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: 1

Integrantes:

David Asmal

Mateo Llumigusin

Richard Gualotuña

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 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**

```
public class Main {

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

La clase Main se instancian los componentes principales que conforman el patrón de diseño MVC (Modelo-Vista-Controlador):

- Se crea una instancia de la vista (StudentView), encargada de la interacción con el usuario (mostrar datos y recibir entradas).
- Se crea una instancia del controlador (StudentController), que recibe la vista como parámetro y es responsable de coordinar la lógica de la aplicación.
- Finalmente, se llama al método start() del controlador para iniciar la ejecución y control del flujo.

Ejecución

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

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**

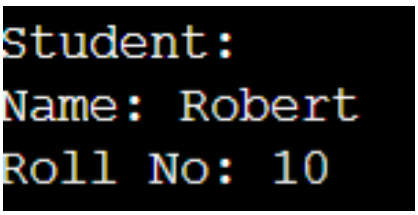
```java
public class Student {

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

La clase Student representa el modelo de datos, encapsulando la información básica de un estudiante: nombre (name) y número de matrícula (rollNo).

Para flexibilizar la creación de objetos Student, se implementaron tres constructores:

- **Constructor vacío:** Permite crear un estudiante con valores por defecto (cadenas vacías).
- **Constructor con parámetros:** Permite crear un estudiante con datos específicos para nombre y matrícula.
- **Constructor copia:** Crea un nuevo objeto Student a partir de otro, copiando sus atributos. Esto es útil para evitar modificaciones no deseadas sobre objetos originales.

Ejecución



Instancias creadas con el modelo Student

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

```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);
    }
}
```

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 "David" con matrícula "1").

Ejecución

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

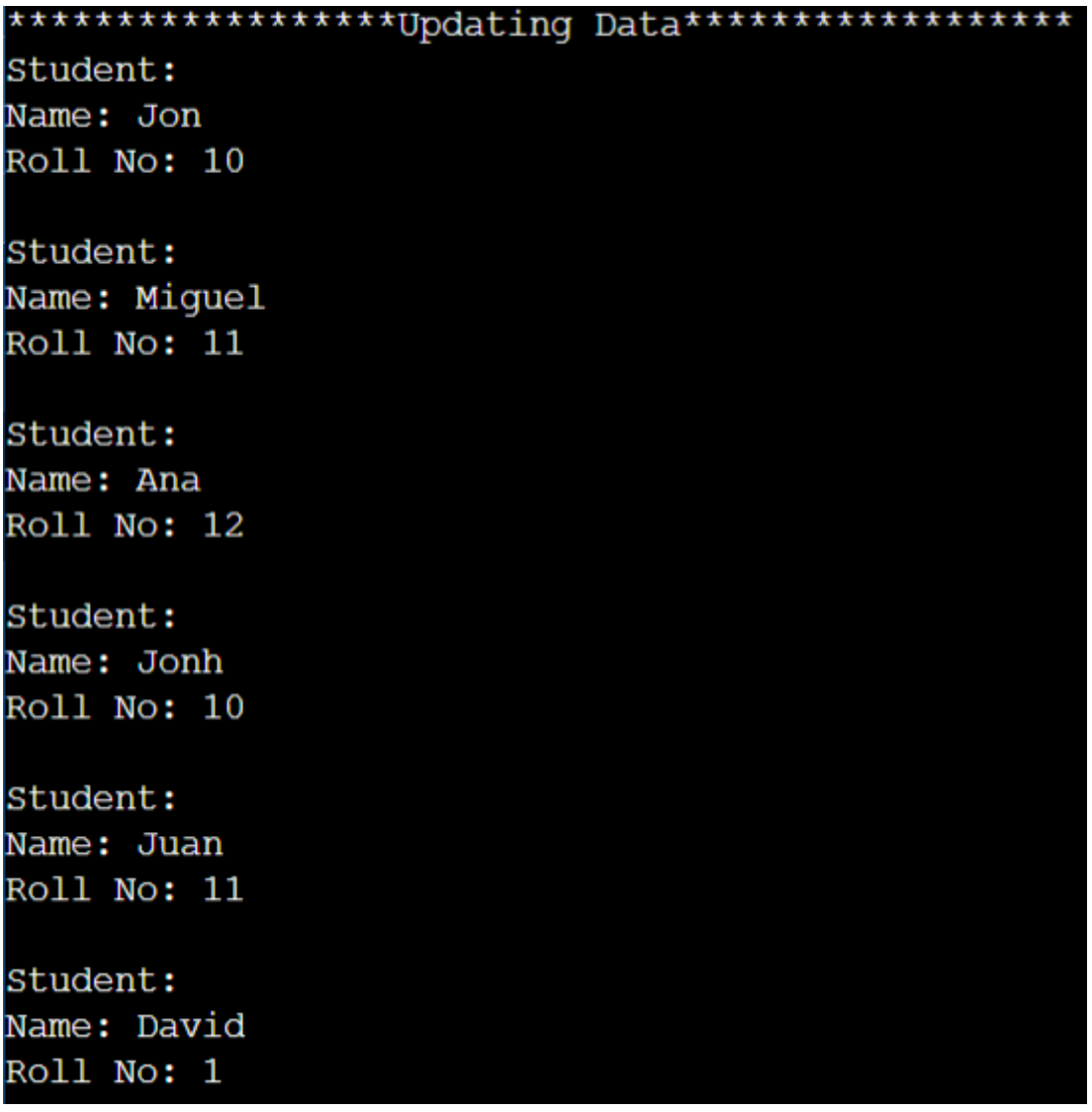
```java
import java.util.List;

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();
    }

    public void createStudent() {
        Student student = this.view.inputStudent();
        this.database.postStudent(student);
        this.updateView();
    }

    public void updateStudent() {
        Student oldStudent = this.database.getStudents().get(0);
        Student studentUpdate = new Student(oldStudent);
        studentUpdate.setName("Jon");

        this.database.putStudent(oldStudent, studentUpdate);
        this.updateView();
    }

    private void updateView() {
        List<Student> data = this.database.getStudents();
        for(Student student: data) {
            this.view.printStudentDetails(student);
        }
    }
```

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



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

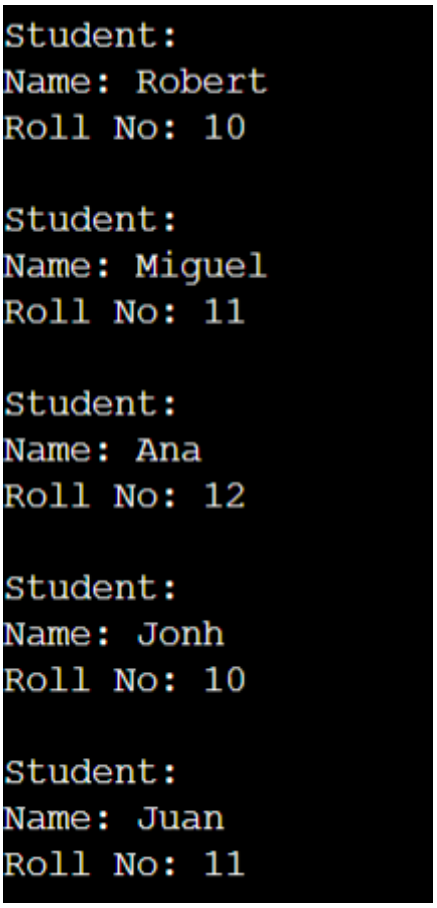
5. **Se creó una clase que simula una base de datos, la cual brinda apoyo en la administración de los datos quemados.**

```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;
    }


    public List<Student> getStudents() {
        return this.students;
    }

    public void postStudent(Student student) {
        this.students.add(student);
    }

    public void putStudent(Student oldStudent, Student updatedStudent) {
        int index = this.students.indexOf(oldStudent);
        this.students.remove(index);
        this.students.add(0, updatedStudent);
    }

    public void deleteStudent(Student student) {
        this.students.remove(student);
    }
}
```

Para administrar los estudiantes y simular una base de datos en memoria, se creó la clase StudentDatabase que cumple con las siguientes funciones:

- Implementa el patrón Singleton, garantizando una única instancia compartida durante la ejecución.

- Mantiene una lista interna (List<Student>) 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.



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

Conclusión

En este programa, la comunicación entre las tres capas Modelo, Vista y Controlado (MVC). El Controlador actúa como intermediario central, coordinando la interacción entre el Modelo y la Vista.

Cuando se inicia el programa, el controlador solicita los datos al modelo, que contiene la lógica y almacenamiento (en este caso, una base de datos simulada). Luego, el controlador envía esos datos a la vista para que se presenten al usuario.

Para crear o modificar datos, la vista simula la entrada de información y entrega un objeto modelo al controlador. Este último procesa la información y actualiza el modelo, manteniendo la coherencia de los datos. Finalmente, el controlador vuelve a obtener la información actualizada del modelo y la pasa a la vista para mostrarla.