UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Sesión 17: Patrones de diseño básicos : builder, facade, observer, state.

Recordando:

¿Que vimos la clase pasada?

Universidad Tecnológica del Perú

Logro de aprendizaje

Al finalizar la sesión, el estudiante soluciona problemas aplicando los Patrones de diseño básicos : builder, facade, observer, state usando Java en la resolución de ejercicios.

Utilidad

Los patrones de diseño básicos como Builder, Facade, Observer y State son herramientas esenciales en la programación orientada a objetos con Java, especialmente cuando se trabaja con bases de datos. A continuación, se detallan las utilidades específicas de cada uno de estos patrones en este contexto.

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.

Builder Pattern

Importancia:

- **Facilita la Creación de Objetos Complejos:** Permite construir objetos paso a paso, ofreciendo una interfaz clara para configurar sus propiedades.
- **Inmutabilidad:** Fomenta la creación de objetos inmutables, lo cual es beneficioso para la concurrencia y la estabilidad.
- **Legibilidad:** Mejora la legibilidad del código al hacer explícito el proceso de construcción del objeto.

Ejemplo en Base de Datos:

En la configuración de conexiones a bases de datos, el patrón Builder puede facilitar la creación de objetos de conexión con múltiples parámetros opcionales, como el tiempo de espera, el número de conexiones máximas, etc.

Facade Pattern

Importancia:

- **Simplicidad:** Proporciona una interfaz simplificada a un subsistema complejo, haciendo que el código sea más fácil de usar y entender.
- **Desacoplamiento:** Reduce las dependencias entre el sistema y los subsistemas, facilitando la mantención y evolución del código.
- **Reusabilidad:** Encapsula la funcionalidad común, promoviendo la reutilización de código.

Ejemplo en Base de Datos:

Al interactuar con una base de datos, una fachada puede simplificar operaciones complejas como conexiones, transacciones, y consultas, proporcionando métodos sencillos para ejecutar estas tareas.

Observer Pattern

Importancia:

- **Desacoplamiento:** Permite una comunicación flexible entre objetos, donde los observadores se registran para recibir actualizaciones sin depender directamente del sujeto.
- **Reactividad:** Facilita la implementación de sistemas reactivos donde los cambios en el estado del objeto desencadenan acciones en otros objetos.
- **Escalabilidad:** Facilita la adición de nuevos observadores sin modificar el código existente del sujeto.

Ejemplo en Base de Datos:

En una aplicación de monitoreo de bases de datos, el patrón Observer puede usarse para notificar a los administradores o sistemas de alerta sobre eventos específicos como cambios en el esquema de la base de datos o la aparición de errores.

State Pattern

Importancia:

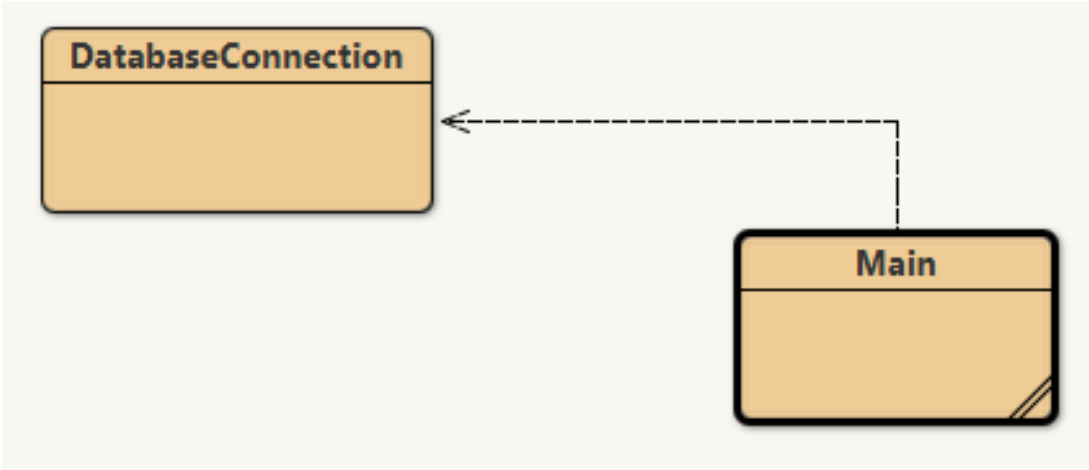
- **Claridad:** Facilita la comprensión y gestión de los estados de un objeto, encapsulando el comportamiento específico de cada estado.
- **Flexibilidad:** Permite cambiar el comportamiento de un objeto en tiempo de ejecución cambiando su estado interno.
- **Mantenibilidad:** Aísla la lógica de los estados, haciendo el código más modular y fácil de mantener.

Ejemplo en Base de Datos:

En un sistema de gestión de transacciones, el patrón State puede manejar los diferentes estados de una transacción (iniciada, en progreso, confirmada, cancelada), encapsulando la lógica correspondiente a cada estado.

Builder Pattern

El patrón Builder se usa para construir un objeto complejo paso a paso. Es útil cuando un objeto tiene muchos parámetros opcionales o complejos.



```
public class DatabaseConnection {
    private String url;
    private String username;
    private String password;

    private DatabaseConnection(DatabaseConnectionBuilder builder) {
        this.url = builder.url;
        this.username = builder.username;
        this.password = builder.password;
    }

    // Getters opcionales para acceder a los atributos
    public String getUrl() {
        return url;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }
```

Clase DatabaseConnection

```
public static class DatabaseConnectionBuilder {
    private String url;
    private String username;
    private String password;

    public DatabaseConnectionBuilder setUrl(String url) {
        if (url == null || url.isEmpty()) {
            throw new IllegalArgumentException("La URL no puede ser nula o vacía");
        }
        this.url = url;
        return this;
    }

    public DatabaseConnectionBuilder setUsername(String username) {
        if (username == null || username.isEmpty()) {
            throw new IllegalArgumentException("El nombre de usuario no puede ser nulo o vacío");
        }
        this.username = username;
        return this;
    }
```

Clase DatabaseConnection

```
public DatabaseConnectionBuilder setPassword(String password) {
        if (password == null || password.isEmpty()) {
            throw new IllegalArgumentException("La contraseña no puede ser nula o vacía");
        }
        this.password = password;
        return this;
    }

    public DatabaseConnection build() {
        // Se pueden agregar más validaciones antes de construir el objeto
        if (this.url == null || this.username == null || this.password == null) {
            throw new IllegalStateException("No se puede crear DatabaseConnection con
valores nulos");
        }
        return new DatabaseConnection(this);
    }
  }
}
```

Clase DatabaseConnection

```
public class Main {
    public static void main(String[] args) {
        DatabaseConnection connection = new
DatabaseConnection.DatabaseConnectionBuilder()
            .setUrl("jdbc:mysql://localhost:3306/mydb")
            .setUsername("root")
            .setPassword(" ")
            .build();

        // Acceso a los atributos si se definieron getters
        System.out.println("URL: " + connection.getUrl());
        System.out.println("Username: " + connection.getUsername());
    }
}
```

Clase Main

Explicación

- **Constructor Privado de la Clase DatabaseConnection:**

Está correctamente definido como privado para evitar que se cree una instancia sin usar el Builder.

- **Atributos del DatabaseConnectionBuilder:**

Están definidos correctamente.

- **Métodos de Construcción (setUrl, setUsername, setPassword):**

Están definidos correctamente y retornan el propio builder para permitir el encadenamiento de métodos.

- **Método build():**

Está correctamente definido para retornar una nueva instancia de DatabaseConnection.

Explicación

- **Validaciones en los Métodos del Builder:**

Se añadieron validaciones para asegurar que los valores no sean nulos ni vacíos.

- **Getters:**

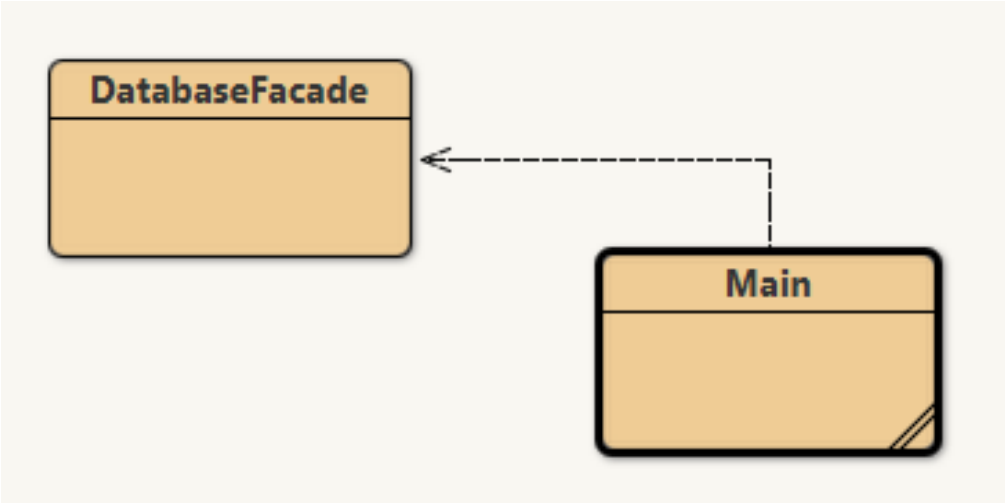
Se añadieron getters para acceder a los atributos de DatabaseConnection si es necesario.

- **Manejo de Excepciones:**

Se utilizan IllegalArgumentException y IllegalStateException para manejar situaciones de valores inválidos o incompletos.

Facade Pattern

Este código proporciona una estructura sólida para trabajar con bases de datos en Java, encapsulando la lógica de conexión y ejecución de consultas dentro de una clase fachada fácil de usar.



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

public class DatabaseFacade {
    private Connection connection;

    // Método para conectar a la base de datos
    public void connect(String url, String username, String password) throws
SQLException {
        if (connection != null && !connection.isClosed()) {
            throw new IllegalStateException("Ya conectado a la base de datos");
        }
        connection = DriverManager.getConnection(url, username, password);
    }
```

Clase DatabaseFacade

```
// Método para ejecutar una consulta
    public void executeQuery(String query) throws SQLException {
        if (connection == null || connection.isClosed()) {
            throw new IllegalStateException("No conectado a la base de datos");
        }

        try (Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(query)) {

            while (resultSet.next()) {
                // Procesar ResultSet (ejemplo)
                System.out.println("Column1: " + resultSet.getString(1));
            }
        }
    }
```

Clase DatabaseFacade

```
// Método para desconectar de la base de datos
    public void disconnect() {
        if (connection != null) {
            try {
                if (!connection.isClosed()) {
                    connection.close();
                    System.out.println("La conexión se cerró correctamente.");
                }
            } catch (SQLException e) {
                System.err.println("Error al cerrar la conexión: " + e.getMessage());
            } finally {
                connection = null;
            }
        } else {
            System.out.println("La conexión ya está cerrada o nunca se abrió.");
        }
    }
}
```

Clase DatabaseFacade

```
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) {
        DatabaseFacade dbFacade = new DatabaseFacade();
        try {
            dbFacade.connect("jdbc:mysql://localhost:3306/mydb", "root", "");
            dbFacade.executeQuery("SELECT * FROM users");
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            dbFacade.disconnect();
        }
    }
}
```

Clase Main

Explicación

Clase `DatabaseFacade`

java Copiar código

```java
public class DatabaseFacade {
    private Connection connection;
```

La clase `DatabaseFacade` actúa como una fachada simplificada para manejar las operaciones básicas de la base de datos. La variable `connection` es una instancia de `Connection` que se usará para conectarse a la base de datos.

Explicación

Método `connect`

java Copiar código

```java
public void connect(String url, String username, String password) throws SQLException {
    if (connection != null && !connection.isClosed()) {
        throw new IllegalStateException("Already connected to the database");
    }
    connection = DriverManager.getConnection(url, username, password);
}
```

Este método establece una conexión con la base de datos.

1. **Verificación de Conexión Existente:** Comprueba si ya hay una conexión abierta. Si la conexión no es nula y no está cerrada, lanza una excepción.
2. **Establecimiento de Conexión:** Usa `DriverManager.getConnection` para establecer una conexión con la base de datos utilizando la URL, el nombre de usuario y la contraseña proporcionados.

Explicación

Método `executeQuery`

java

Copiar código

```java
public void executeQuery(String query) throws SQLException {
    if (connection == null || connection.isClosed()) {
        throw new IllegalStateException("Not connected to the database");
    }

    try (Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(query)) {

        while (resultSet.next()) {
            // Procesar ResultSet (ejemplo)
            System.out.println("Column1: " + resultSet.getString(1));
        }
    }
}
```

Este método ejecuta una consulta SQL y procesa los resultados.

1. **Verificación de Conexión:** Comprueba si la conexión es nula o está cerrada. Si es así, lanza una excepción.
2. **Ejecución de Consulta:** Usa un `Statement` para ejecutar la consulta SQL. El resultado se guarda en un `ResultSet`.
3. **Procesamiento de Resultados:** Itera sobre el `ResultSet` para procesar cada fila de los resultados. Aquí, simplemente imprime el valor de la primera columna.

Universidad Tecnológica del Perú

Explicación

Método `disconnect`

java

Copiar código

```java
public void disconnect() {
    if (connection != null) {
        try {
            if (!connection.isClosed()) {
                connection.close();
                System.out.println("Connection closed successfully.");
            }
        } catch (SQLException e) {
            System.err.println("Error while closing the connection: " + e.getMessage());
        } finally {
            connection = null;
        }
    } else {
        System.out.println("Connection is already closed or was never opened.");
    }
}
```

Este método cierra la conexión con la base de datos.

1. **Verificación de Conexión:** Comprueba si la conexión es nula. Si no lo es, procede a intentar cerrar la conexión.
2. **Cierre de Conexión:** Cierra la conexión y maneja cualquier `SQLException` que pueda ocurrir durante el cierre.
3. **Liberación de Recursos:** Establece la variable `connection` a `null` para liberar el recurso.
4. **Mensaje Informativo:** Imprime mensajes para informar si la conexión ya estaba cerrada o nunca fue abierta.

Explicación

```java
public class Main {
    public static void main(String[] args) {
        DatabaseFacade dbFacade = new DatabaseFacade();
        try {
            dbFacade.connect("jdbc:mysql://localhost:3306/mydb", "root", "password");
            dbFacade.executeQuery("SELECT * FROM users");
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            dbFacade.disconnect();
        }
    }
}
```

Este es el punto de entrada de la aplicación.

1. Creación de `DatabaseFacade`: Crea una instancia de `DatabaseFacade`.
2. Conexión a la Base de Datos: Intenta conectar a la base de datos usando las credenciales proporcionadas.
3. Ejecución de Consulta: Ejecuta una consulta para seleccionar todos los registros de la tabla `users`.
4. Manejo de Excepciones: Atrapa y maneja cualquier `SQLException` que ocurra.
5. Desconexión: En el bloque `finally`, asegura que la conexión a la base de datos se cierre correctamente.

Explicación

- `DatabaseFacade` simplifica la interacción con la base de datos proporcionando métodos claros para conectar, ejecutar consultas y desconectar.
- **Manejo Adecuado de Recursos:** Se asegura de que los recursos se manejen adecuadamente usando bloques `try-with-resources` y validaciones.
- **Robustez y Seguridad:** Maneja excepciones y asegura que la conexión se cierre incluso si ocurre un error.

Este código proporciona una estructura sólida para trabajar con bases de datos en Java, encapsulando la lógica de conexión y ejecución de consultas dentro de una clase fachada fácil de usar.

Observer

En este ejercicio, implementaremos el patrón Observer donde los observadores se actualizarán con los cambios de estado que se guardarán en una base de datos

Necesitamos configurar una base de datos y crearemos una tabla para almacenar el estado.

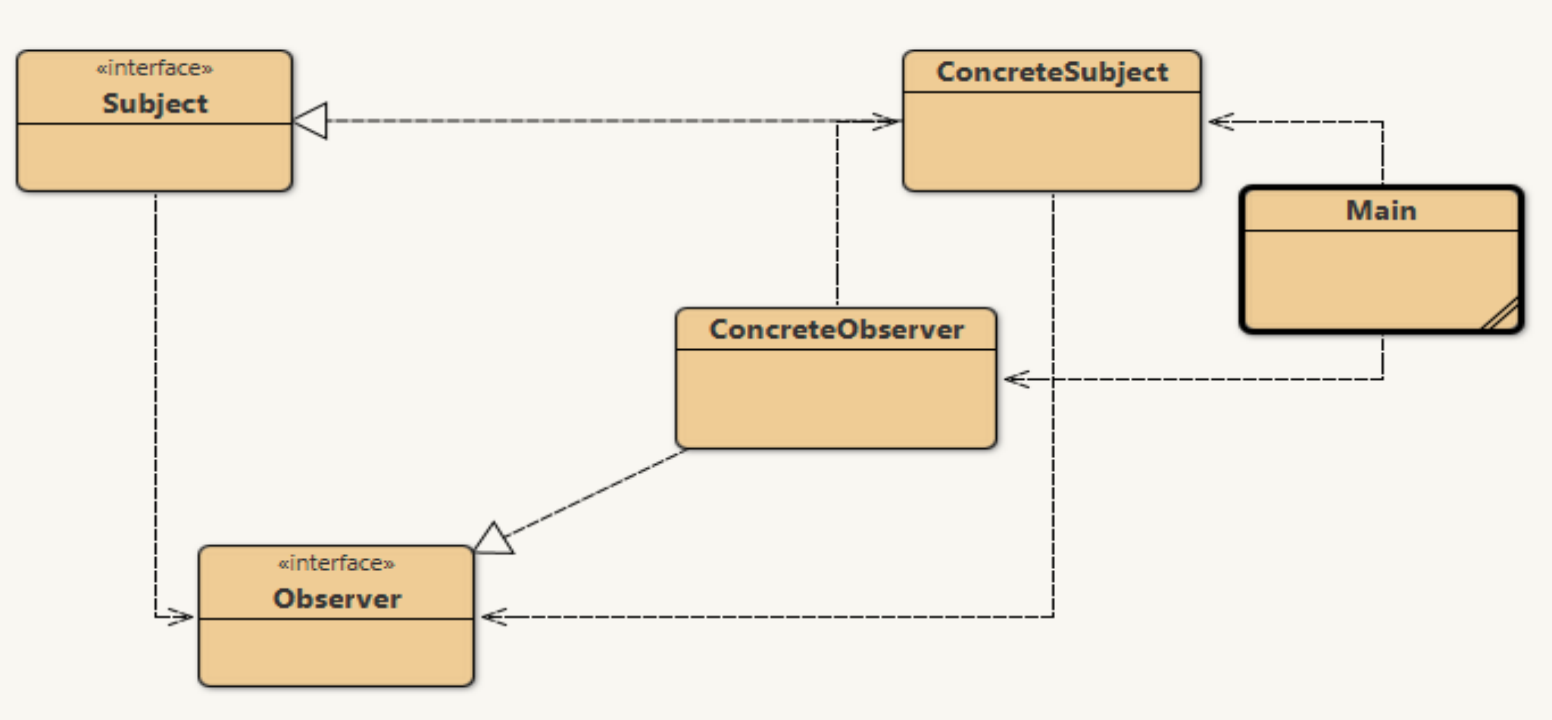
```
CREATE DATABASE design_patterns;

USE design_patterns;

CREATE TABLE StateTable (
    id INT AUTO_INCREMENT PRIMARY KEY,
    state INT NOT NULL
);
```

El patrón de diseño Observer define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Observer



interfaces

Define los métodos para agregar, eliminar y notificar observadores.

```
public interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}
```

Define el método que será llamado cuando el sujeto notifique cambios.

```
public interface Observer {
    void update();
}
```

ConcreteObserver

Implementa la interfaz Observer y actualiza su estado cuando es notificado por el sujeto.

```
public class ConcreteObserver implements Observer {
    private ConcreteSubject subject;

    public ConcreteObserver(ConcreteSubject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("Observador notificado. Nuevo Estado: " + subject.getState());
    }
}
```

ConcreteSubject

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.ArrayList;
import java.util.List;

public class ConcreteSubject implements Subject {
   private List<Observer> observers = new ArrayList<>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      saveStateToDatabase(state);
      notifyObservers();
   }
```

Implementa la interfaz Subject, mantiene una lista de observadores y un estado. Cuando el estado cambia, se guarda en la base de datos y se notifican a los observadores.

```
@Override
  public void attach(Observer o) {
     observers.add(o);
  }

  @Override
  public void detach(Observer o) {
     observers.remove(o);
  }

  @Override
  public void notifyObservers() {
     for (Observer o : observers) {
        o.update();
     }
  }
```

ConcreteSubject

ConcreteSubject

```
    private void saveStateToDatabase(int state) {
        try (Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/design_patterns", "root", " ")) {
            String query = "INSERT INTO StateTable (state) VALUES (?)";
            PreparedStatement preparedStatement = connection.prepareStatement(query);
            preparedStatement.setInt(1, state);
            preparedStatement.executeUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Main

Prueba la implementación del patrón Observer. Crea instancias del sujeto y observadores, cambia el estado del sujeto y observa las notificaciones y los cambios en la base de datos.

```
public class Main {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();

        ConcreteObserver observer1 = new ConcreteObserver(subject);
        ConcreteObserver observer2 = new ConcreteObserver(subject);

        System.out.println("Primer cambio de estado: 15");
        subject.setState(15);

        System.out.println("Segundo cambio de estado: 10");
        subject.setState(10);
    }
}
```

Explicación

Objetivo

Implementar el patrón Observer en Java, donde los cambios en el estado de un sujeto se guardan en una base de datos y se notifica a los observadores.

Componentes del Patrón Observer

1. **Subject (Sujeto):** Define la interfaz para registrar, eliminar y notificar observadores.
2. **ConcreteSubject (Sujeto Concreto):** Mantiene el estado y notifica a los observadores cuando hay un cambio de estado.
3. **Observer (Observador):** Define una interfaz para ser notificado de cambios en el sujeto.
4. **ConcreteObserver (Observador Concreto):** Implementa la interfaz de observador y actualiza su estado cuando es notificado.

Explicación

Estructura de la Base de Datos

Usaremos una base de datos MySQL para almacenar los estados. La tabla tendrá la siguiente estructura:

sql — Copiar código

```sql
CREATE DATABASE design_patterns;

USE design_patterns;

CREATE TABLE StateTable (
    id INT AUTO_INCREMENT PRIMARY KEY,
    state INT NOT NULL
);
```

Explicación

Flujo del Programa

1. **Creación del Sujeto y Observadores:**
   - Se crea una instancia de `ConcreteSubject`.
   - Se crean instancias de `ConcreteObserver` y se asocian con el `ConcreteSubject`.
2. **Cambio de Estado del Sujeto:**
   - Se cambia el estado del `ConcreteSubject` usando el método `setState(int state)`.
   - El nuevo estado se guarda en la base de datos mediante el método `saveStateToDatabase(int state)`.
   - Se notifican a todos los observadores registrados llamando al método `notifyObservers()`.
3. **Actualización de Observadores:**
   - Cada observador ejecuta su método `update()`, que imprime el nuevo estado en la consola.

Explicación

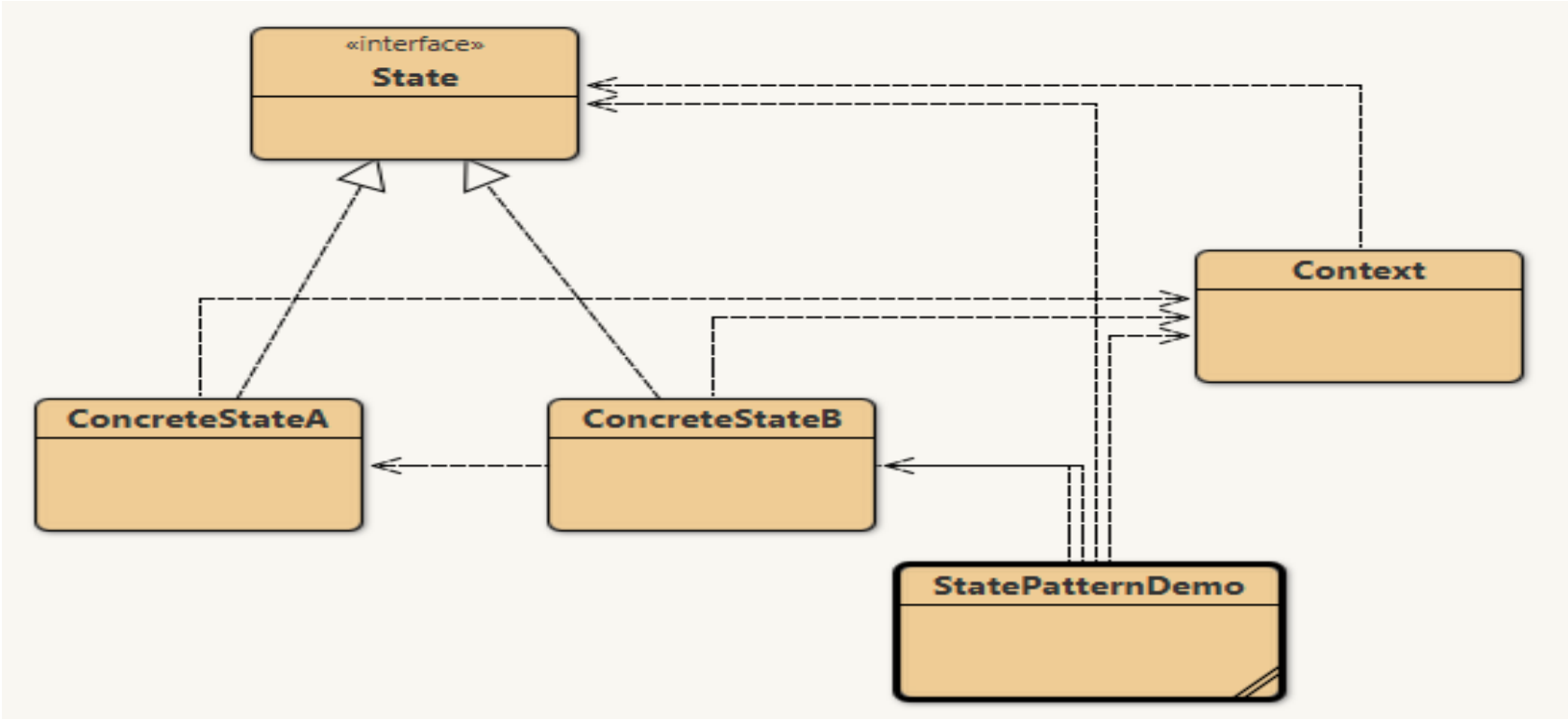
- Esta implementación demuestra cómo integrar el patrón Observer con una base de datos en Java.
- Cada vez que el estado del sujeto cambia, se guarda en la base de datos y se notifica a todos los observadores, quienes actualizan su estado en consecuencia.
- Este patrón es útil para aplicaciones que necesitan mantener a múltiples componentes sincronizados con los cambios de estado de un objeto central.

State Pattern

El patrón de diseño State permite que un objeto altere su comportamiento cuando su estado interno cambia. El objeto parecerá cambiar su clase.

En este ejercicio, implementaremos el patrón State donde el estado del contexto se guarda en una base de datos.

State Pattern



Interface State

Define el método que cambiará el comportamiento del contexto.

```
public interface State {
    void doAction(Context context);
}
```

ConcreteSateA

Implementa el comportamiento asociado con el estado A.

```
public class ConcreteStateA implements State {
    @Override
    public void doAction(Context context) {
        System.out.println("El Estado A está actuando.");
        context.setState(this);
    }

    public String toString(){
        return "State A";
    }
}
```

ConcreteSateB

Implementa el comportamiento asociado con el estado B.

```
public class ConcreteStateB implements State {
    @Override
    public void doAction(Context context) {
        System.out.println("El Estado B está actuando..");
        context.setState(this);
    }

    public String toString(){
        return "Estado B";
    }
}
```

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

public class Context {
    private State state;

    public Context() {
        state = null;
    }

    public void setState(State state) {
        this.state = state;
        saveStateToDatabase(state.toString());
    }

    public State getState() {
        return state;
    }
```

Clase Context

Clase Context

Mantiene el estado actual y cambia su comportamiento en función de su estado. También guarda el estado en la base de datos.

```
 private void saveStateToDatabase(String state) {
       try (Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/design_patterns", "root", " ")) {
          String query = "INSERT INTO StateTable (state) VALUES (?)";
          PreparedStatement preparedStatement = connection.prepareStatement(query);
          preparedStatement.setString(1, state);
          preparedStatement.executeUpdate();
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```

StatePatternDemo

Prueba la implementación del patrón State. Crea un contexto y cambia su estado, observando el cambio de comportamiento y el registro del estado en la base de datos.

```
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      State stateA = new ConcreteStateA();
      stateA.doAction(context);

      System.out.println(context.getState().toString());

      State stateB = new ConcreteStateB();
      stateB.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

Explicación

Objetivo

Implementar el patrón State en Java, donde el estado de un contexto se guarda en una base de datos y el comportamiento del contexto cambia en función de su estado actual.

Componentes del Patrón State

1. State (Estado): Define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto.
2. **ConcreteStateA y ConcreteStateB (Estado Concreto):** Implementan la interfaz `State` y definen el comportamiento específico para un estado particular del contexto.
3. **Context (Contexto):** Mantiene una instancia de una subclase concreta de `State` que define el estado actual y permite cambiar el estado.

Explicación

Flujo del Programa

1. **Creación del Contexto y Estados:**
   - Se crea una instancia de `Context`.
   - Se crean instancias de `ConcreteStateA` y `ConcreteStateB`.
2. **Cambio de Estado del Contexto:**
   - Se cambia el estado del contexto llamando al método `doAction(Context context)` de los estados.
   - El nuevo estado se guarda en la base de datos mediante el método `saveStateToDatabase(String state)` del `Context`.
3. **Cambio de Comportamiento del Contexto:**
   - El comportamiento del contexto cambia en función del estado actual, que se refleja en la salida del programa.

Explicación

- Esta implementación demuestra cómo integrar el patrón State con una base de datos en Java.
- Cada vez que el estado del contexto cambia, se guarda en la base de datos y el comportamiento del contexto se ajusta en función del estado actual.
- Este patrón es útil para aplicaciones donde un objeto debe cambiar su comportamiento dinámicamente en función de su estado interno, y donde es importante mantener un historial de estados en una base de datos.

Explicación de ambos ejemplos

- Estos ejemplos muestran cómo implementar los patrones de diseño Observer y State en Java con la integración de una base de datos. Los estados se guardan en la base de datos usando JDBC.
- Asegúrate de tener una base de datos MySQL configurada y de que el driver JDBC esté incluido en tu proyecto.

Practica

Implementa los ejercicios vistos.

Conclusiones:

- Aplicar estos patrones de diseño en proyectos Java con bases de datos y programación orientada a objetos mejora la estructura, mantenibilidad y flexibilidad del software, permitiendo a los desarrolladores crear sistemas más robustos y escalables.

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