UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Sesión 17: Patrones de diseño básicos: singleton, adaptador, iterator.

Recordando:

¿Que vimos la clase pasada?

Universidad Tecnológica del Perú

Logro de aprendizaje

Al finalizar la sesión, el estudiante soluciona problemas aplicando patrones de diseño: singleton, adaptador, iterator usando Java en la resolución de ejercicios.

Universidad Tecnológica del Perú

Utilidad

Los patrones de diseño básicos como Singleton, Adaptador e Iterator son fundamentales en la programación orientada a objetos (POO) porque proporcionan soluciones comprobadas a problemas comunes de diseño de software.

Saberes Previos

- Conceptos básicos de programación orientada a objetos.
- Familiaridad con el lenguaje de programación (preferiblemente Java).
- Comprender la importancia de la reutilización de código y la mantenibilidad del software.

Importancia de Singleton

- **Control de acceso a recursos únicos:** Garantiza que una clase tenga una única instancia, evitando la creación de múltiples objetos que podrían consumir recursos de manera innecesaria o causar inconsistencias.
- **Punto de acceso global:** Proporciona un punto de acceso global a la instancia única, lo cual es útil para gestionar recursos compartidos como conexiones de base de datos, registros (logs), o configuraciones globales.
- **Facilidad de mantenimiento:** Centraliza la lógica de control de la instancia, lo que facilita el mantenimiento y las modificaciones futuras.

Importancia de Adaptador (Adapter)

- **Interoperabilidad:** Permite que clases con interfaces incompatibles trabajen juntas, facilitando la integración de código existente con nuevas interfaces o bibliotecas sin necesidad de modificar el código original.
- **Reutilización de código:** Permite reutilizar código existente en nuevas aplicaciones al adaptar su interfaz a la requerida por el nuevo contexto.
- **Flexibilidad y extensión:** Facilita la extensión de la funcionalidad de clases existentes sin alterar su código, lo cual es especialmente útil cuando se trabaja con código de terceros o librerías externas.

Importancia de Iterator

- **Abstracción de iteración:** Proporciona una forma de acceder secuencialmente a los elementos de una colección sin exponer su representación interna. Esto hace que el código que utiliza iteradores sea más general y reutilizable.
- **Encapsulación:** Encapsula la lógica de iteración, separándola del código cliente. Esto facilita cambios en la estructura de la colección sin necesidad de modificar el código que la utiliza.
- **Compatibilidad con diferentes colecciones**: Permite que el mismo código de iteración funcione con diferentes tipos de colecciones, siempre y cuando implementen el patrón Iterator.

Beneficios Generales de los Patrones de Diseño

1. **Mejora en la mantenibilidad:**
   Los patrones de diseño facilitan la mantenibilidad del código al proporcionar soluciones estandarizadas y probadas para problemas comunes.
2. **Facilita la comunicación:**
   Al usar nombres y conceptos estandarizados, los patrones de diseño facilitan la comunicación entre desarrolladores, ya que todos entienden qué problema resuelve cada patrón y cómo se implementa.

Beneficios Generales de los Patrones de Diseño

3. **Flexibilidad y escalabilidad:**
   Los patrones de diseño permiten crear sistemas más flexibles y escalables al proporcionar estructuras que pueden adaptarse a cambios y crecer con el tiempo.
4. **Reducción de errores:**
   Al seguir patrones de diseño probados, se reducen los errores comunes y se promueve la escritura de código más robusto y confiable.

Singleton

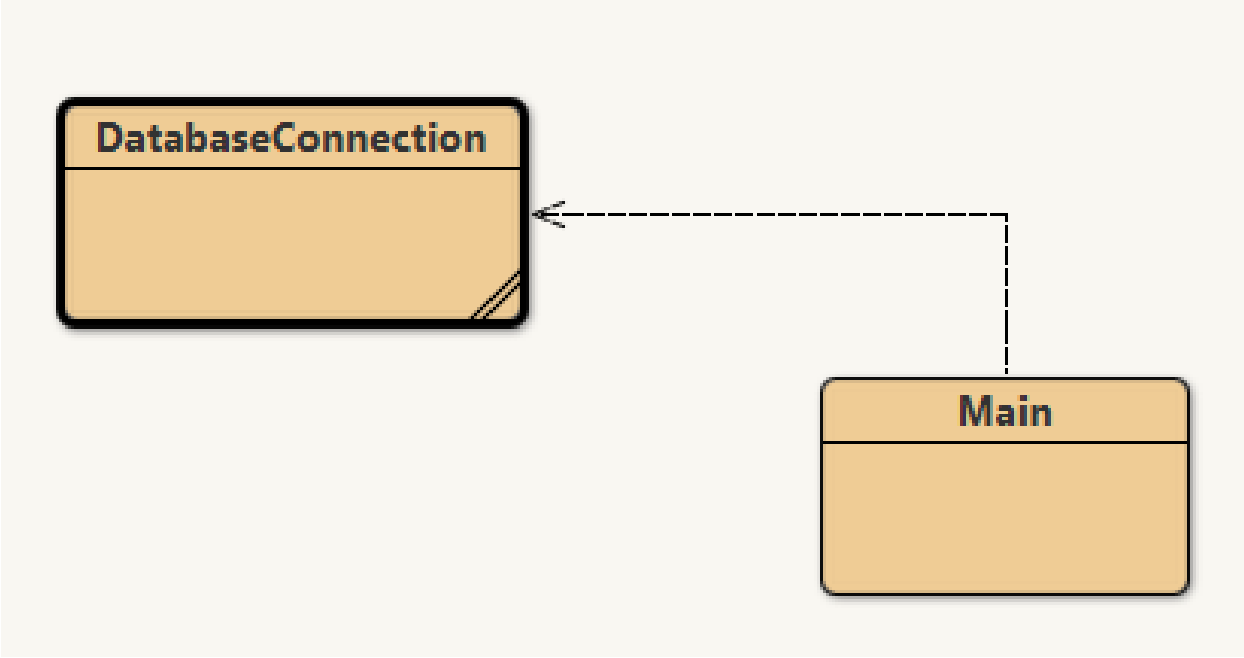
El patrón Singleton asegura que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia.

Singleton

Ejercicio: Implementa un Singleton que gestione una única conexión de base de datos en una aplicación.

```
CREATE DATABASE mydatabase;
CREATE TABLE users ( id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, email VARCHAR(255) NOT NULL );
INSERT INTO users (name, email) VALUES ('Alice', 'alice@example.com'); INSERT INTO users (name, email) VALUES ('Bob', 'bob@example.com'); INSERT INTO users (name, email) VALUES ('Charlie', 'charlie@example.com');
```

Singleton



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

public class DatabaseConnection {
    // Instancia única de la clase DatabaseConnection
    private static DatabaseConnection instance;
    // Objeto Connection para la base de datos
    private Connection connection;
    // URL, usuario y contraseña para la conexión de base de datos
    private String url = "jdbc:mysql://localhost:3306/mydatabase";
    private String username = "root";
    private String password = "";

    // Constructor privado para evitar la instanciación
    private DatabaseConnection() {
        try {
            // Cargar el controlador de la base de datos
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Establecer la conexión a la base de datos
            this.connection = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
```

Clase DatabaseConnection

Universidad Tecnológica del Perú

```
// Método público estático para obtener la instancia
única
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }

    // Método para obtener la conexión de base de datos
    public Connection getConnection() {
        return connection;
    }

    // Método para cerrar la conexión de base de datos
    public void closeConnection() {
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Clase DatabaseConnection

Universidad Tecnológica del Perú

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

public class Main {
    public static void main(String[] args) {
        // Obtener la instancia única de DatabaseConnection
        DatabaseConnection dbConnection = DatabaseConnection.getInstance();
        // Obtener la conexión de base de datos
        Connection connection = dbConnection.getConnection();

        try {
            // Crear una declaración SQL
            Statement statement = connection.createStatement();
            // Ejecutar una consulta SQL
            ResultSet resultSet = statement.executeQuery("SELECT * FROM users");
```

Clase Main

```
// Procesar los resultados de la consulta
        while (resultSet.next()) {
            System.out.println("ID: " + resultSet.getInt("id"));
            System.out.println("Name: " + resultSet.getString("name"));
            System.out.println("Email: " + resultSet.getString("email"));
            System.out.println("---------");
        }

        // Cerrar el ResultSet y el Statement
        resultSet.close();
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        // Cerrar la conexión de base de datos
        dbConnection.closeConnection();
    }
  }
}
```

Clase Main

Explicación

1. DatabaseConnection.java

   - La clase `DatabaseConnection` es el Singleton que gestiona la conexión de base de datos.
   - El constructor es privado para evitar la creación de nuevas instancias.
   - El método `getInstance` proporciona la instancia única de la clase, utilizando la técnica de doble verificación de bloqueo para asegurar que sólo una instancia sea creada en un entorno de múltiples hilos.
   - La conexión de base de datos se establece en el constructor.
   - El método `getConnection` devuelve la conexión de base de datos.
   - El método `closeConnection` cierra la conexión de base de datos.

Explicación

2. Main.java

- La clase `Main` muestra cómo usar el Singleton para obtener una conexión de base de datos y ejecutar una consulta.
- Se obtiene la instancia única de `DatabaseConnection`.
- Se usa la conexión para crear un `Statement` y ejecutar una consulta.
- Se procesan los resultados de la consulta y se cierran los recursos.

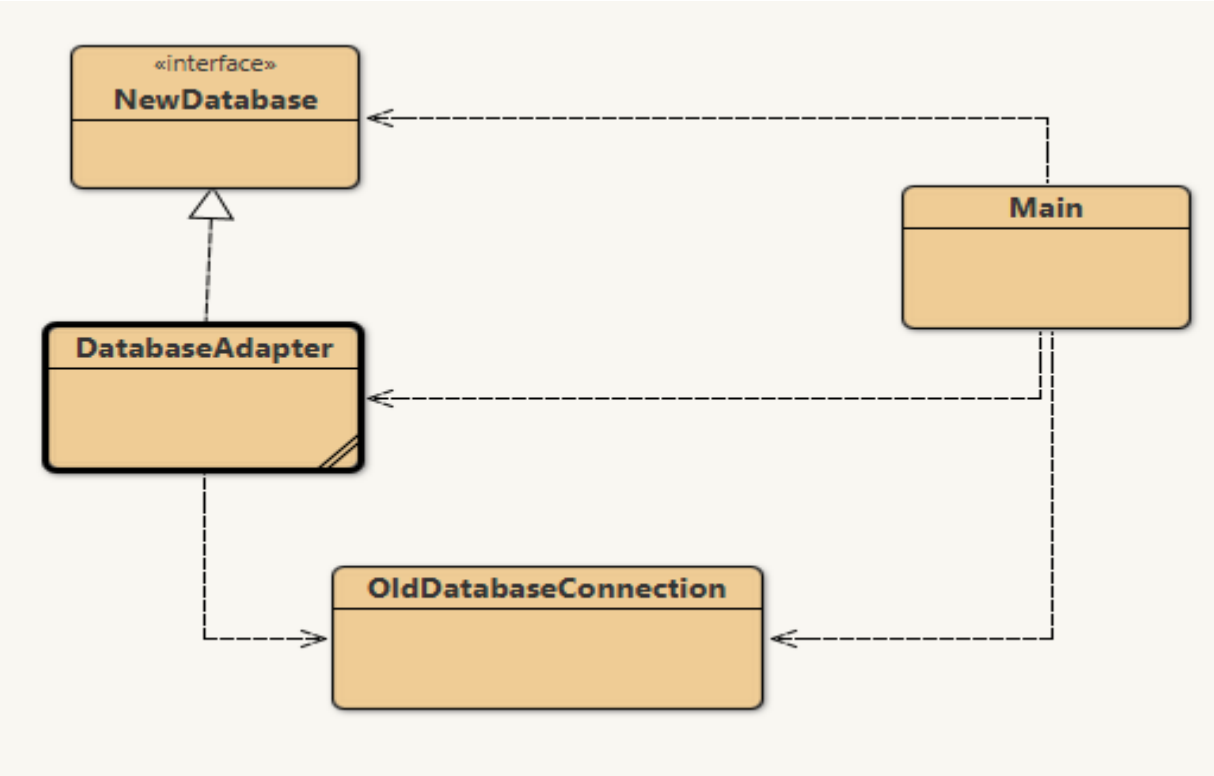
Este ejemplo asume que tienes una base de datos MySQL corriendo en localhost en el puerto 3306, con una base de datos llamada mydatabase y una tabla users con columnas id, name, y email. Asegúrate de ajustar los detalles de la conexión (URL, usuario, contraseña) según tu configuración.

Adaptador (Adapter)

El patrón Adaptador permite que clases con interfaces incompatibles trabajen juntas.

Ejercicio: Supongamos que tienes una clase antigua que maneja la conexión a la base de datos y una nueva interfaz que deseas implementar.

Adaptador (Adapter)



interface NewDatabase

```
public interface NewDatabase {
    void openConnection();
    void closeConnection();
    void runQuery(String query);
}
```

```
public class DatabaseAdapter implements NewDatabase {
    private OldDatabaseConnection oldDatabaseConnection;

    public DatabaseAdapter(OldDatabaseConnection
oldDatabaseConnection) {
        this.oldDatabaseConnection = oldDatabaseConnection;
    }

    @Override
    public void openConnection() {
        oldDatabaseConnection.connect();
    }

    @Override
    public void closeConnection() {
        oldDatabaseConnection.disconnect();
    }

    @Override
    public void runQuery(String query) {
        oldDatabaseConnection.executeQuery(query);
    }
}
```

clase DatabaseAdapter

clase OldDatabaseConnection

```
public class OldDatabaseConnection {
    public void connect() {
        System.out.println("Conectándose usando una conexión de base de datos antigua...");
    }

    public void disconnect() {
        System.out.println("Desconectándose usando una conexión de base de datos antigua...");
    }

    public void executeQuery(String query) {
        System.out.println("Ejecutando consulta en conexión de base de datos antigua:" + query);
    }
}
```

clase main

```
public class Main {
    public static void main(String[] args) {
        // Instancia de la conexión antigua
        OldDatabaseConnection oldDatabaseConnection = new OldDatabaseConnection();

        // Adaptador que permite usar la nueva interfaz
        NewDatabase newDatabase = new DatabaseAdapter(oldDatabaseConnection);

        // Usando la nueva interfaz para manejar la conexión antigua
        newDatabase.openConnection();
        newDatabase.runQuery("SELECT * FROM users");
        newDatabase.closeConnection();
    }
}
```

Implementación

1. **OldDatabaseConnection.java**

   - Esta clase representa la conexión a la base de datos utilizando una interfaz antigua. Proporciona métodos para conectar, desconectar y ejecutar consultas en la base de datos.

2. **NewDatabase.java**

   - Esta interfaz define la nueva forma estándar de interactuar con la base de datos, con métodos para abrir y cerrar la conexión, y ejecutar consultas.

Implementación

3. DatabaseAdapter.java

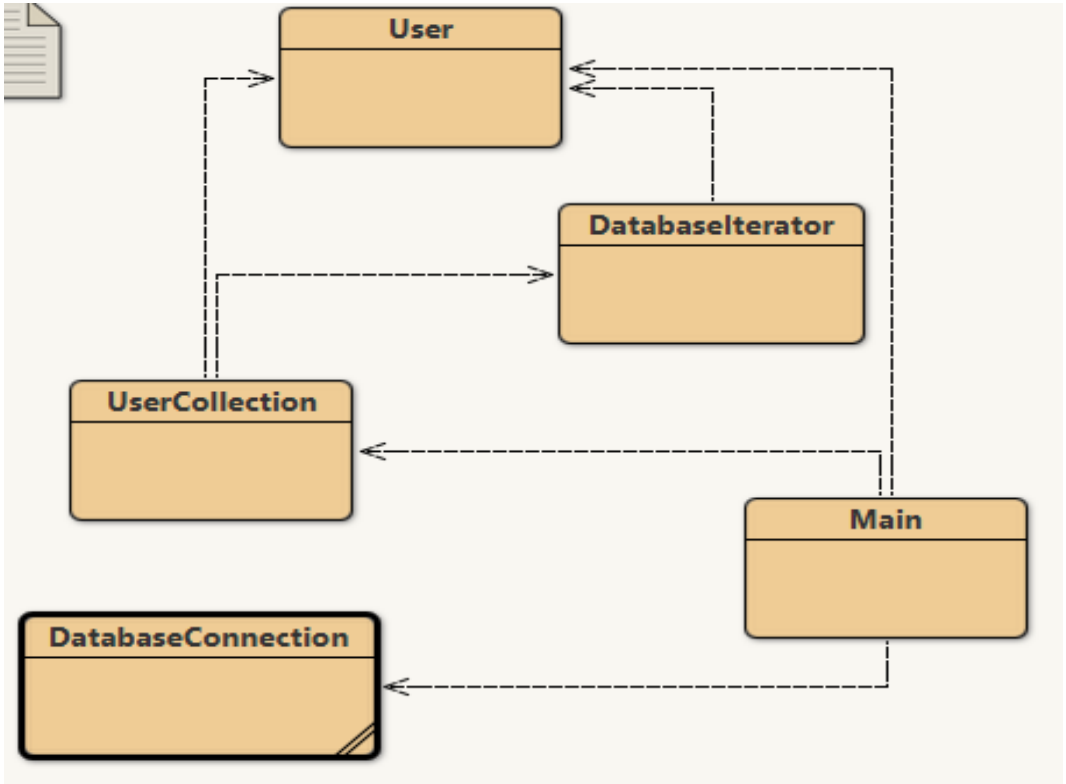
   - Esta clase adapta `OldDatabaseConnection` a la nueva interfaz `NewDatabase`. Implementa los métodos de `NewDatabase` y delega las llamadas a la instancia de `OldDatabaseConnection`.

4. Main.java

   - En esta clase, se demuestra cómo usar el adaptador para trabajar con la nueva interfaz mientras se sigue utilizando la implementación de la conexión antigua.

Este ejemplo muestra cómo el patrón Adaptador puede ser utilizado para integrar una clase existente con una nueva interfaz sin modificar la clase original. Esto es especialmente útil cuando se necesita actualizar o mejorar el sistema sin alterar el código base antiguo.

Iterador



Ejercicio sencillo de cómo implementar el patrón Iterator para iterar sobre los resultados de una consulta a una base de datos.

```
public class User {
    private int id;
    private String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "User{id=" + id + ", name='" + name + "'}";
    }
}
```

Crear el Modelo de Datos

clase User

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class DatabaseIterator implements Iterator<User> {
    private ResultSet resultSet;

    public DatabaseIterator(ResultSet resultSet) {
        this.resultSet = resultSet;
    }

    @Override
    public boolean hasNext() {
        try {
            return resultSet != null && resultSet.next();
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }
```

Crear el Iterador

Clase DatabaseIterator

```
    @Override
    public User next() {
        try {
            if (resultSet == null || resultSet.isAfterLast()) {
                throw new NoSuchElementException();
            }
            int id = resultSet.getInt("id");
            String name = resultSet.getString("name");
            return new User(id, name);
        } catch (SQLException e) {
            e.printStackTrace();
            throw new NoSuchElementException();
        }
    }
}
```

Crear el Iterador

Clase DatabaseIterator

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Iterator;

public class UserCollection implements Iterable<User> {
    private Connection connection;

    public UserCollection(Connection connection) {
        this.connection = connection;
    }

    @Override
    public Iterator<User> iterator() {
        ResultSet resultSet = null;
        try {
            Statement statement = connection.createStatement();
            resultSet = statement.executeQuery("SELECT id, name FROM users");
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return new DatabaseIterator(resultSet);
    }
}
```

Crear la Colección que usa el Iterador

Clase UserCollection

Universidad Tecnológica del Perú

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

public class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;
    private String url = "jdbc:mysql://localhost:3306/mydatabase";
    private String username = "root";
    private String password = "";

    private DatabaseConnection() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            this.connection = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
```

Implementar el Singleton para la Conexión a la Base de Datos

Clase DatabaseConnection

Universidad Tecnológica del Perú

```
public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }

    public void closeConnection() {
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Implementar el Singleton para la Conexión a la Base de Datos

Clase DatabaseConnection

Universidad Tecnológica del Perú

```
import java.sql.Connection;

public class Main {
    public static void main(String[] args) {
        DatabaseConnection dbConnection =
DatabaseConnection.getInstance();
        Connection connection = dbConnection.getConnection();

        UserCollection userCollection = new UserCollection(connection);

        for (User user : userCollection) {
            System.out.println(user);
        }

        dbConnection.closeConnection();
    }
}
```

Usar el Iterador con la Base de Datos

Clase Main

Explicación

1. **User.java:** Clase modelo para representar los usuarios.
2. **DatabaseIterator.java:** Implementa el patrón Iterator para iterar sobre los resultados de una consulta SQL.
3. **UserCollection.java:** Proporciona un método `iterator()` que devuelve un `DatabaseIterator` para iterar sobre los usuarios obtenidos de la base de datos.
4. **DatabaseConnection.java:** Implementa un Singleton para gestionar una única conexión a la base de datos.
5. **Main.java:** Muestra cómo usar la clase `UserCollection` para iterar sobre los usuarios obtenidos de la base de datos y cerrar la conexión.

Este ejemplo es sencillo y debe funcionar sin problemas. Asegúrate de ajustar los detalles de la conexión (URL, usuario, contraseña) según tu configuración de MySQL.

Practica

Implementa los ejercicios vistos.

Conclusiones:

En resumen, los patrones de diseño como Singleton, Adaptador e Iterator son herramientas valiosas en la programación orientada a objetos porque ayudan a resolver problemas de diseño recurrentes de manera eficiente y efectiva, mejorando la calidad, la reutilización y la mantenibilidad del software.

Universidad Tecnológica del Perú

Cierre

¿Qué hemos aprendido hoy?

Bibliografía

- MORENO PÉREZ, J. “Programación orientada a objetos”. RA-MA Editorial. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=31933
- Vélez Serrano, José. “Diseñar y programar, todo es empezar: una introducción a la Programación Orientada a Objetos usando UML y Java”. Dykinson. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=36368

Universidad Tecnológica del Perú