**Universidad de las Fuerzas Armadas - ESPE**

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

Análisis y Diseño de Software - NRC:22426

**Trabajo:**

U2T3 - Proyecto Estudiantes

**Grupo:** 4

**Integrantes:**

Diego Casignia

Anthony Villarreal

Javier Ramos

**Profesora**: Ing. Jenny Ruiz

**Objetivo**

Desarrollar en un entorno IDE de Java una aplicación que implemente un CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar datos de estudiantes (orden, nombre y edad), utilizando el patrón de arquitectura en 3 capas. Se busca demostrar cómo interactúan las capas entre sí y cómo se ejecuta la lógica del sistema desde la clase principal (Main.java).

**Paso por paso de cómo se realizó**

1. **Capa Modelo (Entidad Estudiante):**

   Responsabilidad: Esta capa se encarga de representar la entidad de los estudiantes. Se creó la clase Estudiante que contiene los atributos esenciales como orden, nombre y edad, y métodos para obtener y modificar estos valores.

   Propósito: Proporcionar una estructura de datos para los estudiantes, que será utilizada en todo el sistema.

2. **Capa Repositorio (Persistencia):**

   Responsabilidad: La capa repositorio gestiona la persistencia de los datos. Se implementó la clase EstudianteRepositorio que simula una base de datos en memoria, donde se almacenan, actualizan, eliminan y recuperan los estudiantes.

   Propósito: Mantener los datos centralizados en una estructura controlada, en este caso una lista en memoria.

3. **Capa Servicio (Lógica de Negocio):**

   Responsabilidad: Esta capa implementa la lógica del negocio, gestionando las interacciones entre la interfaz de usuario y los datos. La clase EstudianteServicio maneja las operaciones CRUD (crear, leer, actualizar y eliminar) con validaciones de negocio, como verificar la existencia de un estudiante antes de actualizar o eliminar.

   Propósito: Separar la lógica de negocio de la de persistencia y presentación, asegurando que las reglas de negocio se implementan correctamente.

4. **Capa Presentación (Interfaz de Usuario):**

   Responsabilidad: La capa de presentación interactúa directamente con el usuario, mostrando los datos y recibiendo entradas. En este caso, la clase EstudiantesUI implementa una interfaz gráfica con Swing, donde el usuario puede ver, agregar, actualizar y eliminar estudiantes.

   Propósito: Brindar una forma de interacción directa con el sistema, manejando la entrada y salida de datos de manera visual.

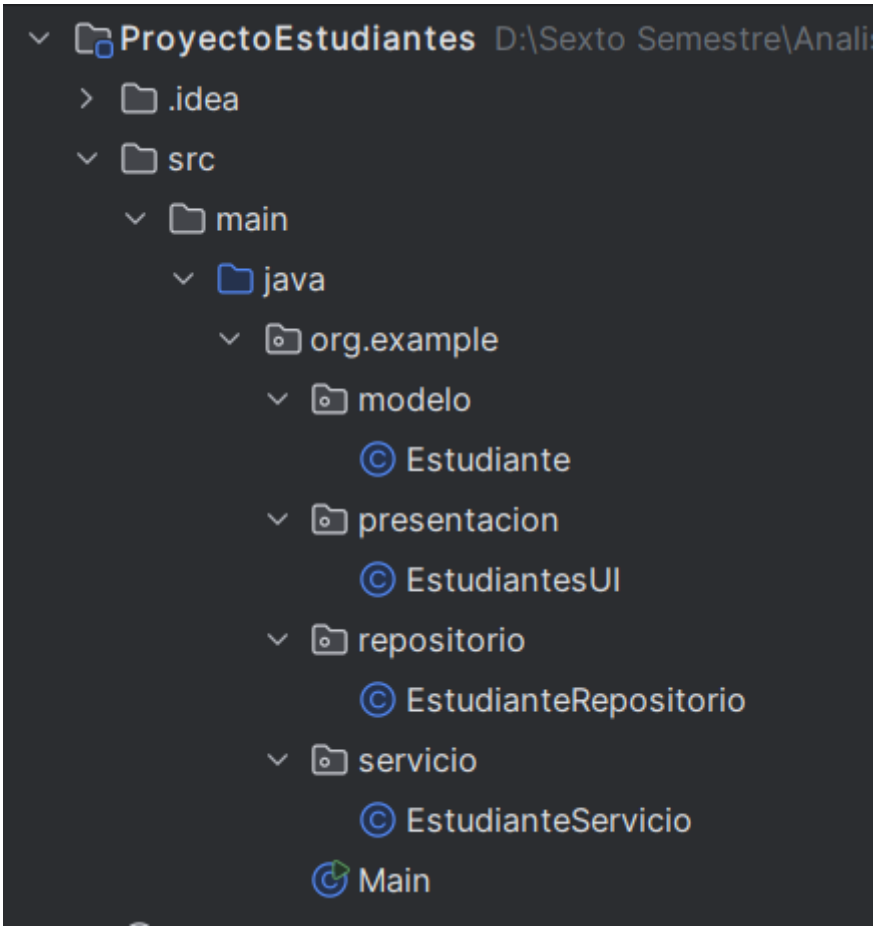
5. **Clase Principal (Punto de Entrada):**

   Responsabilidad: La clase principal, Main.java, es el punto de entrada de la aplicación. Su tarea es inicializar la interfaz gráfica y manejar el flujo de ejecución desde la presentación hacia las capas inferiores (lógica de negocio y persistencia).

   Propósito: Iniciar la aplicación y permitir que el usuario interactúe con la interfaz de usuario.

**Nombre del proyecto:** ProyectoEstudiantes

**Estructura.**




**Capas del Proyecto y su función**

**Capa Modelo**

- **Clase:** Estudiante.java
- **Responsabilidad:** Representa la entidad Estudiante con sus atributos principales: orden, nombre y edad. Es una clase POJO (Plain Old Java Object) que incluye métodos getters y setters.
- **Propósito:** Es la estructura de datos que se manipula en toda la aplicación.

```
package org.example.modelo;

/**
 * Clase que representa un estudiante con sus atributos básicos.
 */
public class Estudiante {
    private int orden;
    private String nombre;
    private int edad;


    /**
     * Constructor de la clase Estudiante.
     * @param orden Número de orden del estudiante.
     * @param nombre Nombre del estudiante.
     * @param edad Edad del estudiante.
     */
    public Estudiante(int orden, String nombre, int edad) {
        this.orden = orden;
        this.nombre = nombre;
        this.edad = edad;
    }
```

```
/**
 * Obtiene el número de orden del estudiante.
 * @return El número de orden.
 */
public int getOrden() {
    return orden;
}

/**
 * Establece el número de orden del estudiante.
 * @param orden El número de orden a establecer.
 */
public void setOrden(int orden) {
    this.orden = orden;
}

/**
 * Obtiene el nombre del estudiante.
 * @return El nombre del estudiante.
 */
public String getNombre() {
```

```
    return nombre;
}

/**
 * Establece el nombre del estudiante.
 * @param nombre El nombre a establecer.
 */
public void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
 * Obtiene la edad del estudiante.
 * @return La edad del estudiante.
 */
public int getEdad() {
    return edad;
}

/**
 * Establece la edad del estudiante.
```

```
     * @param edad La edad a establecer.
     */
    public void setEdad(int edad) {
        this.edad = edad;
    }

    /**
     * Representación en cadena del estudiante.
     * @return Una cadena con los datos del estudiante.
     */
    @Override
    public String toString() {
        return "Estudiante{orden=" + orden + ", nombre='" + nombre + "', edad=" + edad + "}";
    }
}
```

**Capa Repositorio**

- **Clase:** EstudianteRepositorio.java
- **Responsabilidad:** Maneja la **persistencia de los datos en memoria** (simula una base de datos).
- **Métodos principales:**

- agregarEstudiante(Estudiante estudiante)
- obtenerTodos()
- buscarPorOrden(int orden)
- actualizarEstudiante(Estudiante estudiante)

- eliminarEstudiante(int orden)
- **Propósito:** Centralizar la gestión de la lista de estudiantes, manteniéndola encapsulada.

```
package org.example.repositorio;


import org.example.modelo.Estudiante;

import java.util.ArrayList;

import java.util.List;


/**

 * Clase que gestiona la persistencia de datos de estudiantes en memoria.

 */

public class EstudianteRepositorio {

    private List<Estudiante> estudiantes;


    /**

     * Constructor de la clase EstudianteRepositorio.

     * Inicializa la lista de estudiantes.

     */

    public EstudianteRepositorio() {

        this.estudiantes = new ArrayList<>();

    }
```

```
/**
 * Agrega un nuevo estudiante al repositorio.
 * @param estudiante El estudiante a agregar.
 */
public void agregarEstudiante(Estudiante estudiante) {
    estudiantes.add(estudiante);
}

/**
 * Obtiene todos los estudiantes registrados.
 * @return Una lista de todos los estudiantes.
 */
public List<Estudiante> obtenerTodos() {
    return new ArrayList<>(estudiantes);
}

/**
 * Busca un estudiante por su número de orden.
 * @param orden El número de orden del estudiante.
 * @return El estudiante encontrado o null si no existe.
```

```
 */
public Estudiante buscarPorOrden(int orden) {
    for (Estudiante estudiante : estudiantes) {
        if (estudiante.getOrden() == orden) {
            return estudiante;
        }
    }
    return null;
}

/**
 * Actualiza los datos de un estudiante existente.
 * @param estudiante El estudiante con los datos actualizados.
 * @return true si se actualizó correctamente, false si no se encontró.
 */
public boolean actualizarEstudiante(Estudiante estudiante) {
    for (int i = 0; i < estudiantes.size(); i++) {
        if (estudiantes.get(i).getOrden() == estudiante.getOrden()) {
            estudiantes.set(i, estudiante);
            return true;
        }
```

```
        }
        return false;
    }

    /**
     * Elimina un estudiante por su número de orden.
     * @param orden El número de orden del estudiante a eliminar.
     * @return true si se eliminó correctamente, false si no se encontró.
     */
    public boolean eliminarEstudiante(int orden) {
        return estudiantes.removeIf(estudiante -> estudiante.getOrden() == orden);
    }
}
```

**Capa Servicio**

- **Clase:** `EstudianteServicio.java`
- **Responsabilidad:** Contiene la **lógica de negocio** que conecta la interfaz con el repositorio.
- **Funcionalidades:**
- Válida y delega operaciones al repositorio.
- Permite centralizar reglas como validación de existencia antes de actualizar o eliminar.
- **Propósito:** Separar la lógica de negocio de la lógica de almacenamiento y de la interfaz.

```
package org.example.servicio;
```

```
import org.example.modelo.Estudiante;
import org.example.repositorio.EstudianteRepositorio;
import java.util.List;

/**
 * Clase que contiene la lógica de negocio para la gestión de estudiantes.
 */
public class EstudianteServicio {
    private EstudianteRepositorio repositorio;

    /**
     * Constructor de la clase EstudianteServicio.
     * Inicializa el repositorio de estudiantes.
     */
    public EstudianteServicio() {
        this.repositorio = new EstudianteRepositorio();
    }

    /**
     * Agrega un nuevo estudiante al sistema.
```

```
     * @param orden Número de orden del estudiante.
     * @param nombre Nombre del estudiante.
     * @param edad Edad del estudiante.
     * @throws IllegalArgumentException si el orden ya existe o los datos son inválidos.
     */
    public void agregarEstudiante(int orden, String nombre, int edad) {
        if (repositorio.buscarPorOrden(orden) != null) {
            throw new IllegalArgumentException("El número de orden ya existe.");
        }
        if (nombre == null || nombre.trim().isEmpty()) {
            throw new IllegalArgumentException("El nombre no puede estar vacío.");
        }
        if (edad <= 0) {
            throw new IllegalArgumentException("La edad debe ser mayor que 0.");
        }
        Estudiante estudiante = new Estudiante(orden, nombre, edad);
        repositorio.agregarEstudiante(estudiante);
    }

    /**
     * Obtiene todos los estudiantes registrados.
```

```
     * @return Una lista de todos los estudiantes.
     */
    public List<Estudiante> listarEstudiantes() {
        return repositorio.obtenerTodos();
    }

    /**
     * Busca un estudiante por su número de orden.
     * @param orden El número de orden del estudiante.
     * @return El estudiante encontrado o null si no existe.
     */
    public Estudiante buscarEstudiante(int orden) {
        return repositorio.buscarPorOrden(orden);
    }

    /**
     * Actualiza los datos de un estudiante existente.
     * @param orden Número de orden del estudiante.
     * @param nombre Nuevo nombre del estudiante.
     * @param edad Nueva edad del estudiante.
     * @throws IllegalArgumentException si el estudiante no existe o los datos son inválidos.
```

```
 */
public void actualizarEstudiante(int orden, String nombre, int edad) {
    Estudiante estudiante = repositorio.buscarPorOrden(orden);
    if (estudiante == null) {
        throw new IllegalArgumentException("El estudiante no existe.");
    }
    if (nombre == null || nombre.trim().isEmpty()) {
        throw new IllegalArgumentException("El nombre no puede estar vacío.");
    }
    if (edad <= 0) {
        throw new IllegalArgumentException("La edad debe ser mayor que 0.");
    }
    Estudiante actualizado = new Estudiante(orden, nombre, edad);
    repositorio.actualizarEstudiante(actualizado);
}

/**
 * Elimina un estudiante por su número de orden.
 * @param orden El número de orden del estudiante.
 * @throws IllegalArgumentException si el estudiante no existe.
 */
```

```
    public void eliminarEstudiante(int orden) {
        if (repositorio.buscarPorOrden(orden) == null) {
            throw new IllegalArgumentException("El estudiante no existe.");
        }
        repositorio.eliminarEstudiante(orden);
    }
}
```

**Presentación**

- **Clase:** `EstudiantesUI.java`
- **Responsabilidad:** Administra la **interacción con el usuario por consola**.
- **Funciones clave:**
- Muestra un menú interactivo.
- Solicita entradas del usuario y las envía al servicio.
- **Propósito:** Es la capa más cercana al usuario final. Se comunica exclusivamente con la capa de servicio.

```
package org.example.presentacion;


import org.example.modelo.Estudiante;
import org.example.servicio.EstudianteServicio;


import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.ActionEvent;
```

```
import java.util.List;


/**
 * Clase que implementa una interfaz gráfica para gestionar estudiantes utilizando Swing.
 * Proporciona un formulario para ingresar datos de estudiantes y una tabla para visualizarlos,
 * permitiendo operaciones CRUD (Crear, Leer, Actualizar, Eliminar).
 */
public class EstudiantesUI extends JFrame {


    /** Campo de texto para el identificador único del estudiante. */
    private JTextField campoId;


    /** Campo de texto para el nombre del estudiante. */
    private JTextField campoNombre;


    /** Campo de texto para la edad del estudiante. */
    private JTextField campoEdad;


    /** Botón para crear un nuevo estudiante. */
    private JButton btnCrear;
```

```
/** Botón para leer y mostrar todos los estudiantes en la tabla. */
private JButton btnLeer;

/** Botón para actualizar los datos de un estudiante existente. */
private JButton btnActualizar;

/** Botón para eliminar un estudiante por su ID. */
private JButton btnEliminar;

/** Tabla que muestra la lista de estudiantes. */
private JTable tablaEstudiantes;

/** Modelo de datos para la tabla de estudiantes. */
private DefaultTableModel modeloTabla;

/** Servicio que maneja la lógica de negocio para los estudiantes. */
private EstudianteServicio servicio;

/**
 * Constructor que inicializa la interfaz gráfica y configura los componentes.
 */
```

```
public EstudiantesUI() {
    servicio = new EstudianteServicio();

    // Configurar la ventana principal
    setTitle("Gestión de Estudiantes");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(800, 400);
    setLocationRelativeTo(null);

    // Panel de formulario
    JPanel panelFormulario = new JPanel(new GridLayout(2, 3, 10, 10));
    panelFormulario.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

    // Crear campos de entrada
    campoId = new JTextField();
    campoNombre = new JTextField();
    campoEdad = new JTextField();

    // Añadir etiquetas y campos al panel
    panelFormulario.add(new JLabel("ID:"));
    panelFormulario.add(new JLabel("Nombre:"));
```

```
panelFormulario.add(new JLabel("Edad:"));
panelFormulario.add(campoId);
panelFormulario.add(campoNombre);
panelFormulario.add(campoEdad);

// Panel de botones
JPanel panelBotones = new JPanel(new FlowLayout(FlowLayout.CENTER));
btnCrear = new JButton("Crear");
btnLeer = new JButton("Leer");
btnActualizar = new JButton("Actualizar");
btnEliminar = new JButton("Eliminar");

// Añadir botones al panel
panelBotones.add(btnCrear);
panelBotones.add(btnLeer);
panelBotones.add(btnActualizar);
panelBotones.add(btnEliminar);

// Agrupar formulario y botones
JPanel panelSuperior = new JPanel(new BorderLayout());
panelSuperior.add(panelFormulario, BorderLayout.CENTER);
```

```
panelSuperior.add(panelBotones, BorderLayout.SOUTH);

// Configurar tabla
String[] columnas = {"ID", "Nombre", "Edad"};
modeloTabla = new DefaultTableModel(columnas, 0);
tablaEstudiantes = new JTable(modeloTabla);
JScrollPane scrollTabla = new JScrollPane(tablaEstudiantes);

// Layout principal
setLayout(new BorderLayout(10, 10));
add(panelSuperior, BorderLayout.NORTH);
add(scrollTabla, BorderLayout.CENTER);

// Configurar eventos de los botones
btnCrear.addActionListener(this::crearEstudiante);
btnLeer.addActionListener(e -> cargarTabla());
btnActualizar.addActionListener(this::actualizarEstudiante);
btnEliminar.addActionListener(this::eliminarEstudiante);

// Cargar datos iniciales en la tabla
cargarTabla();
```

```
    }

    /**
     * Crea un nuevo estudiante y lo agrega al sistema, actualizando la tabla.
     * @param e Evento de acción generado por el botón "Crear".
     */
    private void crearEstudiante(ActionEvent e) {
        try {
            int id = Integer.parseInt(campoId.getText());
            String nombre = campoNombre.getText();
            int edad = Integer.parseInt(campoEdad.getText());
            servicio.agregarEstudiante(id, nombre, edad);
            JOptionPane.showMessageDialog(this, "Estudiante agregado exitosamente.", "Éxito",
JOptionPane.INFORMATION_MESSAGE);
            limpiarCampos();
            cargarTabla();
        } catch (NumberFormatException ex) {
            mostrarError("Ingrese valores numéricos válidos para ID y edad.");
        } catch (IllegalArgumentException ex) {
            mostrarError(ex.getMessage());
        }
    }
```

```
    /**
     * Actualiza los datos de un estudiante existente y refresca la tabla.
     * @param e Evento de acción generado por el botón "Actualizar".
     */
    private void actualizarEstudiante(ActionEvent e) {
        try {
            int id = Integer.parseInt(campoId.getText());
            String nombre = campoNombre.getText();
            int edad = Integer.parseInt(campoEdad.getText());
            servicio.actualizarEstudiante(id, nombre, edad);
            JOptionPane.showMessageDialog(this, "Estudiante actualizado exitosamente.",
"Éxito", JOptionPane.INFORMATION_MESSAGE);
            limpiarCampos();
            cargarTabla();
        } catch (NumberFormatException ex) {
            mostrarError("Ingrese valores numéricos válidos para ID y edad.");
        } catch (IllegalArgumentException ex) {
            mostrarError(ex.getMessage());
        }
    }
```

```
    /**
     * Elimina un estudiante por su ID y actualiza la tabla.
     * @param e Evento de acción generado por el botón "Eliminar".
     */
    private void eliminarEstudiante(ActionEvent e) {
        try {
            int id = Integer.parseInt(campoId.getText());
            servicio.eliminarEstudiante(id);
            JOptionPane.showMessageDialog(this, "Estudiante eliminado exitosamente.",
"Éxito", JOptionPane.INFORMATION_MESSAGE);
            limpiarCampos();
            cargarTabla();
        } catch (NumberFormatException ex) {
            mostrarError("Ingrese un valor numérico válido para el ID.");
        } catch (IllegalArgumentException ex) {
            mostrarError(ex.getMessage());
        }
    }

    /**
     * Carga la lista de estudiantes en la tabla.
     */
```

```
private void cargarTabla() {
    modeloTabla.setRowCount(0);
    List<Estudiante> estudiantes = servicio.listarEstudiantes();
    for (Estudiante est : estudiantes) {
        modeloTabla.addRow(new Object[]{est.getOrden(), est.getNombre(), est.getEdad()});
    }
}

/**
 * Limpia los campos de texto del formulario.
 */
private void limpiarCampos() {
    campoId.setText("");
    campoNombre.setText("");
    campoEdad.setText("");
}

/**
 * Muestra un mensaje de error en una ventana emergente.
 * @param mensaje El mensaje de error a mostrar.
 */
```

```
    private void mostrarError(String mensaje) {
        JOptionPane.showMessageDialog(this, mensaje, "Error",
JOptionPane.ERROR_MESSAGE);
    }
}
```

## Clase Principal

- **Clase:** `Main.java`
- **Responsabilidad:** Es el punto de entrada de la aplicación.
- **Funcionalidad:** Crea una instancia de `EstudiantesUI` y ejecuta el método `mostrarMenu()`, lo cual inicia la interacción del usuario.
- **Propósito:** Inicia el sistema y da paso a la ejecución lógica mediante el flujo desde la interfaz hacia las capas inferiores.

```
package org.example;

import org.example.presentacion.EstudiantesUI;

/**
 * Clase principal que sirve como punto de entrada para la aplicación.
 * Inicia la interfaz gráfica de gestión de estudiantes.
 */
public class Main {

    /**
```

```
     * Método principal que inicia la aplicación.
     * Crea y muestra la interfaz gráfica {@link EstudiantesUI} en el hilo de despacho de eventos de Swing.
     * @param args Argumentos de la línea de comandos (no utilizados).
     */
    public static void main(String[] args) {
        // Iniciar la interfaz gráfica de Swing en el hilo de eventos de Swing
        javax.swing.SwingUtilities.invokeLater(() -> {
            EstudiantesUI ventana = new EstudiantesUI();
            ventana.setVisible(true);
        });
    }
}
```

| **Para qué sirve la arquitectura en capas** | |
|---|---|
| Organización y claridad: | Cada capa tiene una responsabilidad específica, lo que permite un desarrollo más estructurado y fácil de entender. Esto también facilita el mantenimiento y la escalabilidad del sistema. |
| Separación de Preocupaciones: | Cada capa se encarga de una parte del sistema (modelo, lógica de negocio, persistencia, presentación), lo que permite que cada componente evolucione de manera independiente. Si se necesita cambiar la forma de almacenar los datos o actualizar la interfaz de usuario, se puede hacer sin afectar al resto del sistema. |
| Facilidad de Mantenimiento: | Gracias a la separación de responsabilidades, los cambios o mejoras se pueden implementar en capas específicas sin causar |

| | |
|---|---|
| | impacto en el sistema global. Esto es clave para proyectos que deben evolucionar a largo plazo. |
| Reusabilidad: | Al separar las responsabilidades, se facilita la reutilización del código. Por ejemplo, la capa de servicio se puede usar con diferentes interfaces de usuario (como una versión web o de consola) sin tener que modificar la lógica de negocio. |
| Flexibilidad y Escalabilidad: | Esta arquitectura permite la integración de nuevas funcionalidades o la modificación de las existentes sin afectar a las demás capas, lo que facilita la adición de nuevas características o la expansión del sistema. |

**Conclusiones.**

La arquitectura en tres capas permitió estructurar el sistema de manera lógica y clara, definiendo con precisión el propósito de cada componente. Esta organización favorece una comprensión más sencilla del flujo del programa y la identificación del rol de cada clase dentro del proyecto.

Cada capa cumplió una función específica: el modelo definió la estructura de los datos, el repositorio gestionó la persistencia en memoria, el servicio se encargó de aplicar la lógica del negocio, y la presentación facilitó la interacción del usuario con el sistema. Esta división permitió un desarrollo ordenado y una mejor organización del código.

La mantenibilidad del sistema se vio altamente beneficiada gracias a esta separación. Cualquier modificación en la forma de almacenar datos, en las reglas de negocio o en la interfaz de usuario puede realizarse de manera independiente sin afectar al resto de componentes. Esta característica es clave en proyectos que requieren adaptaciones frecuentes o evolución a largo plazo.

En cuanto a los patrones utilizados, el sistema se apoyó en el patrón de arquitectura en capas y presenta similitudes con el patrón MVC, lo cual mejora la reutilización del código y su escalabilidad. Esto permite, por ejemplo, sustituir la consola por una interfaz gráfica sin alterar la lógica del negocio ni la persistencia de datos.