Universidad de las Fuerzas Armadas - ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

Análisis y Diseño de Software - NRC:23305

Taller 1 Patrón MVC
Grupo: 1
Integrantes:

- ***Erick Moreira***
- ***Carlos Granda***
- ***Kevin Coloma***

Docente:
ng. 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. Finalmente, crear una clase que simula una base de datos, la cual brinda apoyo en la administración de los datos quemados. (05 estudiantes).	1		
EJECUCIÓN			
TOTAL	**5**	**/5**	

REQUISITOS:

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

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

Sí, la clase Main crea instancias de StudentView (la vista) y StudentController (el controlador). Al crear el controlador, se le pasa la vista como argumento, lo que posibilita la interacción con la vista y la gestión de las operaciones entre el modelo y la vista. Aquí el flujo de las ejecuciones muestra cómo se crean, actualizan y visualizan los datos de los estudiantes, con el controlador gestionando las interacciones y la vista encargada de mostrarlas al usuario.

```
public class Main {

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

Ejecución :

```
**********Fetching Data**********
Student:
Name: Erick
Roll No: 22
```

El controlador está llamando a la vista para mostrar datos.

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

Los tres constructores que se añadieron fueron los siguientes, primero un constructor vacío llamado Student() que inicializa los atributos como cadenas vacías.

```
6       public Student() {
7           this("", "");
8       }
```

El segundo constructor permite asignar directamente los datos a los atributos correspondientes de los estudiantes.

```
10      public Student(String name, String rollNo) {
11          this.rollNo = rollNo;
12          this.name = name;
13      }
```

Y el tercer constructor crea un nuevo objeto de tipo Student para duplicar los datos de algún estudiante.

```
15      public Student(Student student) {
16          this.rollNo = student.getRollNo();
17          this.name = student.getRollNo();
18      }
```

En la simulación de la base de datos podemos ver que hacemos uso del constructor añadiendo datos tanto para el name, como para el rollNo que en este caso es “Robert” y “10”.

```
4  public class StudentDatabase {
5
6      private static StudentDatabase instance;
7      private List<Student> students;
8
9      private StudentDatabase() {
10         this.students = new ArrayList<>();
11         this.students.add(new Student("Erick", "22"));
```

En la ejecución podemos ver la creación de este estudiante

```
**********Fetching Data**********
Student:
Name: Erick
Roll No: 22
```

Instancias creadas con el modelo Student.

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

La clase StudentView se encarga de la interfaz con el usuario.
Tiene dos funciones principales:

● Mostrar detalles: El método printStudentDetails(Student student) imprime en consola el nombre y número de matrícula del estudiante recibido.

● Simulación de entrada de datos: El método inputStudent() simula la inserción tradicional de un estudiante por el usuario, pero con valores hardcodeados (en este caso, el estudiante "Erick" con matrícula "22").

```java
import java.util.Scanner;

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);
    }
}
```

Sí, el método inputStudent() simula la inserción de un estudiante con datos predeterminados. Debido a que este método utiliza el segundo constructor de la clase Student.java, y es llamado en la creación del StudentController.java, la ejecución de este se puede observar

No se modifican datos ni se hace lógica de negocio aquí, solo se muestra y captura.

```java
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);
}
```

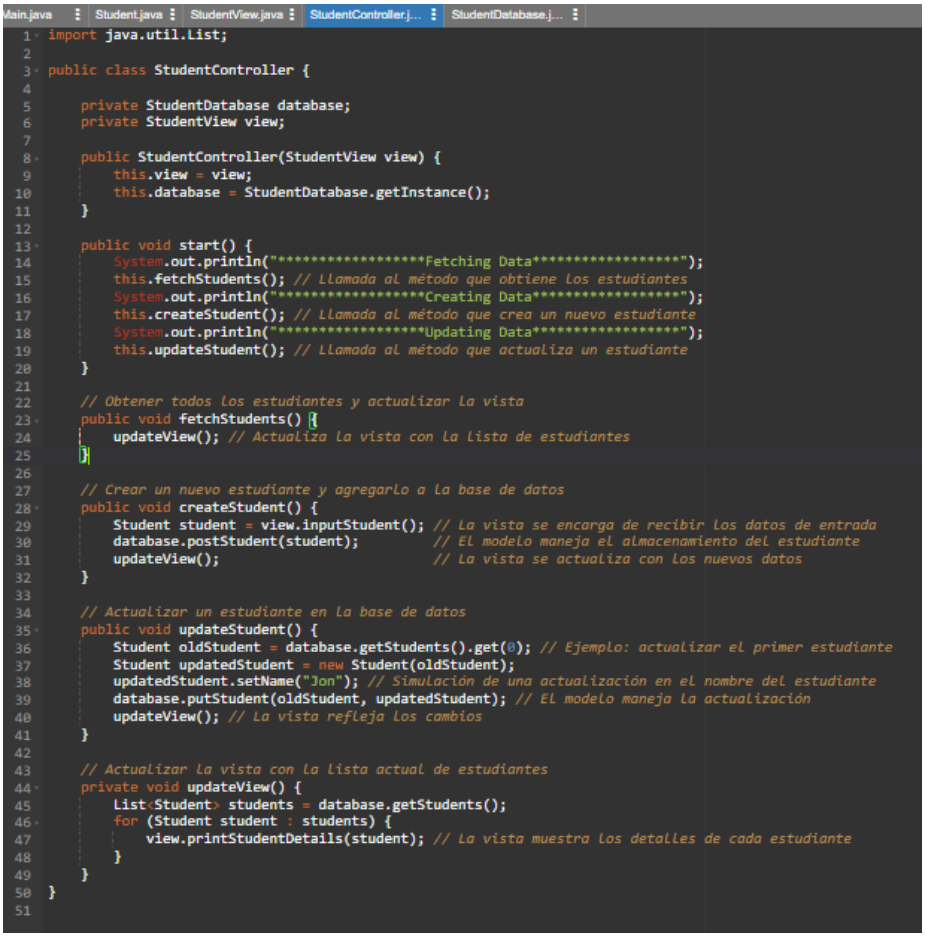
Ejecución

```
***Create:
Student:
Name:
Input: Erick
Roll No: 22
Input: 22
***End:
```

Aquí se ve claramente el método inputStudent() que simula la inserción tradicional mostrando qué valores se “ingresan” antes de crear el objeto.

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

Sí, el StudentController se ha simplificado para actuar exclusivamente como un intermediario entre el modelo (StudentDatabase) y la vista (StudentView), coordinando acciones como obtener estudiantes, crear y actualizar registros sin involucrarse en la lógica de negocio compleja. Delega la entrada de datos a la vista y luego usa el modelo para almacenar o actualizar los datos, manteniendo así una clara separación de responsabilidades y mejorando la claridad y el mantenimiento del código. Esto asegura que cada componente tenga un único propósito y facilita futuras modificaciones, permitiendo realizar cambios sin afectar otras partes del código, lo que mejora la escalabilidad y el mantenimiento a largo plazo.



El StudentController ha sido adaptado para funcionar estrictamente como intermediario entre el modelo (Student y StudentDatabase) y la vista (StudentView).

Sus responsabilidades principales son:
- Obtener datos del modelo o base de datos.
- Solicitar a la vista que muestre dichos datos.
- Recibir nuevos datos (como el estudiante insertado desde la vista)
- Actualizar el modelo en consecuencia.
- Mantener la lógica y el flujo de la aplicación centralizados en este componente.

Ejecución

```
******************Updating Data*******
Student:
Name: Jon
Roll No: 10

Student:
Name: Miguel
Roll No: 11

Student:
Name: Ana
Roll No: 12

Student:
Name: Jonh
Roll No: 10

Student:
Name: Juan
Roll No: 11

Student:
Name: Erick
Roll No: 22



...Program finished with exit code 0
```

Esta salida muestra que el controlador recibió el estudiante creado por la vista y actualizó la base de datos (modelo)

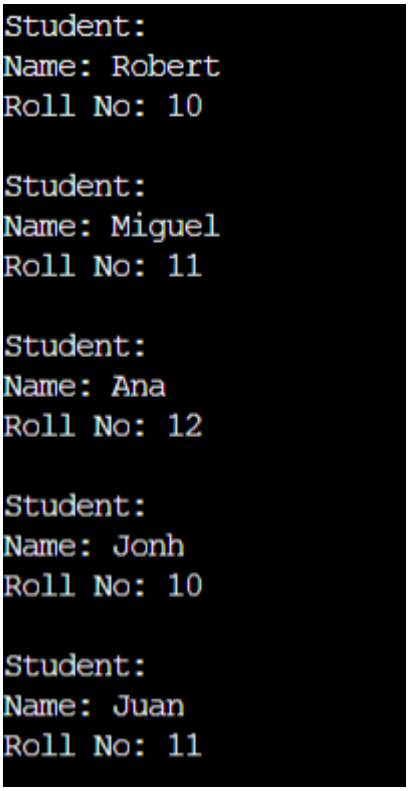
5. Se creó una clase que simula una base de datos, la cual brinda apoyo en la administración de los datos quemados.

La clase que sirvió para realizar la simulación de la base de datos fue StudentDataBase donde tenemos una lista de tipo Student con datos quemados.

```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"));
    }
```

● Implementa el patrón Singleton, garantizando una única instancia compartida durante la ejecución.
● Mantiene una lista interna (List) con datos quemados (precargados) para pruebas.
● Provee métodos para agregar (postStudent), modificar (putStudent) y eliminar (deleteStudent) estudiantes.
● Facilita la gestión centralizada y persistente de los datos mientras la aplicación está activa.

Y en la ejecución se puede observar estos datos

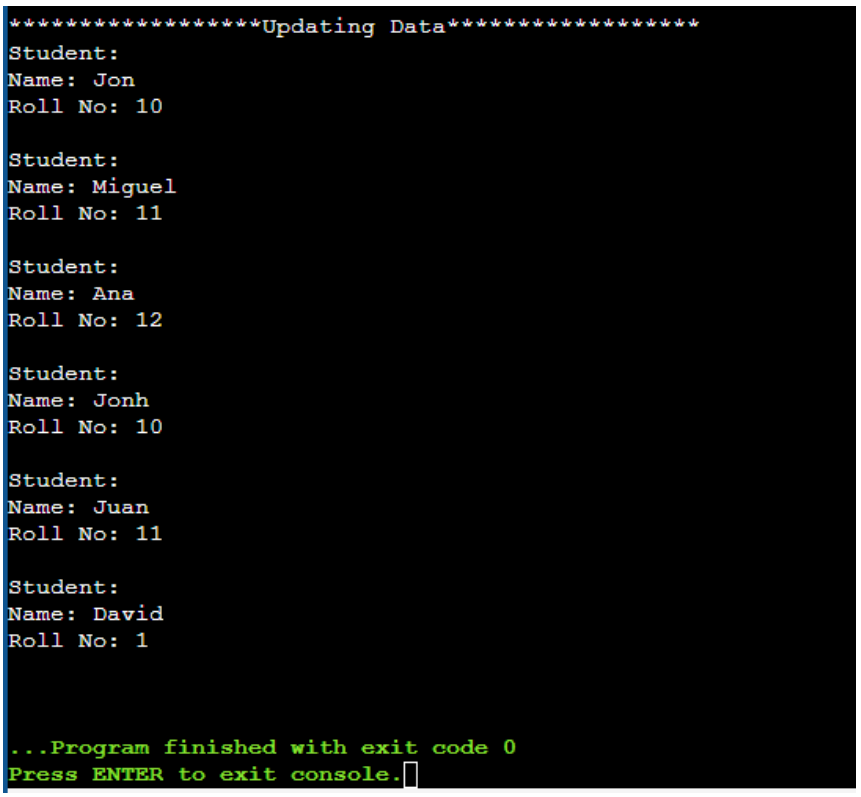


La lista inicial de estudiantes que aparece primero está precargada en la base de datos simulada (StudentDatabase), que tiene estos datos “quemados”

Ejecucion del codigo/ PantallasLa lista inicial de estudiantes que aparece primero está precargada en la base de datos simulada (StudentDatabase), que tiene estos datos “quemados”

```
******************Fetching Data******************
Student:
Name: Robert
Roll No: 10

Student:
Name: Miguel
Roll No: 11

Student:
Name: Ana
Roll No: 12

Student:
Name: Jonh
Roll No: 10

Student:
Name: Juan
Roll No: 11
```

```
******************Creating Data******************
***Create:
Student:
Name:
Input: David
Roll No: 1
Input: 1
***End:

Student:
Name: Robert
Roll No: 10

Student:
Name: Miguel
Roll No: 11

Student:
Name: Ana
Roll No: 12

Student:
Name: Jonh
Roll No: 10

Student:
Name: Juan
Roll No: 11

Student:
Name: David
Roll No: 1
```



La lista inicial de estudiantes que aparece primero precargada en la base de datos simulada se actualiza mediante el controlador donde entra en juego la capa modelo con (StudentDatabase)

Conclusión

En este programa, la comunicación entre las tres capas del Modelo-Vista-Controlador (MVC) se estructura de la siguiente manera: el Controlador juega un papel crucial como intermediario central, facilitando la interacción entre el Modelo y la Vista. Al iniciar el programa, el Controlador solicita los datos al Modelo, que contiene la lógica y el almacenamiento, representado en este caso por una base de datos simulada. Posteriormente, el Controlador transmite estos datos a la Vista para su presentación al usuario. Para la creación o modificación de datos, la Vista simula la entrada de información y envía un objeto modelo al Controlador. Este último procesa la información y actualiza el Modelo, asegurando la coherencia de los datos. Finalmente, el Controlador obtiene la información actualizada del Modelo y la transmite de nuevo a la Vista para su visualización.