en la caché. En cambio, la Adquisición Perezosa se centra más en evitar la adquisición innecesaria de los recursos, ya que no intenta recuperarlos hasta que se le solicita expresamente. La Adquisición Anticipada es útil cuando conseguir un recurso tarda bastante tiempo. La Adquisición Perezosa es una opción cuando conseguir un recurso no conlleva un retardo pero se trata de un recurso escaso que no conviene tener bloqueado o reservado.

En resumen, ¿cómo se puede garantizar la adquisición de recursos sin comprometer el rendimiento ni la estabilidad del sistema?

Solución

La solución pasa por gestionar múltiples instancias de un tipo de recurso en un pool (conjunto, agrupación). Un pool permite la reutilización de sus recursos liberados, entendiendo por liberado un recurso que ha sido devuelto al pool después de su uso, cuando ya no es necesario.

Una vez creado el propio pool de recursos, para aumentar su eficiencia, se puede utilizar la técnica de la Adquisición Anticipada para obtener un número determinado de recursos y cachearlos. Si en algún momento la demanda excede los recursos disponibles en el pool, entonces se utilizará la técnica de la Adquisición Perezosa para obtener más recursos. Para liberar los recursos no utilizados existen varias alternativas, como el patrón Desalojar (Evict o Evictor) o el patrón Alquiler (Leasing).

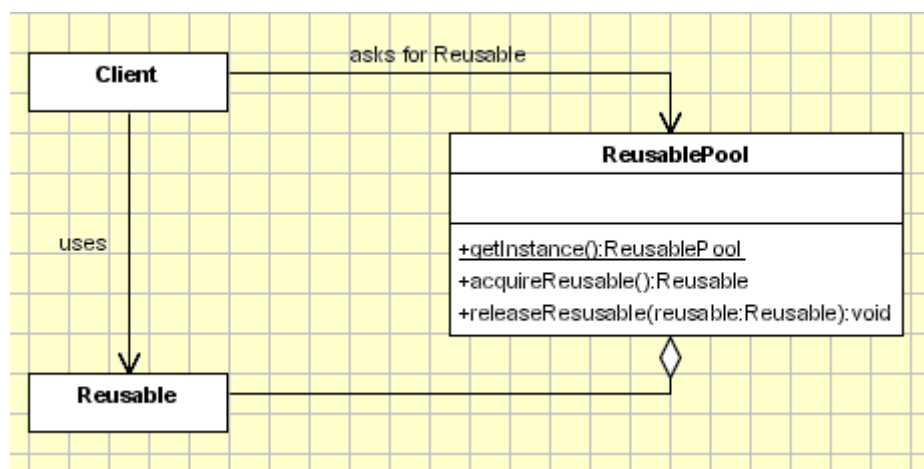
La liberación de un recurso y su devolución al pool tiene que realizarse de manera anónima y dependiendo de la estrategia utilizada, la entrega del recurso puede ser una

responsabilidad del usuario del recurso o del propio pool. Antes de que el recurso sea reutilizado tiene que ser reiniciado.

Si el recurso es un objeto, puede ser útil proporcionar por separado una interfaz de inicialización. El identificador (ID) del objeto, normalmente un puntero o una referencia, no es utilizado por el usuario del recurso o por el pool para identificar recursos.

Implementación

El siguiente diagrama muestra los participantes en el patrón:



Descripción de los participantes:

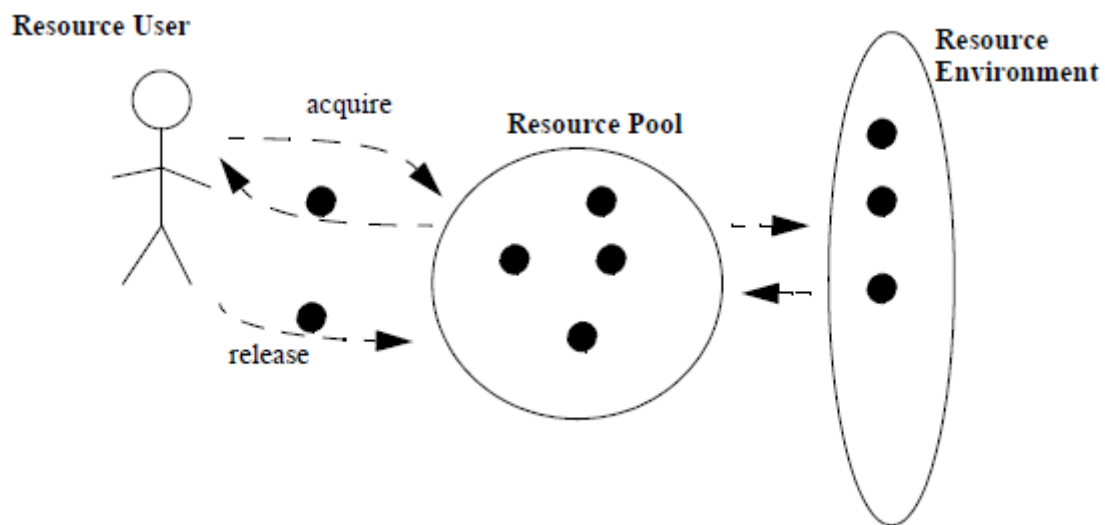
Aunque no aparece en el diagrama, habría que considerar a un participante especial: el **Entorno del recurso** (Resource Environment). Es el propietario del recurso. Por ejemplo, un sistema operativo es el propietario de los recursos del ordenador y se encarga de gestionarlos en primera instancia. Otro ejemplo, sería el caso de un RDBMS, que es el propietario de las conexiones que los programas establecen con la base de datos.

Cliente (Client): Usuario que realiza peticiones para conseguir y utilizar un recurso. También tiene la responsabilidad de devolver al pool un recurso una vez que ya no lo necesita.

Recurso reutilizable (Reusable): Es la entidad que necesita el Cliente, como la memoria, un thread o una conexión a una base de datos. Estas entidades (recursos) se obtienen del Entorno del recurso a través del pool.

Pool de Recursos (ReusablePool): Gestiona los recursos, entregándolos a los usuarios cuando recibe peticiones de adquisición y devolviéndolos al pool cuando los usuarios los liberan. Lo normal es que el pool, debido a la técnica de Adquisición Anticipada, ya tenga cacheados cierto número de recursos y las peticiones de adquisición de los usuarios puedan satisfacerse ipso facto. Cuando no hay recursos disponibles en el pool, mediante la técnica de Adquisición Perezosa se obtienen los recursos del Entorno del recurso.

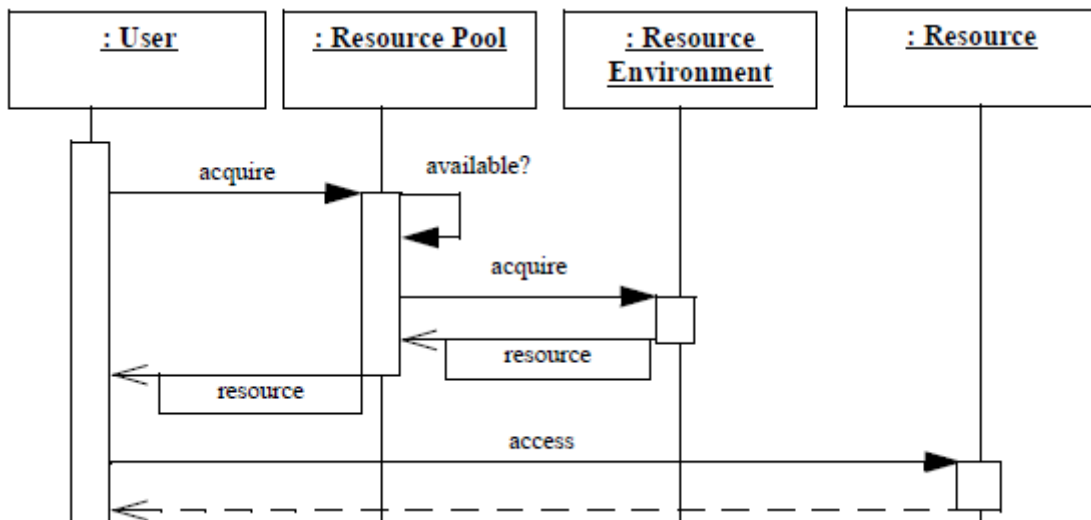
Las interacciones entre los participantes se muestran en la figura siguiente:



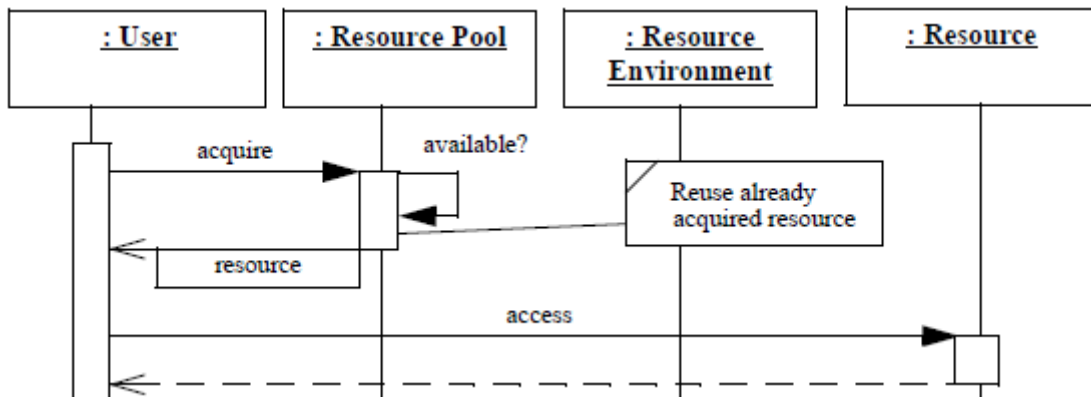
La interacción entre los participantes varía ligeramente dependiendo de si el pool, cuando se inicializa, utiliza o no la técnica de Adquisición Anticipada a fin de cachear recursos. Suponiendo que el pool consigue los recursos por adelantado, las peticiones de los usuarios se despachan con los recursos cacheados en el pool. Cuando los usuarios ya no los necesitan los entregan al pool, donde serán reciclados para poder ser reutilizados en nuevas peticiones de adquisición.

Una situación un poco más compleja se describe en el siguiente diagrama de secuencias, que se ha dividido en tres partes para facilitar su comprensión.

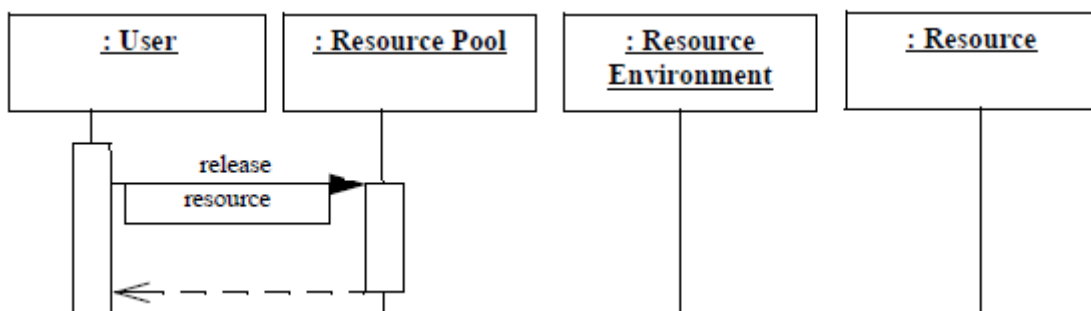
En esta primera sección del diagrama se produce una petición de adquisición por parte de un usuario pero debido a que el pool no dispone de ningún recurso cacheado, le obliga a solicitar bajo demanda (Adquisición Perezosa) un recurso al Entorno del recurso. Cuando el usuario obtiene el recurso, lo utiliza.



En la siguiente sección del diagrama, se muestra el caso en que sí hay recursos cacheados disponibles en el pool. Cuando el usuario obtiene el recurso, lo utiliza.



La tercera sección del diagrama ilustra la situación en que el usuario ya no necesita el recurso y lo entrega al pool. Obviamente, esto sucede siempre igual, independientemente de la manera en que se haya obtenido un recurso:



El pool utiliza datos estadísticos sobre la utilización de los recursos para determinar de la manera óptima de mantener y desalojar (evict) recursos de la caché. Estos datos

estadísticos contemplan características de uso, tales como: la hora exacta del último uso, la frecuencia de uso, etc.

Detalle de los pasos necesarios para la implementación del patrón Object Pool

1º) Seleccionar los recursos

Identificar los recursos que queremos que sean gestionados por el pool. En caso de tener objetos de diferente tipo, los agrupamos por el tipo para simplificar su gestión. No obstante, tener grupos de recursos de diferente tipo siempre complica la administración del pool, ya que requiere crear sub-pools y realizar búsquedas ad-hoc.

2º) Determinar el tamaño máximo del pool

El pool no puede almacenar un número infinito de recursos, por lo que se necesita establecer el número máximo de recursos que se almacenarán en su caché, llamemos a este número MAX_RESOURCES. La suma del número de recursos adquiridos mediante la técnica de Adquisición Anticipada (digamos MAX_EAGER) más el número de recursos adquiridos mediante Adquisición Perezosa (digamos MAX_LAZY) nunca puede ser mayor que MAX_RESOURCES. Visualmente: $MAX_EAGER + MAX_LAZY \leq MAX_RESOURCES$

El número máximo de recursos (MAX_RESOURCES) normalmente se establece en la inicialización del pool, aunque también es posible cambiar este valor una vez inicializado el pool, esto es, ya en tiempo de ejecución, en función de la carga del sistema y/o del número de nodos hardware disponibles en un cluster, por ejemplo.

3º) Determinar el número de recursos que se pueden obtener mediante Adquisición Anticipada

Para minimizar el tiempo de adquisición de recursos en tiempo de ejecución, esto es, evitar que el pool deba acceder al Entorno del recurso, se recomienda adquirir anticipadamente (cachear) al menos la mitad de los recursos que se utilizan en la aplicación en circunstancias normales. Esto reduce las operaciones de adquisición perezosa en tiempo de ejecución. Las necesidades de los usuarios junto con un análisis del sistema, ayuda a determinar el número promedio de recursos que habría que cachear anticipadamente. Hay que tener en cuenta que adquirir demasiados

recursos anticipadamente puede ser contraproducente, ya que implica reservar recursos algunos de los cuales puede que nunca se utilicen, por lo se debería evitar.

4º) Definir una interfaz de recursos

Se debe crear una interfaz que todos los recursos del pool tienen que implementar. Esta interfaz proporciona una clase base común para todos los recursos y facilita gestionarlos en una colección. Por ejemplo, una posible interfaz sería la siguiente:

```
public interface Resource {  
  
}
```

Una clase que implemente la interfaz Resource ha de mantener la información de contexto necesaria para determinar qué recursos deben desalojarse del pool y cuándo es necesario expulsarlos. Esta información incluye marcas de tiempo (timestamp) y contadores de uso de cada recurso. Por ejemplo, a continuación se muestra una clase llamada Connection que implementa la interfaz Resource:

```
public class Connection implements Resource {  
    // Información de contexto para desalojar recursos del pool  
    private Date lastUsage_;  
    private boolean currentlyUsed_;  
  
    public Connection() {  
        // ....  
    }  
    // ....  
}
```

Para casos en los que se necesita integrar código heredado (legacy code) y no sea posible hacer que el recurso implemente la interfaz Resource, se puede utilizar el patrón Adaptador (patrón estructural de GoF, que estudiaremos en detalle a su debido tiempo). La clase Adaptador puede implementar la interfaz Resource y envolver al recurso real (el código heredado). La información de contexto puede ser responsabilidad de la clase Adaptador. El siguiente código muestra como sería una clase Adaptador para la clase de Connection:

```
public class ConnectionAdapter implements Resource {  
  
    // Información de contexto para desalojar recursos del pool  
    private Date lastUsage_;  
    private boolean currentlyUsed_;  
  
    // Recurso  
    private Connection connection_;
```

```

// Constructor (envuelve al recurso)
public ConnectionAdapter(Connection connection) {
    connection_ = connection;
}

public Connection getConnection() {
    return connection_;
}
}

```

5º) Definir una interfaz para el pool y clases que la implementen (los recursos)

Hay que proporcionar una interfaz que permita a los usuarios de los recursos adquirirlos y liberarlos. Por ejemplo:

```

public interface Pool {
    Resource acquire();
    void release(Resource resource);
}

```

Una clase que implemente la interfaz Pool ha de mantener una colección de objetos Resource a modo de caché. De esta manera, cuando un usuario intente adquirir un recurso, éste se obtendrá de la colección. Del mismo modo, cuando el usuario entregue el recurso, éste será añadido de nuevo a la colección. Un ejemplo del tal clase podría ser el siguiente:

```

public class ConnectionPool implements Pool {

    // Colección de objetos Connection
    private java.util.Vector connectionPool_;

    /*
     * Metodos publicos
     */
    public Resource acquire() {
        Connection connection = findFreeConnection();
        if (connection == null) {
            // En este caso acudir al Entorno del recurso
            connection = new Connection();
            connectionPool_.add(connection);
        }
        return connection;
    }

    public void release (Resource resource) {
        if (resource instanceof Connection) {
            recycleOrEvictConnection((Connection) resource);
        }
    }

    /*
     * Metodos privados

```

```

    */
    private Connection findFreeConnection () {
        // ...
    }

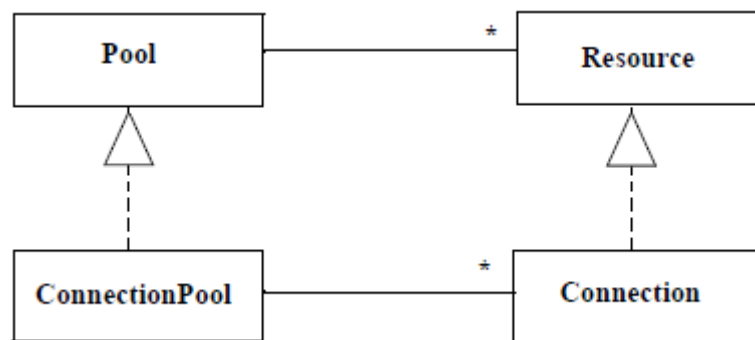
    private recycleOrEvictConnection (Connection connection) {
        // ...
    }

    // ...
}

```

El código anterior muestra una manera de llevar a cabo la creación, inicialización y eliminación de conexiones. Por supuesto, es posible hacer que estas operaciones sean muy flexibles mediante el uso de los patrones Estrategia (Strategy) y Factoría (Factory).

El siguiente diagrama de clases muestra la estructura y las colaboraciones de las clases e interfaces antes mencionadas.



6º) Desalojar recursos del pool

Muchos recursos en el pool es indicativo de una mala estrategia en su configuración. Hay que tener presente que un número importante de recursos cargará significativamente el sistema y degradará su rendimiento. Los recursos ocupan espacio y consumen ciclos de CPU, ya que a su vez requieren de otros recursos de más bajo nivel. Y todo esto sin reportar ningún beneficio.

Para contrarrestar estos inconvenientes hay que reducir el número de recursos del pool a un tamaño razonable. Una buena estrategia consiste en desalojar aquellos recursos que se utilicen con poca frecuencia. El patrón Desalojar (Evictor) se puede utilizar para este propósito, ya que permite configurar diferentes estrategias para determinar los recursos a desalojar y con qué frecuencia desalojarlos.

7º) Determinar cómo se reciclan los recursos

El reciclaje de los recursos varía en función del tipo del recurso. Por ejemplo, reciclar un thread requiere limpiar su pila e inicializar la memoria. En el caso de los objetos, se tiene que comprobar si el objeto está compuesto por otros objetos más pequeños y entonces reciclarlos uno a uno.

8º) Determinar estrategias ante fallos

Se debe gestionar adecuadamente cualquier fallo recuperable en operaciones de adquisición o liberación de recursos. Si no es posible recuperarse de un fallo, se debe lanzar la correspondiente excepción al usuario del recurso.

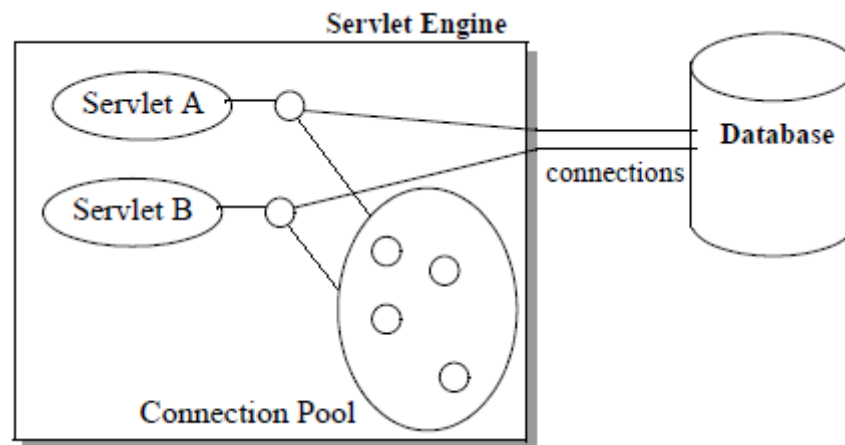
Aplicabilidad y Ejemplos

Caso 1: Aplicación web de catálogo de productos

Las conexiones de base de datos se agrupan en un pool de conexiones. En cada petición del usuario, un servlet obtiene una conexión del pool, la cual es utilizada para acceder a la base de datos y realizar operaciones de consulta y de actualización. Una vez ejecutadas estas operaciones, se entrega la conexión al pool.

Hay que tener en mente que el número de peticiones simultáneas siempre es inferior al número de usuarios de la aplicación, por lo que se necesitan pocas conexiones para proporcionar un buen rendimiento. Por lo general, con un número pequeño de conexiones cacheadas en el pool (obtenidas anticipadamente) es posible satisfacer todas o la mayoría de las peticiones de obtención de conexión. Cuando esto no sea posible el pool deberá acudir al Entorno del recurso a por nuevas conexiones. En cierta manera, es un tema de ir afinando la configuración del pool en función de los hábitos de uso de la aplicación.

La figura siguiente muestra cómo los servlets adquieren conexiones del pool para poder conectar a la base de datos.



Hay que advertir que en el momento en que el pool no tiene ninguna conexión disponible y no se ha alcanzado el número máximo de conexiones, se crean nuevas conexiones bajo demanda.

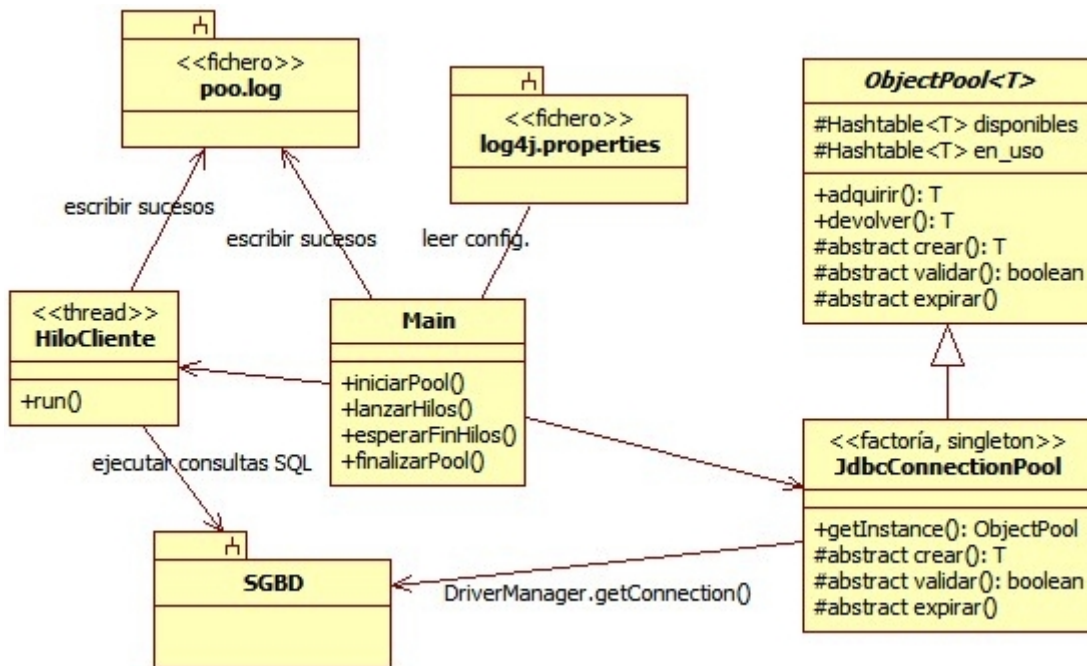
Caso 2: Aplicación de escritorio que lanza varios hilos concurrentemente para simular diferentes usuarios realizando consultas SQL

En este ejemplo sí que se adjunta el código fuente de la aplicación.

Vamos a diseñar un pool genérico que podríamos utilizar como código base trabajar con diferentes tipos de recursos. No obstante, en este ejemplo, sólo tendremos una subclase para el pool, especializada en recursos `java.sql.Connection`.

El punto de entrada a la aplicación será una clase que se encargará de crear el pool y de lanzar un número determinado de usuarios simultáneamente. Para representar a un usuario se utiliza una clase basada en la clase `Thread`, por lo que tendremos diferentes hilos de ejecución sobre esta clase. Además existen dos clases de soporte cuyo código es secundario y no conviene mezclar en las clases más relevantes.

El diagrama siguiente muestra la estructura de la aplicación:



Librerías necesarias:

- mysql-connector-java-5.1.12-bin.jar
- log4j-1.2.16.jar

No tienen que ser exactamente estas versiones.

Comenzamos por la clase DatosConexion, que como su nombre deja ver, se trata de una clase que encapsula los datos relativos a la conexión con la base de datos:

DatosConexion.java

```

package creacionales.objectpool;

/*
 * Clase que encapsula la información necesaria para
 * conectar con una base de datos.
 * Se utiliza para no tener que pasar tantos
 * parámetros en las llamadas a métodos
 */
public class DatosConexion {
    String driver, dsn, usr, pwd;
    public DatosConexion(String driver_, String dsn_, String usr_,
String pwd_) {
        driver = driver_;
    }
}
  
```

```

        dsn = dsn_;
        usr = usr_;
        pwd = pwd_;
    }
}

```

Veamos ahora la jerarquía de clases que conforman el pool. El código está ampliamente comentado, por lo que para entender los detalles se recomienda leerlos con atención.

La clase base, `ObjectPool`, es un pool de conexiones abstracto que puede gestionar cualquier tipo de recurso. Para ello está implementada usando los Generics aparecidos en Java 5. Las clases de implementación deberán proporcionar el tipo `T` con el que trabajará `ObjectPool`. `T` es el tipo del recurso a cachear, que en nuestro ejemplo se trata de tipo `java.sql.Connection`.

`ObjectPool` incluye el código fundamental para la gestión del pool, válido para cualquier subclase. Utiliza dos tablas hash, una para almacenar objetos en uso y otra para almacenar objetos disponibles.

- Funcionamiento de la tabla de objetos disponibles: Cuando un usuario solicita un objeto y todavía queda alguno en la tabla de disponibles, se quita de la tabla de disponibles, se pone en la tabla de objetos en uso y se le entrega al usuario para que lo utilice.
- Funcionamiento de la tabla de objetos en uso: Cuando el usuario ya no utiliza más un objeto, lo devuelve al pool, esto es, se quita de la tabla de objetos en uso y se añade a la de objetos disponibles.

Para desalojar recursos del pool utiliza una estrategia basada en el patrón Evictor, esto es, se lleva un control del tiempo que los recursos permanecen sin utilizarse en la tabla de objetos disponibles. Pasado un tiempo el objeto es invalidado y devuelto al Entorno del recurso, es decir, al SGBD. El tiempo es configurable y viene determinado por un parámetro en el constructor.

`ObjectPool` desconoce los detalles de implementación de los objetos que gestiona, ya que eso es responsabilidad de las subclases. Consecuentemente, las siguientes operaciones las define como abstractas y son las subclases las que deben proporcionar una implementación.

- Creación de un recurso: Adquisición desde el Entorno del recurso.

- Verificar vigencia: Esto se hace siguiendo una estrategia de desalojo del pool.
- Caducar objeto: Marcarlo como obsoleto y devolverlo al Entorno del recurso.

ObjectPool no puede implementar directamente un patrón de adquisición anticipada de recursos, ya que esto implicaría conocer detalles de implementación que corresponden a las subclases. Por tanto, son las subclases las que se encargan de permitir que el pool pueda cachear recursos de modo anticipado, como veremos en breve.

ObjectPool proporciona dos métodos públicos permiten, respectivamente, adquirir recursos y entregarlos después de su uso:

- adquirir(): Si hay alguno objeto en la cache (tabla de disponibles) y aún es válido (vigente) entonces retirarlo de la tabla y entregarlo al usuario para su uso. En caso contrario adquirirlo desde el Entorno del recurso y ponerlo en la tabla de recursos en uso.
- devolver(): Quitarlo de la tabla de recursos en uso y ponerlo en la de disponibles indicando el instante en que este último hecho se produce.

Por último, hay otra serie de métodos más básicos: operaciones que devuelven información sobre el estado del pool, operaciones que limpian completamente el pool antes de finalizar el programa del cliente, etc.

El código para el pool genérico queda como sigue:

ObjectPool.java

```
package creacionales.objectpool;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Map.Entry;

/**
 * Pool de conexiones abstracto.
 */

// T = java.sql.Connection (en nuestro ejemplo)
public abstract class ObjectPool<T> {

    /**
     * Creamos dos tablas Hash, una para almacenar objetos
     * bloqueados y otra para objetos disponibles.
     * -La clave de la tabla es el objeto T, esto es, el recurso,
```



```

    * que en este caso es una Conexion.
    * -El valor es un Long que representa el instante temporal
    * en el que el recurso se ha añadido a la tabla.
    */
    protected Hashtable<T, Long> en_uso, disponibles;

    // Tiempo máximo que una objeto puede permanecer en el pool
    private long expirationTime;

    /**
     * Constructor
     */
    public ObjectPool(int expirationTime_) {
        expirationTime = expirationTime_;

        // Inicializar las tablas
        en_uso = new Hashtable<T, Long>();
        disponibles = new Hashtable<T, Long>();
    }

    /*
     * Métodos abstractos que deben implementar las subclases
     */

    // Obtiene un objeto desde el Entorno del recurso.
    // Carga bajo demanda (adquisicion perezosa)
    protected abstract T crear();

    protected abstract boolean validar(T o);

    protected abstract void expirar(T o);

    /*
     * Metodo sincronizado que permite al usuario adquirir una
    conexion.
     */
    public synchronized T adquirir() {
        T t; // 't' es una java.sql.Connection

        // Si hay objetos disponibles
        if (!disponibles.isEmpty()) {
            // Obtener las claves de la tabla de objetos
            disponibles
            Enumeration<T> e = disponibles.keys();
            // Recorrer la tabla de objetos disponibles
            while (e.hasMoreElements()) {
                t = e.nextElement();
                // Si el objeto actual ha superado el tiempo
                // maximo de
                // permanencia en la tabla, hay que desalojarlo
                // de ella

                if (haExpirado(t)) {
                    // Quitarlo de la tabla de disponibles
                    disponibles.remove(t);
                }
            }
            /*
             * Liberar el recurso --> cerrar la
            Conexion.
             * Hay que advertir que la implementacion
            del metodo

```

```

        * expire() está en alguna subclase de
esta clase abstracta.
        */
        expirar(t);

        // Lista para el recolector de basura
        t = null;

    } else { // Si el objeto actual tiene una
vigencia adecuada

        /*
        * Hay que advertir que la implementacion
del metodo
        * validate() está en alguna subclase de
esta clase abstracta.
        */

        // Si la conexión aun está abierta
        if (validar(t)) {
            // Quitarlo de la tabla de
disponibles

            disponibles.remove(t);

            // Ponerlo en la tabla de
bloqueados

            en_uso.put(t, ahora());

            // Devolverlo al usuario
            return (t);

        } else { // La conexión está cerrada
            // Quitarlo de la tabla de
disponibles

            disponibles.remove(t);

            /*
            * Liberar el recurso --> cerrar la
Conexion.
            * Hay que advertir que la
implementacion del metodo
            * expire() está en alguna subclase
de esta clase abstracta.
            */
            expirar(t);

            // Lista para el recolector de
basura

            t = null;

        }
    }
}

/*
* No hay objetos en la tabla de disponibles, se debe crear
uno nuevo.
* Hay que advertir que la implementacion del metodo

```

```

        * create() está en alguna subclase de esta clase
abstracta.
        * El método accederá al Entorno del recurso para
obtenerlo.
        */
        t = crear();

        // Añadirlo en la tabla de bloqueados
        en_uso.put(t, ahora());

        // Devolverlo al usuario
        return (t);
    }

    /*
    * Metodo sincronizado que permite al usuario devolver una
conexión al pool.
    */
    public synchronized void devolver(T t) {
        // Quitarlo de la tabla de 'en uso'
        en_uso.remove(t);

        // Ponerlo en la tabla de disponibles
        disponibles.put(t, ahora());
    }

    /*
    * Los tres siguientes métodos permiten conocer
    * el estado del pool
    */
    public int getDisponiblesEnPool() {
        return disponibles.size();
    }

    public int getOcupadosEnPool() {
        return en_uso.size();
    }

    public int getOcupacionTotal() {
        return getDisponiblesEnPool() + getOcupadosEnPool();
    }

    /*
    * Metodo para liberar todos los recursos del pool.
    * La aplicación cliente lo puede invocar antes de terminar.
    */
    public void liberarTodo() {
        for (Entry<T, Long> e: disponibles.entrySet()) {
            T t = e.getKey();
            expirar(t);
            t = null;
        }
        disponibles.clear();

        for (Entry<T, Long> e: en_uso.entrySet()) {
            T t = e.getKey();
            expirar(t);
            t = null;
        }
    }

```

```

        en_uso.clear();
    }

    /*
     * Métodos protegidos/privados
     */

    // Devolver la marca de tiempo actual
    protected long ahora() {
        return System.currentTimeMillis();
    }

    /*
     * Informar si el objeto recibido por parámetro ha
     * agotado el tiempo máximo de permanencia como
     * disponible en el pool. Esto es necesario para
     * mantener un tamaño adecuado de pool.
     */
    private boolean haExpirado(T t) {
        return (ahora() - disponibles.get(t)) > expirationTime;
    }
}

```

Veamos ahora la clase `JdbcConnectionPool`, la única subclase de `ObjectPool`. Esta clase es una factoría de objetos `Connection` del paquete `java.sql` y, como casi todas las factorías en entornos multihilo, está implementada como un `Singleton`. Concretamente se trata de la variante que utiliza una clase estática para la obtención de la instancia `JdbcConnectionPool`. Por la manera en que se cargan las clases en Java, utilizar una clase interna estática permite no utilizar la verbosa técnica del doble bloqueo para evitar el problema de que dos hilos tengan la posibilidad de crear sendas instancias del `Singleton`.

Recordemos el funcionamiento de esta variante `Singleton`: La clase cliente invoca al método `getInstance()` del `Singleton` para obtener la instancia, el pool en nuestro caso. Este método hace referencia a un atributo estático de la clase estática interna al `Singleton`, que en nuestro caso se denomina `SingletonHolder`. La declaración y la asignación del atributo estático se hacen en la misma sentencia, invocando al constructor privado de la clase externa. La idea es similar a la siguiente:

```

public class JdbcConectionPool {
    private JdbcConectionPool() {
        ...
    }
    ...
    private static class SingletonHolder {
        // Atributo estático
    }
}

```

```

        private static ObjectPool<Connection> pool =
                                new JdbcConectionPool();
    }
    ...
    // El cliente llama a este método y obtiene el pool
    // Se devuelve el atributo estático de la clase estática
    public static ObjectPool<Connection> getInstance() {
        return SingletonHolder.pool;
    }
}

```

Sin embargo, en nuestro caso difiere sensiblemente porque tenemos que pasar tres parámetros para que se construya el pool. A saber:

- Los datos de conexión a la base de datos, encapsulados en una instancia de una clase auxiliar denominada DatosConexion.
- El número de conexiones que se deberán adquirir anticipadamente una vez que se inicie el pool.
- El tiempo límite para que una conexión en la caché deje de considerarse válida.

Por otro lado, como se comentó anteriormente en el análisis de la clase base, JdbcConnectionPool, en tanto que subclase de ObjectPool, tiene que implementar los métodos abstractos declarados en la superclase. Básicamente se trata de código específico para la creación, verificación y cierre del objeto java.sql.Connection.

A destacar el código que realiza el constructor privado para adquirir anticipadamente conexiones desde el Entorno del recurso y guardarlas en la caché para acelerar la entrega a los clientes que lo soliciten. Ya se dijo anteriormente, que esto es responsabilidad de las subclases de ObjectPool, pues se trata de detalles de implementación específicos al tipo del recurso.

El código queda como sigue:

JdbcConnectionPool.java

```

package creacionales.objectpool;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

```

```

/*
 * SINGLETON
 *
 * SubClase de ObjectPool especializada en objetos Connection de
 java.sql
 * Es un Singleton que utiliza la técnica de la clase estática para la
 * obtención de la instancia Singleton.
 */
public class JdbcConnectionPool extends ObjectPool<Connection> {

    // Límites de la caché
    private static final short MIN_CACHEADAS = 0, MAX_CACHEADAS =
20;

    // atributos para almacenar los diferentes valores de la
conexión
    private String dsn, usr, pwd;

    /*
     * Atributos estáticos necesarios para la clase interna
     */
    private static DatosConexion dc;
    private static short conexionesCacheadas;
    private static int expirationTime;

    /*
     * Constructor privado
     */
    private JdbcConnectionPool(DatosConexion dc, short
conexionesCacheadas, int expirationTime) {
        super(expirationTime);
        if (conexionesCacheadas < MIN_CACHEADAS ||
            conexionesCacheadas > MAX_CACHEADAS) {
            throw new IllegalArgumentException("El valor para la
conexiones que " +
                "se deben cachear no está en el intervalo
permitido");
        }

        try {
            Class.forName(dc.driver).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        this.dsn = dc.dsn;
        this.usr = dc.usr;
        this.pwd = dc.pwd;

        /*
         * Si el cliente ha indicado que el pool se debe
         * iniciar con conexiones cacheadas.
         * Hay que advertir que esta comprobación está a nivel
         * de subclase en lugar de estar en la superclase.
         * Esto es así porque la llamada a crear() es abstracta
         * en la subclase y en consecuencia se llamaría a crear()
         * de la subclase cuando aun no estaría construido.
         */
        if (conexionesCacheadas > MIN_CACHEADAS) {

```

```

        for (int i=1; i<=conexionesCacheadas; i++) {

            disponibles.put(crear(), ahora());

        }

    }

    /*
     * La clase SingletonHolder se carga en la primera ejecucion de
     * JdbcConnectionPool.getInstance(), pero no antes.
     */
    private static class SingletonHolder {
        private static ObjectPool<Connection> pool_ =
            new JdbcConnectionPool(dc, conexionesCacheadas,
expirationTime);
    }

    /*
     * Metodo estatico publico que retona la instancia. Notad que lo
que se
     * devuelve es el atributo estatico de la clase estatica
SingletonHolder.
     * Esto asegura que dos threads no pueden obtener sendas
instancias del Singleton.
     */
    public static ObjectPool<Connection> getInstance(DatosConexion
dc_,
            short conexionesCacheadas_, int expirationTime_)
    {
        // Asignamos los parámetros a los atributos estáticos de
JdbcConnectionPool
        // para que los tenga disponibles la clase SingletonHolder
        dc = dc_;
        conexionesCacheadas = conexionesCacheadas_;
        expirationTime = expirationTime_;
        return SingletonHolder.pool_;
    }

    /*
     * El pool obtiene una conexion del Entorno del recurso
     */
    @Override
    protected Connection crear() {
        try {
            return (DriverManager.getConnection(dsn, usr, pwd));
        } catch (SQLException e) {
            e.printStackTrace();
            return (null);
        }
    }

    @Override
    protected void expirar(Connection cnn) {
        try {
            cnn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```

@Override
protected boolean validar(Connection cnn) {
    try {
        return (!cnn.isClosed());
    } catch (SQLException e) {
        e.printStackTrace();
        return (false);
    }
}
}

```

Pasemos ahora con el código que hace uso del pool.

Por un lado, necesitamos crear un fichero de configuración para log4j, ya que la aplicación escribe todos los resultados en un archivo siguiendo el formato aquí especificado. El nombre y la ubicación para este archivo de resultados se determina en este fichero de configuración.

El fichero de configuración para log4j lo creamos en el paquete creacionales.objectpool.main y bajo el nombre log4j.properties. Su contenido es el siguiente:

log4j.properties

```

log.dir=c://
rrd.dir=${log.dir}/rrd
datestamp=yyyy-MM-dd/HH:mm:ss.SSS/zzz
roll.pattern.hourly=.yyyy-MM-dd.HH
roll.pattern.daily=.yyyy-MM-dd

# catchAll.log -- Default catch-all.
log4j.rootLogger=DEBUG, defaultLog
log4j.appender.defaultLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.defaultLog.DatePattern=${roll.pattern.daily}
log4j.appender.defaultLog.File=${log.dir}/pool.log
log4j.appender.defaultLog.layout=org.apache.log4j.PatternLayout
log4j.appender.defaultLog.layout.ConversionPattern=%d{%datestamp}

[%t] %-5p %m%n

```

Notad que según esta configuración se creará un fichero llamado pool.log en la raíz de la unidad C:/ (adaptarlo como sea necesario para otras plataformas).

Por otro lado, nos quedan ver las clases cliente: Main, HiloCliente y Utiles. Main dirige el programa, HiloCliente realiza el trabajo que simula las consultas SQL que haría cada usuario y Utiles contiene una serie de métodos estáticos que utilizan las otras dos clases del lado cliente.

La clase principal:

Main.java

```
package creacionales.objectpool.main;

import java.sql.Connection;
import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Logger;

import creacionales.objectpool.DatosConexion;
import creacionales.objectpool.JdbcConnectionPool;
import creacionales.objectpool.ObjectPool;

public class Main {

    // Nombre del fichero log4j
    private static final String LOG_PROPERTIES_FILE =
"log4j.properties";
    // ¿El fichero log4j está en un paquete de la aplicacion?
    private static final boolean LOG_FILE_IN_PACKAGE = true;

    // log para que la clase Main escriba los mensajes que se
necesite
    private static final Logger LOG = Logger.getLogger(Main.class);

    /*
    * POOL de conexiones
    * Visibilidad pública; es un Singleton. Todos los hilos deben
leer su valor
    */
    public static ObjectPool<Connection> pool;

    /*
    * Inicializados estático.
    * Leer la configuracion del log4j.
    */
    static {
        Utiles.loadConfigLogging(Main.class, LOG_PROPERTIES_FILE,
LOG_FILE_IN_PACKAGE);
        if (LOG.isDebugEnabled())
            LOG.debug("Fichero "+LOG_PROPERTIES_FILE+"
cargado.");
    }

    // Lista para almacenar los hilos
    private static List<HiloCliente> listaHilos;

    /* * * * * *
    * Punto de entrada al programa
    * * * * * */
    public static void main(String args[]) {
        // nanoTime() devuelve la milmillonesima parte de un
segundo
```

```

        long tiempoInicial = System.nanoTime();

        iniciarPool(); // crear el pool
        lanzarHilos(); // crear y arrancar los usuarios
        concurrentes
        esperarFinalizacionHilos(); // esperar a que acaben todos
        finalizarPool(); // cierre y limpieza del pool

        long duracion = System.nanoTime() - tiempoInicial;
        System.out.println("Tiempo de ejecucion: " + duracion + "
nanosegundos.");
        System.out.println("Programa finalizado. Consultar
resultados en el LOG.");
    }

    /*
     * Crear el pool de conexiones
     */
    private static void iniciarPool() {

        // El pool se creara con una cache que tendra
        // el siguiente numero de conexiones conexiones
        final short CONEXIONES_CACHEADAS = 3;

        // El pool eliminará recursos de la cache si no se
        // usan más de 30 segundos
        final short CACHE_EXPIRE = 30000;

        if (LOG.isDebugEnabled()) {

LOG.debug ("-----");
            LOG.debug ("INICIANDO APLICACION");

LOG.debug ("-----");
            LOG.debug ("Creando pool de conexiones con cache de "
+
CONEXIONES_CACHEADAS + " conexiones y
caducidad de "+
(CACHE_EXPIRE/1000) + " segundos.");
        }

        // Bean para encapsular la informacion de la conexion
        DatosConexion dc = new
DatosConexion("com.mysql.jdbc.Driver",
                "jdbc:mysql://localhost:3306/video",
                "root", "root");

        // Crear el pool de conexiones
        pool = JdbcConnectionPool.getInstance(dc,
CONEXIONES_CACHEADAS, CACHE_EXPIRE);
    }

    /*
     * Lanzar los hilos (usuarios) simultaneos
     */

```

```

private static void lanzarHilos() {
    // Numero maximo de hilos concurrentes?
    final int MAX_NUM_USERS = 10;

    if (LOG.isDebugEnabled()) {
        LOG.debug("Lanzando " + MAX_NUM_USERS + " hilos concurrentemente.");
    }

    String nombreHilo = null;
    listaHilos = new ArrayList<HiloCliente>();
    for (int i=1; i<MAX_NUM_USERS; i++) {
        nombreHilo = String.valueOf("Hilo " + i);
        HiloCliente hilo = new HiloCliente(nombreHilo);

        // Guardar cada objeto en la lista
        listaHilos.add(hilo);

        // Arrancar cada hilo
        if (LOG.isDebugEnabled()) {
            LOG.debug("Arrancado el hilo " + i);
        }
        hilo.start();
    }
}

/*
 * Esperar a que finalice cada usuario
 */
private static void esperarFinalizacionHilos() {

    for (HiloCliente hilo : listaHilos) {
        try {
            if (LOG.isDebugEnabled()) {
                LOG.debug(hilo.getName() + "
finalizado.");
            }
            hilo.join();
        } catch (InterruptedException e) {
            LOG.debug(e);
            e.printStackTrace();
        }
    }
}

/*
 * Solicitar al pool que cierre las conexiones cacheadas
 */
private static void finalizarPool() {
    if (pool != null) {
        pool.liberarTodo();
        if (LOG.isDebugEnabled()) {
            LOG.debug("Conexiones del pool cerradas.
Devueltas al Entorno del recurso (SGBD).");
            LOG.debug("Conexiones -- disponibles: " +
pool.getDisponiblesEnPool())

```

```

        + " Ocupadas: " +
pool.getOcupadosEnPool()
        + " Totales: " +
pool.getOcupacionTotal());
    }
}
}

```

La clase que extiende a Thread:

HiloCliente.java

```

package creacionales.objectpool.main;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import org.apache.log4j.Logger;

public class HiloCliente extends Thread {

    // Disponer un log para que cada hilo escriba los mensajes que
    considere oportunos
    static final Logger LOG = Logger.getLogger(HiloCliente.class);

    // Cada hilo ejecutará 10 consultas SQL
    static final int MAX_CONSULTAS = 10;

    HiloCliente(String nombre) {
        this.setName(nombre);
    }

    @Override
    public void run() {

        // Obtener una conexion
        Connection con = Main.pool.adquirir();

        // Mostrar estadísticas del pool
        printEstadisticas("UNA VEZ ADQUIRIDA LA CONEXION.");

        // Usar la conexion
        try {
            // Se ejecutaran MAX_CONSULTAS, pero cada hilo en un
            orden aleatorio
            // para que así la simulacion sea mas real
            for (int i=1; i<=MAX_CONSULTAS; i++) {
                String consulta =
                Utiles.getConsultaAleatoriamente();
                Statement s = con.createStatement();
                ResultSet rs = s.executeQuery(consulta);
                Utiles.mostrarSelect(rs, consulta);
                if (!s.isClosed()) {

```

```

        s.close();
    }
    if (!rs.isClosed()) {
        rs.close();
    }
}
} catch (SQLException ex) {
    LOG.error(ex);
} catch (Exception e) {
    LOG.error(e);
} finally { // Devolver la conexion al pool
    Main.pool.devolver(con);

    // Mostrar estadísticas del pool
    printEstadisticas("DESPUES DE DEVOLVER LA
CONEXION.");
}

}

private void printEstadisticas(String msjVariable) {
    if (LOG.isDebugEnabled()) {
        LOG.debug(msjVariable + " Conexiones -- disponibles:
" + Main.pool.getDisponiblesEnPool()
        + " Ocupadas: " +
Main.pool.getOcupadosEnPool()
        + " Totales: " +
Main.pool.getOcupacionTotal());
    }
}
}
}

```

Finalmente, la clase de utilidad:

Utiles.java

```

package creacionales.objectpool.main;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import org.apache.log4j.Logger;

public class HiloCliente extends Thread {

    // Disponer un log para que cada hilo escriba los mensajes que
    considere oportunos
    static final Logger LOG = Logger.getLogger(HiloCliente.class);

    // Cada hilo ejecutará 10 consultas SQL
    static final int MAX_CONSULTAS = 10;

    HiloCliente(String nombre) {

```

```

        this.setName(nombre);
    }

    @Override
    public void run() {

        // Obtener una conexion
        Connection con = Main.pool.adquirir();

        // Mostrar estadísticas del pool
        printEstadísticas("UNA VEZ ADQUIRIDA LA CONEXION.");

        // Usar la conexion
        try {
            // Se ejecutaran MAX_CONSULTAS, pero cada hilo en un
orden aleatorio
            // para que así la simulacion sea mas real
            for (int i=1; i<=MAX_CONSULTAS; i++) {
                String consulta =
Utiles.getConsultaAleatoriamente();
                Statement s = con.createStatement();
                ResultSet rs = s.executeQuery(consulta);
                Utiles.mostrarSelect(rs, consulta);
                if (!s.isClosed()) {
                    s.close();
                }
                if (!rs.isClosed()) {
                    rs.close();
                }
            }
        } catch (SQLException ex) {
            LOG.error(ex);
        } catch (Exception e) {
            LOG.error(e);
        } finally { // Devolver la conexion al pool
            Main.pool.devolver(con);

            // Mostrar estadísticas del pool
            printEstadísticas("DESPUES DE DEVOLVER LA
CONEXION.");
        }

    }

    private void printEstadísticas(String msjVariable) {
        if (LOG.isDebugEnabled()) {
            LOG.debug(msjVariable + " Conexiones -- disponibles:
" + Main.pool.getDisponiblesEnPool()
                + " Ocupadas: " +
Main.pool.getOcupadosEnPool()
                + " Totales: " +
Main.pool.getOcupacionTotal());
        }
    }
}

```

Finalmente, nos queda por ver una clase de utilidad en el lado del cliente. Está formada únicamente por métodos estáticos. Notad que el método `mostrarSelect()` tiene

comentadas las sentencias que imprimen el resultado del ResultSet. Esto es así para evitar el exceso de datos en el fichero de log y facilitar su análisis durante las pruebas.

Utiles.java

```
package creacionales.objectpool.main;

import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Properties;
import java.util.Random;

import org.apache.log4j.PropertyConfigurator;

import static creacionales.objectpool.main.HiloCliente.*;

/*
 * Clase que contiene métodos de utilidad (estáticos)
 */
class Utiles {

    /*
     * Carga el fichero de configuracion de Log4j
     * y lo inicializa
     */
    static void loadConfigLogging(Class<?> clase, String fichero,
                                  boolean ficheroEnPaquete)
    {
        Properties logProperties = new Properties();

        try { // Cargar la configuracion del log
            if (ficheroEnPaquete) {
                // Busca el fichero en el mismo paquete que la
clase
                logProperties.load(clase.getResourceAsStream(fichero));
            } else {
                // Busca en el classpath de la aplicacion
                logProperties.load(clase.getClassLoader().getResourceAsStream(fichero)
);
            }

            PropertyConfigurator.configure(logProperties);

        } catch (Exception e) {
            String msg="No se ha podido cargar el fichero de
configuracion de logging: "
                + fichero;
            System.out.println(msg);
        }
    }
}
```

```

/*
 * Aleatoriamente retorna una cadena que representa una de
 * las N consultas disponibles
 */
static String getConsultaAleatoriamente() {
    int numConsulta = new Random().nextInt(MAX_CONSULTAS)+1;
    String qry = null;
    switch (numConsulta) {
        case 1: // Mostrar las columnas de título y artista
de la tabla 'pel'
            qry = "SELECT titulo, artista FROM pel";
            break;
        case 2: // Mostrar TODAS las columnas de la tabla
'pel'.
            qry = "SELECT * FROM pel";
            break;
        case 3: // Mostrar todos los tipos de películas
existentes
            qry = "SELECT DISTINCT tipo FROM pel";
            break;
        case 4: // Mostrar todas las películas cuyo tipo sea
comedia.
            qry = "SELECT * FROM pel WHERE tipo='COMEDIA'";
            break;
        case 5: // Mostrar las películas de tipo comedia que
costaron a la empresa más de 12€.
            qry = "SELECT * FROM pel WHERE tipo='COMEDIA'
AND precio>12";
            break;
        case 6: // Listar el título de cada película, el
número de copias disponibles y la inversión hecha por la empresa.
            qry = "SELECT titulo, copias, precio*copias AS
inversion FROM pel";
            break;
        case 7: // Listar las películas en las que la empresa
ha invertido más de 60€.
            qry = "SELECT titulo, copias, precio*copias
FROM pel WHERE precio*copias>60";
            break;
        case 8: // Obtener el número de películas que hay en
la tabla 'pel'.
            qry = "SELECT COUNT(*) FROM pel";
            break;
        case 9: // Obtener la facturación total.
            qry = "SELECT SUM(total) FROM fac";
            break;
        case 10: // Obtener la inversión máxima, mínima y
promedio hecha por la compra de películas.
            qry = "SELECT MAX(precio*copias),
MIN(precio*copias), AVG(precio*copias) FROM pel";
            break;
        default:
            LOG.error("Se ha intentado ejecutar una
consulta cuyo numero no se contemplaba: "+numConsulta);
    }
    return qry;
}
/*

```



```

        * Metodo que evalua un ResultSet y lo imprime por pantalla
        */
        static void mostrarSelect(ResultSet res, String consulta) throws
SQLException {
            if (LOG.isDebugEnabled()) {
                LOG.debug(consulta);
            }
            ResultSetMetaData resMD = res.getMetaData();
            int num_cols = resMD.getColumnCount();

            res.beforeFirst();
            while (res.next()) {
                StringBuffer sbTitle = new StringBuffer();
                StringBuffer sbValues = new StringBuffer();

                for (int cont=0; cont<num_cols; cont++) {
                    sbTitle.append(resMD.getColumnName(cont+1));
                    sbTitle.append("\t");

                    if
(resMD.getColumnTypeName(cont+1).equals("DATE")) {
                        java.sql.Date fecha =
res.getDate(cont+1);
                        DateFormat df = new
SimpleDateFormat("dd/MM/yyyy");
                        String sDate = df.format(fecha);
                        sbValues.append(sDate);
                    } else {
                        sbValues.append(res.getString(cont+1));
                    }
                    sbValues.append("\t");
                }

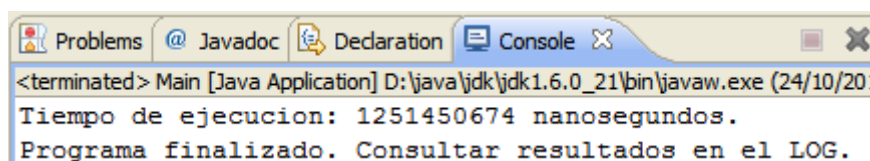
                //LOG.debug(sbTitle.toString());
                //LOG.debug(sbValues.toString());

                //LOG.debug("-----");
                //LOG.debug(sbValues.toString());
            }
        }
    }
}

```

A continuación se muestran dos imágenes relacionadas con las salidas generadas por la aplicación.

Salida por consola: se trata de la salida normal al finalizar la ejecución del programa.



```

<terminated> Main [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (24/10/20
Tiempo de ejecucion: 1251450674 nanosegundos.
Programa finalizado. Consultar resultados en el LOG.

```

Esta imagen es un fragmento de lo que sería el fichero de log de la aplicación (pool.log), ubicado en la raíz de la unidad C:/

```
[main] DEBUG Fichero log4j.properties cargado.
[main] DEBUG -----
[main] DEBUG INICIANDO APLICACION
[main] DEBUG -----
[main] DEBUG Creando pool de conexiones
[main] DEBUG Con cache de 3 conexiones y caducidad de 30 segundos.
[main] DEBUG Lanzando 10 hilos concurrentemente.
[main] DEBUG Arrancado el hilo 1
[main] DEBUG Arrancado el hilo 2
[main] DEBUG Arrancado el hilo 3
[Hilo 1] DEBUG Conexiones -- disponibles: 2 Ocupadas: 1 Totales: 3
[main] DEBUG Arrancado el hilo 4
[Hilo 2] DEBUG Conexiones -- disponibles: 1 Ocupadas: 2 Totales: 3
[main] DEBUG Arrancado el hilo 5
[main] DEBUG Arrancado el hilo 6
[main] DEBUG Arrancado el hilo 7
[main] DEBUG Arrancado el hilo 8
[main] DEBUG Arrancado el hilo 9
[Hilo 3] DEBUG Conexiones -- disponibles: 0 Ocupadas: 3 Totales: 3
[main] DEBUG Hilo 1 finalizado.
[Hilo 4] DEBUG Conexiones -- disponibles: 0 Ocupadas: 4 Totales: 4
[Hilo 4] DEBUG SELECT titulo, copias, precio*copias AS inversion FR
```

Problemas específicos e implementación

Veamos diferentes especialidades en las que pueden aplicar el patrón Object Pool, tales como la asignación de memoria, la gestión de conexiones, o la tecnología de objetos.

Pool de conexiones

El pool de conexiones es utilizado por frameworks (por ejemplo, EJB) o aplicaciones (sobretudo aplicaciones web) para acceder a servicios remotos, como servicios de bases de datos, a través de JDBC o servidores web, a través de HTTP.

Pool de Threads (hilos de ejecución, subprocessos)

Los pools de threads se utilizan habitualmente en entornos que son inherentemente asíncronos. Los sistemas altamente escalables y los sistemas en tiempo real utilizan un pool de threads como mecanismo fundamental para la gestión de varios hilos de ejecución. Sin un pool de threads, este tipo de sistemas agotarían fácilmente los recursos del sistema subyacente, ya que estas aplicaciones tienden a hacer un uso intensivo de threads, necesitando que haya muchos simultáneamente en ejecución. Un típico pool de threads al iniciarse obtiene de manera anticipadamente un número

determinado de threads para cachearlos. Cuando sea necesario, de manera perezosa adquirirá otros desde el Entorno del recurso.

Un caso típico de utilización de este tipo de pools son los servidores web. Un servidor web han de atender cientos o miles de solicitudes simultáneas, la mayoría de las cuales se procesan rápidamente, lo que no se podría conseguir creando un nuevo thread por petición (sería imposible o muy ineficiente crear en un mismo instante 1000 threads, por ejemplo). La mayoría de los servidores web utilizan un pool de threads para gestionarlos de manera eficaz. Los threads se vuelven a utilizar después de completar una solicitud (de cerrar un ciclo petición/respuesta).

Pool de instancias para componentes

Este tipo de pool es el empleado normalmente por los servidores de aplicaciones. Los Enterprise Java Bean (EJB) son un claro ejemplo de componentes cuyas instancias residen en un pool de estas características. Los servidores de aplicaciones optimizan el uso sus recursos cacheando una cantidad limitada de componentes que son muy utilizados. De esta manera pueden servir componentes rápidamente durante las primeras peticiones de los usuarios. Cuando no quedan componentes en el pool, el servidor instancia tantos como sean necesarios para atender la demanda. Cuando un cliente deja de utilizar un componente, éste se recicla y queda disponible nuevamente en el pool.

Pool de memoria

Un pool de memoria reserva bloques de memoria para servir rápidamente peticiones de adquisición de memoria. Los bloques, una vez devueltos, se reciclan para poderse reutilizar en nuevas peticiones. Normalmente, este tipo de pool no libera la memoria hasta que el sistema/aplicación finaliza. Es frecuente que un sistema disponga de varios pools de memoria, cada uno con un tamaño de bloque diferente, con el fin de evitar que se desperdicien bloques grandes en solicitudes de pequeño tamaño.

Beneficios de utilizar el patrón Object Pool

Los principales beneficios son los siguientes:

Rendimiento (Performance)

Object Pool puede mejorar el rendimiento de una aplicación, ya que ayuda a reducir el tiempo empleado en la adquisición y liberación de recursos computacionalmente costosos de crear, inicializar y destruir. En los casos en que las peticiones de adquisición se satisfacen mediante la caché, el tiempo de adquisición es muy bajo.

Previsibilidad (Predictability)

Si un pool al iniciar adquiere anticipadamente recursos y sigue una estrategia de desalojo adecuada (para evitar su saturación), puede satisfacer correctamente la mayoría de las peticiones de adquisición enviadas por los usuarios. Esto permite establecer un promedio de los recursos que un usuario puede llegar a obtener en un determinado tiempo. Si no se utiliza un pool, la aplicación del servidor (en caso de una aplicación web) o una aplicación de escritorio, tienen que adquirir directamente los recursos del Entorno del recurso, lo cual difícilmente permitirá fijar una media aceptable de recursos por unidad de tiempo, luego la adquisición mediante pool es mucho más previsible que la adquisición directa.

Simplicidad (Simplicity)

El uso de un pool simplifica la programación de la clase cliente, ya que ésta no tiene que conocer determinados aspectos del Entorno del recurso. Normalmente, para obtener un recurso invocará a una operación adquirir() y para retornarlo al pool, a una operación devolver().

Estabilidad y Escalabilidad (Stability and Scalability)

En un pool se crean nuevos recursos si la demanda excede los recursos cacheados. El pool no debe entregar un recurso devuelto por un usuario sistemáticamente al Entorno del recurso, puesto que los recursos se deben reciclar para entregarlos en nuevas solicitudes. No obstante, con el fin de no llenarse y degradar el sistema, el pool tiene que utilizar una estrategia de desalojo que devuelva definitivamente un recurso a su entorno original, normalmente dependiendo del tiempo sin usarse que lleva un recurso. Esto evita las costosas operaciones de liberación y de adquisición de recursos, lo que aumenta el rendimiento general del sistema y la estabilidad.

Compartir (Sharing)

Un pool se caracteriza por compartir en diferentes momentos recursos no utilizados. Esto también beneficia el uso de memoria, por lo que resulta en una reducción global

del consumo de memoria de la aplicación. Hay que advertir que para que los recursos se puedan compartir es necesario implementar un mecanismo de sincronización.

Patrones relacionados

Adquisición Perezosa o Bajo Demanda (Eager Acquisition)

Describe cómo retardar la adquisición de recursos hasta el último momento a fin de no desperdiciar recursos innecesariamente.

Adquisición Aticipada (Lazy Acquisition)

Describe la forma de cachear recursos por adelantado a fin de permitir a los usuarios la adquisición rápida y predecible de recursos.

Desalojador (Evictor)

Describe cómo, mediante la monitorización de su uso, se pueden liberar recursos del pool que no estén siendo utilizados con regularidad.

Almacenamiento en caché (Caching)

El patrón Caching es parecido a Object Pool, aunque la principal diferencia es que Caching gestiona recursos que tienen identidad, por lo que son los usuarios de los recursos quienes debe responsabilizarse de devolver a la caché el objeto X y no el Y. En el caso de Object Pool, los usuarios de los recursos no se ocupan de su identidad, dado que todos los recursos del pool son idénticos.