



Universidad de las Fuerzas Armadas - ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

Análisis y Diseño de Software - NRC:22426

Tema:

Grupo: 4

Integrantes:

Diego Casignia

Anthony Villarreal

Javier Ramos

Profesora: Ing. Jenny Ruiz

Taller 1 Patron

Link:

<https://onlinegdb.com/dt7Nbq3E8>

RÚBRICA PARA EVALUACIÓN	
--------------------------------	--

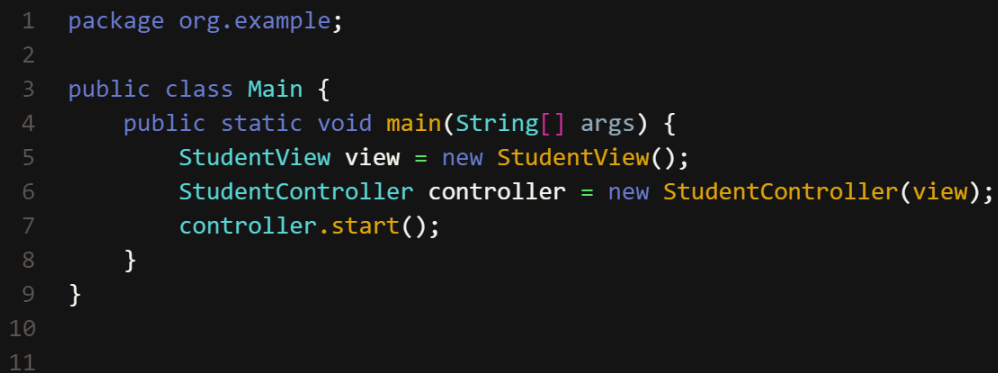
Preguntas	Puntos	Calificación	Observación
1. La clase main, llama a la vista (View), y al controlador (Controller)	1		
2. Elaborar el modelo en base de datos (BD) al dado, únicamente se adaptando, añadiendo tres nuevos Constructores	1		
3. Para la Vista view crear un método de inserción, el cual simula 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	1		

apoyo en la administración de los datos 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

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



```
1 package org.example;
2
3 public class Main {
4     public static void main(String[] args) {
5         StudentView view = new StudentView();
6         StudentController controller = new StudentController(view);
7         controller.start();
8     }
9 }
10
11
```

La clase Main crea una instancia de StudentView (la vista) y una instancia de StudentController (el controlador), pasando la vista como parámetro al controlador.

Se llama al método start() del controlador, iniciando la lógica del programa.

Esto sigue el patrón MVC, donde el Main actúa como el inicializador que conecta la vista y el controlador.

En esta parte del Main llama a la vista y al controlador y mostrando los datos quemados inicialmente

```
*****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
```

Luego se muestra la parte donde se añade un nuevo dato

```
*****Creating Data*****
***Create:
Student:
Name: javier
Roll No: 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: javier
Roll No: 1
```

Y luego donde se actualiza el primer dato que está en la lista.

```
*****Updating Data*****
Updating student: Robert
***Create:
Student:
Name: xavi
Roll No: 1
***End:

Student:
Name: xavi
Roll No: 1

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: javier
Roll No: 1
```

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

```
1 package org.example;
2
3 public class Student {
4     private String rollNo;
5     private String name;
6
7     public Student() {
8         this("", "");
9     }
10
11     public Student(String name, String rollNo) {
12         this.rollNo = rollNo;
13         this.name = name;
14     }
15
16     public Student(Student student) {
17         this.rollNo = student.getRollNo();
18         this.name = student.getName();
19     }
20
21     public Student(String name) {
22         this.name = name;
23         this.rollNo = "0";
24     }
25
26     public String getRollNo() {
27         return rollNo;
28     }
29
30     public void setRollNo(String rollNo) {
31         this.rollNo = rollNo;
32     }
33
34     public String getName() {
35         return name;
36     }
37
38     public void setName(String name) {
39         this.name = name;
40     }
41 }
```

El modelo Student tiene dos atributos: rollNo (número de matrícula) y name (nombre), con getters y setters correspondientes.

Incluye tres constructores:

Constructor por defecto: Student().

Constructor parametrizado: Student(String name, String rollNo).

Constructor de copia: Student(Student student).

Problema detectado: En el constructor de copia, hay un error donde `this.name = student.getRollNo()` debería ser `this.name = student.getName()`. Esto causa que el nombre del estudiante se establezca incorrectamente como el número de matrícula.

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

```
1 package org.example;
2
3 import java.util.Scanner;
4
5 public class StudentView {
6     private Scanner scanner = new Scanner(System.in);
7
8     public void printStudentDetails(Student student) {
9         System.out.println("Student: ");
10        System.out.println("Name: " + student.getName());
11        System.out.println("Roll No: " + student.getRollNo());
12        System.out.println("");
13    }
14
15    public Student inputStudent() {
16        System.out.println("***Create: ");
17        System.out.println("Student: ");
18        System.out.print("Name: ");
19        String name = scanner.nextLine();
20        System.out.print("Roll No: ");
21        String rollNo = scanner.nextLine();
22        System.out.println("***End: ");
23        System.out.println("");
24        return new Student(name, rollNo);
25    }
26 }
```

El método `inputStudent()` está diseñado para simular la inserción de un estudiante, pero utiliza valores quemados (`name = "David"`, `rollNo = "1"`) en lugar de solicitar entrada del usuario.

Una "inserción tradicional" implica típicamente que el usuario ingrese datos interactivamente, por ejemplo, a través de la consola usando `Scanner`.

El método imprime los valores como si fueran ingresados, pero no permite interacción real, lo cual no cumple completamente con el espíritu de una inserción tradicional.

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 package org.example;
2
3 import java.util.List;
4
5 public class StudentController {
6     private StudentDatabase database;
7     private StudentView view;
8
9     public StudentController(StudentView view) {
10         this.view = view;
11         this.database = StudentDatabase.getInstance();
12     }
13
14     public void start() {
15         System.out.println("*****Fetching Data*****");
16         this.fetchStudents();
17         System.out.println("*****Creating Data*****");
18         this.createStudent();
19         System.out.println("*****Updating Data*****");
20         this.updateStudent();
21     }
22
23     public void fetchStudents() {
24         this.updateView();
25     }
26
27     public void createStudent() {
28         Student student = this.view.inputStudent();
29         this.database.postStudent(student);
30         this.updateView();
31     }
32
33     public void updateStudent() {
34         List<Student> students = this.database.getStudents();
35         if (!students.isEmpty()) {
36             Student oldStudent = students.get(0);
37             System.out.println("Updating student: " + oldStudent.getName());
38             Student updatedStudent = this.view.inputStudent();
39             this.database.putStudent(oldStudent, updatedStudent);
40             this.updateView();
41         }
42     }
43
44     private void updateView() {
45         List<Student> data = this.database.getStudents();
46         for (Student student : data) {
47             this.view.printStudentDetails(student);
48         }
49     }
50 }
```

El controlador actúa como intermediario entre el modelo (StudentDatabase) y la vista (StudentView).

Métodos como fetchStudents(), createStudent(), y updateStudent() coordinan las operaciones:

fetchStudents(): Obtiene los datos del modelo y los muestra en la vista.

createStudent(): Solicita datos a la vista, los guarda en el modelo y actualiza la vista.

updateStudent(): Actualiza un estudiante en el modelo y muestra los cambios en la vista.

El controlador no realiza lógica de almacenamiento ni de presentación directamente, cumpliendo con su rol de intermediario.

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

```
1 package org.example;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class StudentDatabase {
7     private static StudentDatabase instance;
8     private List<Student> students;
9
10    private StudentDatabase() {
11        this.students = new ArrayList<>();
12        this.students.add(new Student("Robert", "10"));
13        this.students.add(new Student("Miguel", "11"));
14        this.students.add(new Student("Ana", "12"));
15        this.students.add(new Student("Jonh", "10"));
16        this.students.add(new Student("Juan", "11"));
17    }
18
19    public static StudentDatabase getInstance() {
20        if (instance == null) {
21            instance = new StudentDatabase();
22        }
23        return instance;
24    }
25
26    public List<Student> getStudents() {
27        return this.students;
28    }
29
30    public void postStudent(Student student) {
31        this.students.add(student);
32    }
33
34    public void putStudent(Student oldStudent, Student updatedStudent) {
35        int index = this.students.indexOf(oldStudent);
36        this.students.remove(index);
37        this.students.add(0, updatedStudent);
38    }
39
40    public void deleteStudent(Student student) {
41        this.students.remove(student);
42    }
43 }
```

La clase utiliza el patrón singleton para garantizar una única instancia de la base de datos.

Inicializa una lista con 5 estudiantes predefinidos, cumpliendo con el requisito de "datos quemados".

Proporciona métodos CRUD (getStudents(), postStudent(), putStudent(), deleteStudent()) para administrar los datos.

La implementación es adecuada para simular una base de datos simple en memoria.

Ejecución del código/ Pantallas

```
*****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: |
```

*****Creating Data*****

***Create:

Student:

Name: *Javier*

Roll No: *12*

***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: Javier

Roll No: 12

*****Updating Data*****

Updating student: Robert

***Create:

Student:

Name: |