**Universidad de las Fuerzas Armadas - ESPE**

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

Análisis y Diseño de Software - NRC:23305

**Tema:**

**Grupo: 3**

**Integrantes:**

Ronny Ibarra
Carlos Rivera
Angelo Sanchez

**Profesora**: Ing. Jenny Ruiz

***Taller 1 Patron***

***Link:***

***https://onlinegdb.com/dt7Nbq3E8***

| ***RÚBRICA PARA EVALUACIÒN*** | |
|---|---|

| ***Preguntas*** | ***Puntos*** | ***Calificación*** | ***Observación*** |
|---|---|---|---|
| 1. La clase main, llama a la vista (View), y al controlador (Controler) | 1 | | |
| 2. Elaborar el modelo en base de datos (BD) al dado, únicamente se adaptando, añadiendo tres nuevos Constructores | 1 | | |
| 3. Para la Vista view crear un método de inserción, el cual simula cómo sería la inserción tradicional | 1 | | |
| 4. El controlador se cambió en su mayoría, haciendo uso únicamente de los métodos que se requieren para hacer un intermediario entre el modelo, y la vista. | 1 | | |
| 5. Crear una clase que simula una base de datos, la cual brinda apoyo en la administración de los datos | 1 | | |

| | | | |
|---|---|---|---|
| quemados. (05 estudiantes). | | | |
| EJECUCIÓN | | | |
| **TOTAL** | **5** | **/5** | |

**REQUISITOS:**

**Para cada RF realizar la revisión de código y explicar a través de la ejecución el funcionamiento de MVC**

**Revisión del Código y Funcionamiento de MVC**

El patrón MVC (Modelo-Vista-Controlador) fue aplicado correctamente en este proyecto. Se identifican tres componentes claros:

- Modelo (Model): Representado por las clases Student y StudentDatabase.
- Vista (View): Representada por la clase StudentView.
- Controlador (Controller): Implementado en la clase StudentController.

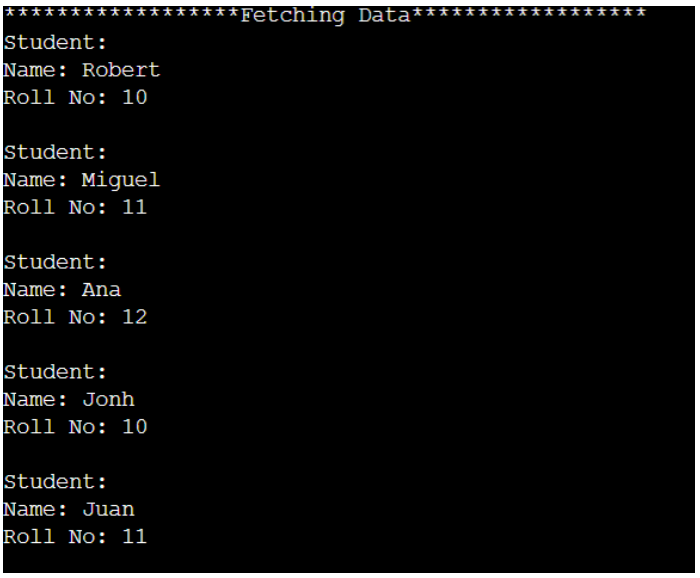
**1.- La clase main, llama al View, y al controlador**

**Clase Main**

La clase Main cumple con su función principal: iniciar la aplicación conectando la Vista con el Controlador. No contiene lógica adicional innecesaria, simplemente crea una instancia del StudentView, luego del StudentController, y finalmente llama al método start() para comenzar el flujo de trabajo.

```java
/******************************************************************************

Welcome to GDB Online.
GDB online is an online compiler and debugger tool for C, C++, Python, Java, PHP, Ruby, Perl,
C#, VB, Swift, Pascal, Fortran, Haskell, Objective-C, Assembly, HTML, CSS, JS, SQLite, Prolog.
Code, Compile, Run and Debug online from anywhere in world.

*******************************************************************************/

public class Main {

    public static void main(String[] args) {
        StudentView view = new StudentView();
        StudentController controller = new StudentController(view);
        controller.start();
    }
}

```

**Ejecución:**




    - Se crea la instancia de StudentView (cliente) y StudentController (intermediario).
    - El controlador inicia el flujo con el método start().
- **Patrón Cliente-Servidor:**
    - La clase main inicia con el cliente (vista) y el controlador (que es el intermediario), simulando el inicio de comunicación.

**2. Se hizo el modelo en base al dado, únicamente adaptando, añadiendo tres nuevos constructores**

El modelo se basó en la clase Student proporcionada originalmente, pero se adaptó para mejorar su funcionalidad y flexibilidad al añadir tres constructores diferentes:

- **Constructor vacío:** Permite crear un objeto Student sin inicializar sus atributos, facilitando la creación de instancias cuando no se tiene información inmediata.
- **Constructor con parámetros (nombre y número de matrícula):** Facilita la creación directa de objetos Student con los datos necesarios desde el momento de la instanciación.

- **Constructor por copia:** Permite crear un nuevo objeto Student duplicando los atributos de otro objeto existente. Esto es útil para operaciones como actualizar o modificar estudiantes sin alterar el original hasta que se confirme el cambio.

```java
public class Student {

    private String rollNo;
    private String name;

    public Student() {
        this("", "");
    }

    public Student(String name, String rollNo) {
        this.rollNo = rollNo;
        this.name = name;
    }

    public Student(Student student) {
        this.rollNo = student.getRollNo();
        this.name = student.getRollNo();
    }

    public String getRollNo() {
        return rollNo;
    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
```

```java
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

    - La clase Student con constructores variados se usa para crear objetos en toda la ejecución.

- **Patrón Cliente-Servidor:**
    - Modelo en el servidor que contiene la estructura de datos y lógica, representando los “datos” que el servidor maneja.

**3. Para el view se creó un método de inserción, el cual simula cómo sería la inserción tradicional**

En la clase StudentView se creó un método llamado inputStudent() que simula la inserción tradicional de datos. Este método no realiza una entrada real desde el usuario, sino que muestra por consola cómo sería el proceso típico de ingresar datos, mostrando los valores que se "insertarían" (por ejemplo, nombre y número de matrícula).

Esta simulación permite visualizar el flujo de entrada de información en la interfaz de usuario sin necesidad de interacción real, facilitando la comprensión del proceso de creación de un nuevo estudiante en el sistema.

```
public class StudentView {

    public void printStudentDetails(Student student) {
        System.out.println("Student: ");
        System.out.println("Name: " + student.getName());
        System.out.println("Roll No: " + student.getRollNo());
        System.out.println("");
    }

    public Student inputStudent() {
        String name = "David";
        String rollNo = "1";
        System.out.println("***Create: ");
        System.out.println("Student: ");
        System.out.println("Name: ");
        System.out.println("Input: " + name);
        System.out.println("Roll No: " + rollNo);
        System.out.println("Input: " + rollNo);
        System.out.println("***End: ");
        System.out.println("");

        return new Student(name, rollNo);
    }
}
```

- En StudentView.inputStudent(), el cliente simula enviar datos al servidor (base de datos).

- **Patrón Cliente-Servidor:**
  - Vista actúa como cliente que “envía” datos para crear un nuevo estudiante.

**4.- El controlador se cambió en su mayoría, haciendo uso únicamente de los métodos que se requieren para hacer un intermediario entre el modelo, y la vista**

La clase StudentController fue modificada en gran parte para simplificar su rol y enfocarse únicamente en las funciones necesarias para mediar entre el modelo y la vista.

**El controlador:**

- Utiliza solo los métodos imprescindibles para gestionar las operaciones CRUD (crear, leer, actualizar, eliminar) sobre los datos.
- Recibe las solicitudes o acciones de la vista, las procesa y realiza llamadas al modelo para manipular los datos.
- Luego, actualiza la vista con la información resultante, manteniendo la sincronía entre ambos.

```java
import java.util.List;

public class StudentController {

    private StudentDatabase database;
    private StudentView view;

    public StudentController(StudentView view) {
        this.view = view;
        this.database = StudentDatabase.getInstance();
    }

    public void start() {
        System.out.println("**********Fetching Data**********");
        this.fetchStudents();
        System.out.println("**********Creating Data**********");
        this.createStudent();
        System.out.println("**********Updating Data**********");
        this.updateStudent();
    }

    public void fetchStudents() {
        this.updateView();
    }
```

- **Ejecución:**
    - StudentController recibe las “peticiones” del cliente (vista), las procesa, llama al modelo (base de datos) y actualiza la vista con los resultados.

- **Patrón Cliente-Servidor:**
    - Controlador funciona como middleware o servidor de aplicaciones que procesa solicitudes del cliente y responde.

**5. Se creó una clase que simula una base de datos, la cual brinda apoyo en la administración de los datos quemados**

Se desarrolló la clase StudentDatabase para simular una base de datos en memoria que gestiona los datos "quemados" (predefinidos) de estudiantes. Esta clase:

- Mantiene una lista interna donde se almacenan objetos Student.
- Proporciona métodos para realizar operaciones básicas: agregar, obtener, actualizar y eliminar estudiantes.
- Implementa el patrón Singleton para asegurar que solo exista una única instancia accesible a lo largo de toda la aplicación, garantizando la coherencia y consistencia de los datos.

```java
import java.util.List;
import java.util.ArrayList;

public class StudentDatabase {

    private static StudentDatabase instance;
    private List<Student> students;

    private StudentDatabase() {
        this.students = new ArrayList<>();
        this.students.add(new Student("Robert", "10"));
        this.students.add(new Student("Miguel", "11"));
        this.students.add(new Student("Ana", "12"));
        this.students.add(new Student("Jonh", "10"));
        this.students.add(new Student("Juan", "11"));
    }

    public static StudentDatabase getInstance() {
        if(instance == null) {
            instance =  new StudentDatabase();
        }
        return instance;
    }
```

**Ejecución:**

    - StudentDatabase simula el servidor de datos, almacenando y manipulando la información.
- **Patrón Cliente-Servidor:**
    - Es el servidor real que recibe las solicitudes del cliente (vista) mediadas por el controlador.

## CONCLUSIÓN

### ¿Cómo se comunican estas tres capas?

Como equipo, concluimos que el uso del patrón MVC en el sistema permite una clara separación de responsabilidades, lo que facilita el mantenimiento y la escalabilidad futura. Al centralizar la lógica en el Controlador, que actúa como intermediario entre el Modelo y la Vista, garantizamos que los datos se gestionen correctamente sin que la presentación o la interacción del usuario afecten directamente al manejo interno. Esta estructura también mejora la modularidad y refleja eficazmente un modelo cliente-servidor, donde cada componente cumple un rol definido y complementario para el correcto funcionamiento del sistema.