UNIVERSIDAD MAYOR DE SAN ANDRES

CARRERA DE INFORMATICA

PROGRAMACION II

INF-121

Proyecto: Plataforma de Eventos Sociales

GRUPO 21

Estudiantes:

Albarracin Flores Wanda Esnider

Aruquipa Aguilar Rimer Diego

Chuquimia Limachi Cristhian José

Pacha Choque Angel

Docente: Lic. Rosalía López Montalvo

Fecha: 24 de julio del 2025

LA PAZ – BOLIVIA

1.Descripción del Proyecto

El Sistema de Gestión de Eventos Sociales es una aplicación integral desarrollada en Java que revoluciona la manera de organizar y administrar eventos sociales de gran escala. Este sistema surge de la necesidad de contar con una herramienta tecnológica que permita a organizadores, empresas de eventos y entidades culturales gestionar de manera eficiente todos los aspectos involucrados en la planificación y ejecución de eventos como festivales musicales, ferias gastronómicas, exposiciones culturales y celebraciones comunitarias.

La aplicación está diseñada bajo los principios fundamentales de la Programación Orientada a Objetos (POO), implementando una arquitectura modular y escalable que facilita tanto el mantenimiento como la extensión de funcionalidades. El sistema aborda las complejidades inherentes a la gestión de eventos, desde la coordinación de múltiples organizadores hasta el control detallado de recursos financieros, logísticos y humanos.

2. Objetivos

Objetivos Específicos

1. Implementar una Arquitectura Orientada a Objetos:

-Aplicar los cuatro pilares fundamentales de POO: encapsulamiento, herencia, polimorfismo y abstracción

-Establecer relaciones apropiadas entre clases (composición, agregación, asociación, herencia)

- Implementar interfaces y clases abstractas donde sea apropiado para maximizar la flexibilidad del diseño

2. Desarrollar una Interfaz Gráfica Profesional y Funcional:

- Crear una interfaz gráfica completa utilizando Java Swing con componentes profesionales

- Implementar múltiples paneles especializados para diferentes funciones del sistema

- Desarrollar formularios intuitivos para la entrada y manipulación de datos

- Crear sistemas de navegación fluidos entre diferentes secciones de la aplicación

- Implementar validaciones en tiempo real en la interfaz gráfica

- Establecer un diseño visual atractivo y consistente en toda la aplicación

3. Implementar un Sistema de Persistencia:

-Desarrollar conectividad completa con base de datos MySQL para almacenamiento permanente

Implementar operaciones CRUD completas para todas las entidades del sistema

Crear un sistema de serialización de objetos Java para manejo de datos temporales

- Implementar transacciones de base de datos para garantizar la consistencia

- Desarrollar consultas optimizadas para mejorar el rendimiento del sistema

4. Crear un Sistema de Validación y Control de Errores:

- Implementar validaciones comprehensivas en todos los niveles de la aplicación

- Desarrollar un sistema de manejo de excepciones que cubra todos los escenarios posibles

- Crear mecanismos de recuperación de errores que mantengan la estabilidad del sistema

- Establecer mensajes de error informativos y orientados al usuario

3. Análisis del problema

Descripción del Contexto

Los eventos sociales requieren una gestión compleja que involucra múltiples actores: organizadores, participantes, proveedores de servicios (grupos musicales, decoraciones), y sistemas de control de acceso (entradas). La falta de un sistema centralizado genera problemas de coordinación y control.

Requisitos Funcionales

1. RF01: El sistema debe permitir crear eventos de tipo Festival y Feria
2. RF02: Debe gestionar usuarios y organizadores con diferentes niveles de acceso
3. RF03: Debe administrar entradas con tipos, códigos y precios variables
4. RF04: Debe permitir inscripción de usuarios a eventos específicos
5. RF05: Debe gestionar recursos del evento (decoraciones, grupos musicales)
6. RF06: Debe proporcionar interfaces gráficas para todas las operaciones CRUD
7. RF07: Debe mantener persistencia de datos entre sesiones

Requisitos No Funcionales

1. RNF01: Usabilidad** - Interfaz intuitiva con navegación clara
2. RNF02: Confiabilidad** - Validación de datos y manejo de excepciones
3. RNF03: Mantenibilidad** - Código modular con separación de responsabilidades
4. RNF04: Escalabilidad** - Uso de colecciones dinámicas para crecimiento

Casos de Uso

- CU01: Crear Evento** - Organizador crea festival o feria con datos básicos
- CU02: **Gestionar Entradas** - Administrar tipos, precios y disponibilidad
- CU03: Inscribir Usuario** - Usuario se registra a evento específico
- CU04: Ver Eventos Activos** - Consultar eventos disponibles
- CU05: Administrar Recursos** - Gestionar decoraciones y grupos musicales

4. Diseño del sistema

4.1. Clases y jerarquías

 Diagrama UML

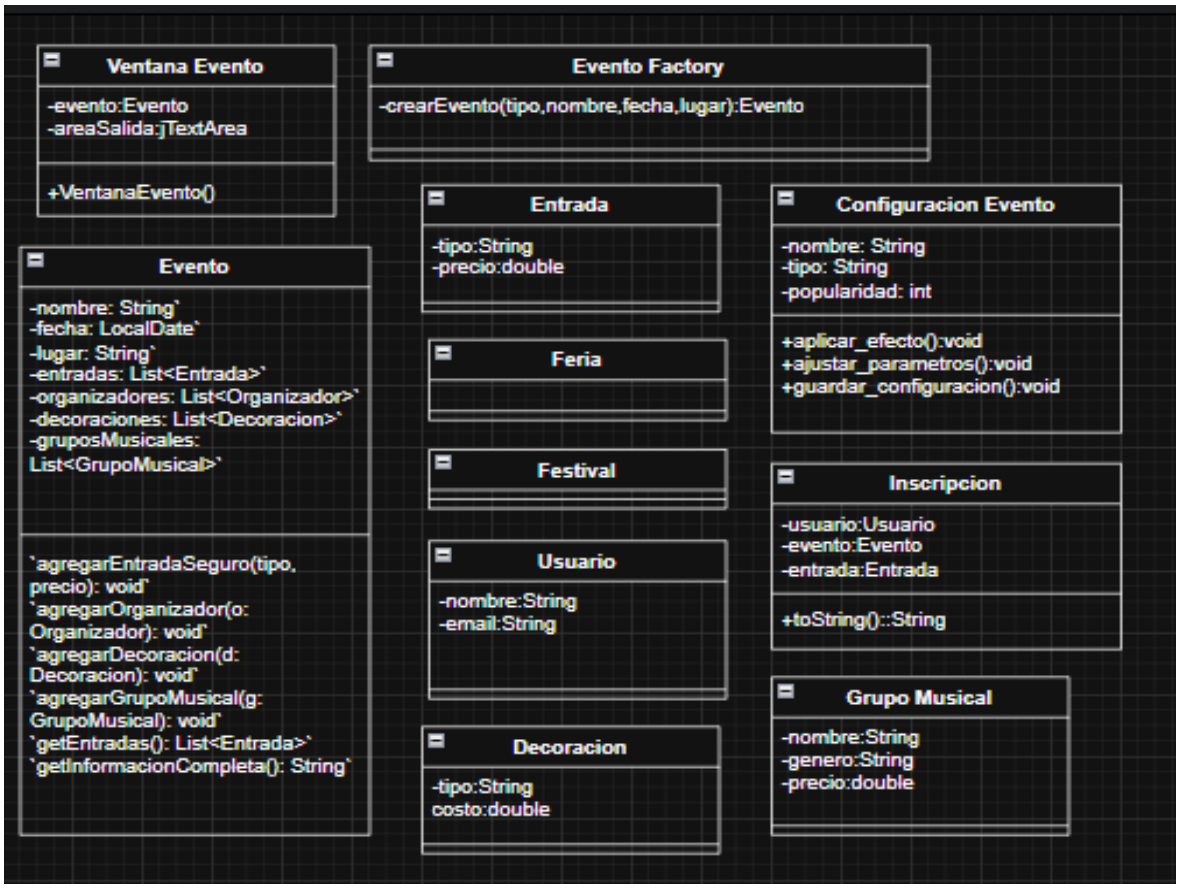


Tabla de clases:

Clase	Atributos principales	Metodos clave
Usuario	nombre, email	toString()
Organizador	Nombre,teléfono	toString()
Evento	nombre,fecha,lugar,entradas,organizadores decoraciones, grupos Musicales	agregarEntradaSeguro(tipo,precio): void agregarOrganizador(o:Organizador):void agregarDecoracion(d:Decoracion)void agregarGrupoMusical(g:GrupoMusical):void getEntradas()List<Entrada> getInformacionCompleta():String
Festival	hereda de Evento	mostrarInfo()
Feria	hereda de Evento	mostrarInfo()
Entrada	Tipo,precio	mostrar(),
Inscripcion	usuario, evento, entrada	toString()
Decoracion	tipo, costo	toString()

GrupoMusical	nombre, genero, cache	toString()
VentanaEvento	Evento,areaSalida	VentanaEvento()
EventoFactory	Hereda de Evento	crearEvento(tipo,nombre,fecha,lugar)
ConfiguracionEvento	Instancia,versión,plataforma, totaldeParticipantes	getInstancia(),motrarConfiguracion()

4.2. Relaciones (Herencia, Agregación, Composición)

🔷 HERENCIA

1. Evento → Feria, Festival
    - Feria y Festival **heredan** de la clase base Evento.
    - Se representa con una flecha blanca con punta triangular.
2. Usuario → Organizador (esta relación **no está dibujada**, pero tú la mencionas)
    - Organizador sería una subclase que extiende Usuario.
    - Debería estar representada como flecha de herencia (aunque en el diagrama solo figura como clase separada)

🔶 COMPOSICIÓN (diamante negro = dependencia fuerte, vida compartida)

En el UML, estas relaciones indican que **Evento es dueño y responsable** de la vida de los siguientes objetos:

1. Evento ◆——→ Entrada
    - Evento contiene una lista de entradas (List<Entrada> entradas)
    - Si el evento se elimina, sus entradas también.
2. Evento ◆——→ Decoracion
    - Evento posee decoraciones específicas (List<Decoracion> decoraciones)
3. Evento ◆——→ GrupoMusical
    - Evento mantiene una lista de grupos musicales que le pertenecen (List<GrupoMusical> gruposMusicales)

◇ AGREGACIÓN (diamante blanco = dependencia débil, vida independiente)

Estas relaciones indican que un objeto **puede existir independientemente del otro**:

1. Evento ◇——→ Organizador
    - Evento puede tener múltiples organizadores (List<Organizador> organizadores)
    - Los organizadores existen independientemente del evento.
2. Inscripcion ◇——→ Usuario
    - Una inscripción **usa** un usuario.
    - El usuario no depende de la inscripción.
3. Inscripcion ◇——→ Evento
    - Una inscripción **se refiere** a un evento ya existente.
    - El evento no depende de la inscripción.

5. Desarrollo

Estructura general del código:

Clases y Subclases

Herencia implementada

```
public class Festival extends Evento {
   @Override
   public void mostrarInfo() {
      System.out.println("\n====== FESTIVAL ======");
      super.mostrarInfo();
   }
}

public class Organizador extends Usuario {
   protected String telefono;

   public void crearEvento() { /* implementación */ }
   public void gestionaEvento() { /* implementación */ }
}
```

Uso de Genéricos

```
public class EventosActivos<T> extends javax.swing.JPanel {
   private T op;

   public T optener(T op, String n1) throws IOException, ClassNotFoundException {
      FileInputStream fi = new FileInputStream("C:\\Users\\...\\"+n1);
      ObjectInputStream ob = new ObjectInputStream(fi);
      op = (T) ob.readObject();
      ob.close();
      return op;
   }
}
```

Métodos Sobrescritos

Polimorfismo en acción

```
@Override
public void mostrarInfo() {
   System.out.println("\n====== FERIA ======");
   super.mostrarInfo(); // Reutiliza código de la clase padre
}
```

Validaciones

```
public Entrada(String tipo, String codigo, double precio, String lugar, String descripcion) {
   this.tipo = tipo;
   this.codigo = codigo;
   if (precio < 0) {
      throw new IllegalArgumentException("El precio de la entrada no puede ser negativo.");
   }
   this.precio = precio;
   this.lugar = lugar;
   this.descripcion = descripcion;
}
```

Persistencia con Archivos

Serialización

```
public T optener(T op, String n1) throws IOException, ClassNotFoundException {
   FileInputStream fi = new FileInputStream("C:\\Users\\WINDOWS 11 PRO\\Desktop\\Serializacion\\"+n1);
   ObjectInputStream ob = new ObjectInputStream(fi);
   op = (T) ob.readObject();
   ob.close();
   return op;
}
```

Base de datos MySQL

```
Connection cn = DriverManager.getConnection("jdbc:mysql://localhost/persistencia", "root", "");
PreparedStatement pst = cn.prepareStatement("insert into eventos values(?,?,?,?,?)");
```

Excepciones Personalizadas

Manejo de excepciones en inscripciones

```
public Inscripcion(Usuario usuario, Evento evento, Entrada entrada) {
   if (usuario == null || evento == null || entrada == null) {
      throw new IllegalArgumentException("Ningún dato puede ser nulo para la inscripción.");
   }
   this.usuario = usuario;
   this.evento = evento;
   this.entrada = entrada;
}
```

6. Aplicación de Patrones de Diseño

El patrón Singleton se usa en la clase ConfiguracionEvento para garantizar que solo exista una configuración global para todo el sistema, incluyendo la versión del sistema y el total de participantes. Solo se crea una vez y se reutiliza cuando es necesario.

Usamos el patrón Factory para encapsular la lógica de creación de los eventos, separando la lógica de VentanaEvento y haciendo el código más reutilizable y extensible. Si en el futuro se agregan más tipos de eventos, solo se modifica la clase EventoFactory.

6.1. Patrón(es) aplicados

Clase	Patron de diseño Implementado
ConfiguracionEvento	Singleton
EventoFactory	Factory
Evento,Feria,Festival	Herencia / Polimorfismo
VentanaEvento	Interfaz gráfica (GUI) + relación con clases del modelo
Serializable	Persistencia (no visible en el diagrama, pero aplicada en clases del modelo)

6.2. Justificación

El patrón Singleton se usa en la clase ConfiguracionEvento para garantizar que solo exista una configuración global para todo el sistema, incluyendo la versión del sistema y el total de participantes. Solo se crea una vez y se reutiliza cuando es necesario.

Usamos el patrón Factory para encapsular la lógica de creación de los eventos, separando la lógica de VentanaEvento y haciendo el código más reutilizable y extensible. Si en el futuro se agregan más tipos de eventos, solo se modifica la clase EventoFactory.

6.3. Diagrama y ejemplo de uso

Patrón Singleton

```
mostrarConfig.addActionListener(ev -> {
    ConfiguracionEvento config = ConfiguracionEvento.getInstancia();
    config.mostrarConfiguracion();
    areaSalida.append("✓ Mostrando configuración del sistema en consola.\n");
});
```

Patrón Factory (Fábrica)

```
public class EventoFactory {
    public static Evento crearEvento(String tipo, String nombre, LocalDate fecha, String lugar) {
        if (tipo.equalsIgnoreCase("feria")) {
            return new Feria(nombre, fecha, lugar);
        } else if (tipo.equalsIgnoreCase("festival")) {
            return new Festival(nombre, fecha, lugar);
        } else {
            throw new IllegalArgumentException("Tipo de evento desconocido: " + tipo);
        }
    }
}
```

7. Persistencia de datos

Descripción del Formato Usado

El sistema implementa persistencia dual:

1. Base de Datos MySQL (.sql)

- Tabla: `eventos`
- Campos: Id, Nombre, Fecha, Lugar, Descripción

2. Serialización de Objetos(.dat)

- Archivos binarios para objetos complejos
- Ubicación: `C:\Users\WINDOWS 11 PRO\Desktop\Serializacion\`

Clase Encargada de Lectura/Escritura

MySQL - Clase `Crear`

Inserción

Connection cn = DriverManager.getConnection("jdbc:mysql://localhost/persistencia", "root", "");

```
PreparedStatement pst = cn.prepareStatement("insert into eventos values(?,?,?,?,?)");
pst.setString(1, txt1.getText().trim());
pst.setString(2, txt2.getText().trim());
pst.setString(3, txt3.getText().trim());
pst.setString(4, txt4.getText().trim());
pst.setString(5, txt5.getText().trim());
pst.executeUpdate();
```

Consulta

```
PreparedStatement pst = cn.prepareStatement("select * from eventos where Id=?");
pst.setString(1, txtBuscar.getText().trim());
ResultSet rs = pst.executeQuery();
```

Serialización - Clase `EventosActivos`

```
public T optener(T op, String n1) throws IOException, ClassNotFoundException {
   FileInputStream fi = new FileInputStream("C:\\Users\\WINDOWS 11
PRO\\Desktop\\Serializacion\\"+n1);
   ObjectInputStream ob = new ObjectInputStream(fi);
   op = (T) ob.readObject();
   ob.close();
   return op;
}
```

Ejemplo de Archivo Generado

Estructura de Base de Datos

```
CREATE TABLE eventos (
   Id VARCHAR(50) PRIMARY KEY,
   Nombre VARCHAR(100),
   Fecha VARCHAR(50),
   Lugar VARCHAR(100),
   Descripción TEXT
);

-- Ejemplo de registro
INSERT INTO eventos VALUES
('1', 'Festival de Rock 2024', '2024-08-15', 'Estadio Nacional', 'Gran festival de música
rock');
```

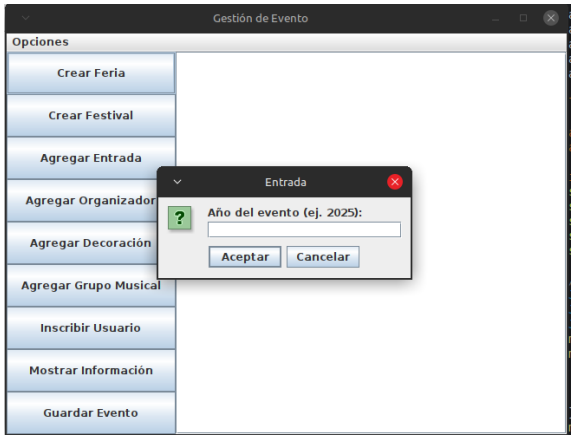
Archivo Serializado (Datos.dat)

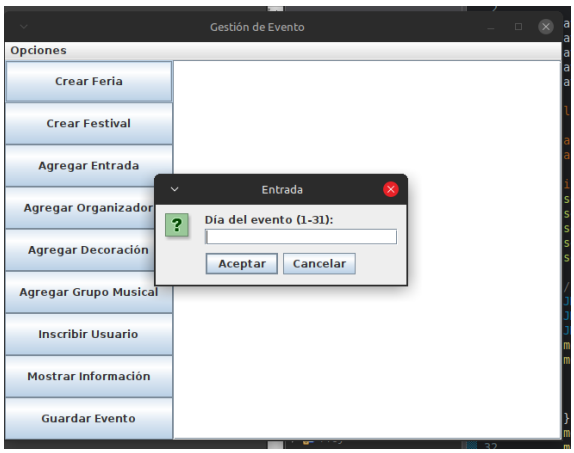
Archivo binario que contiene:

- Array de Strings con nombres de eventos
- Objetos serializados de tipo EventosActivos<T>
- Metadatos de configuración del sistema

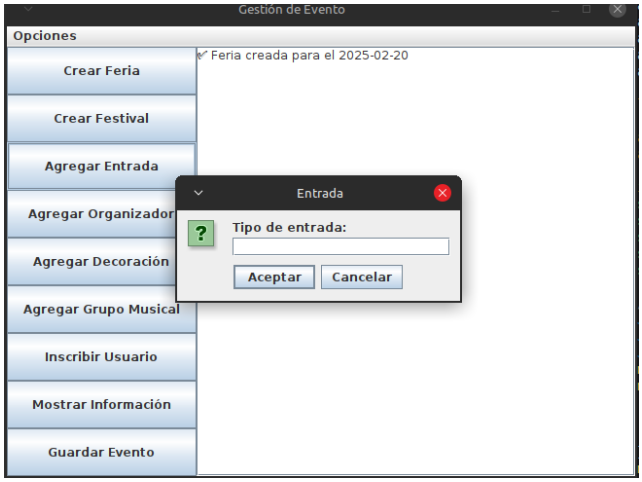
8. Pruebas y validación

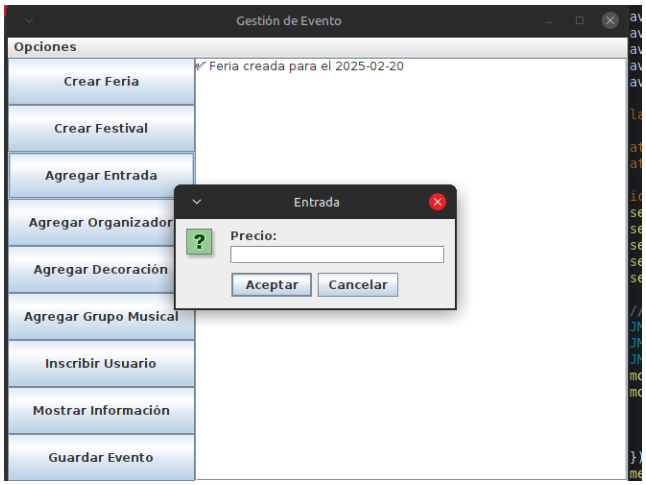
Casos de Prueba Realizados
CP01: Creación de Feria
- **Entrada**: Feria("GOOGLE UMSA", "2025-02-20", "UMSA")
- **Salida Esperada**: Feria creada para el 2025-02-20



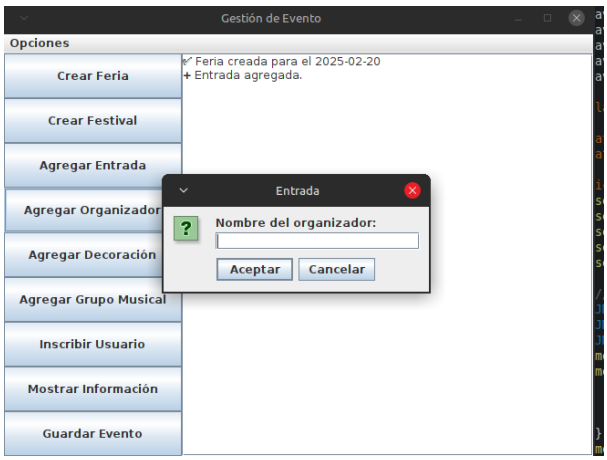


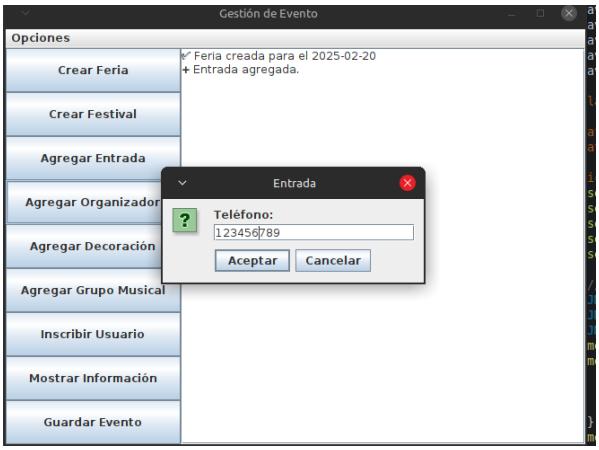
CP02: Validación de Entrada
- **Entrada**: Entrada(General", 20.0Bs)
- **Salida Esperada**: IllegalArgumentException("El precio de la entrada no puede ser negativo")



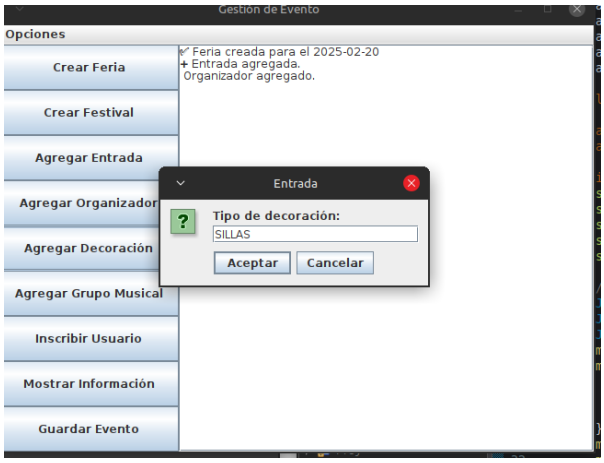


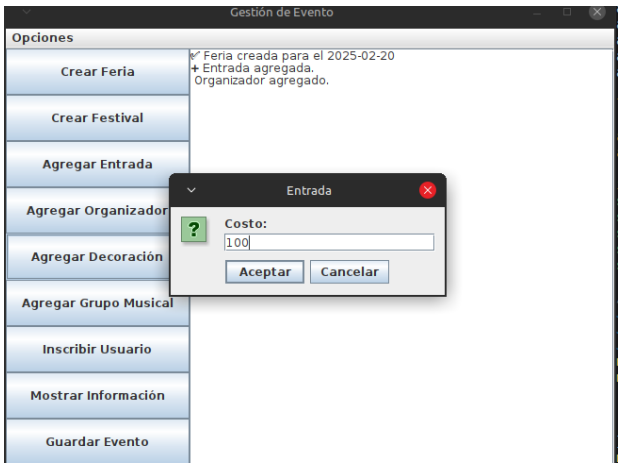
CP03: Agregar Organizador
- **Entrada**: Nombre("ESTUDIANTES DE INFORMATICA",123456789)
- **Salida Esperada**: (ESTUDIANTES DE INFORMATICA-123456789



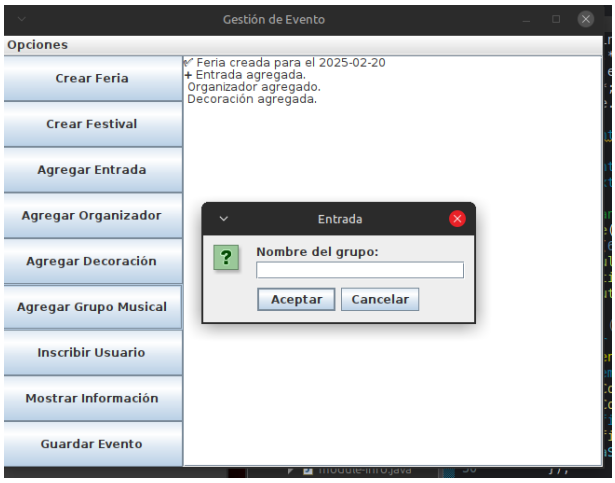


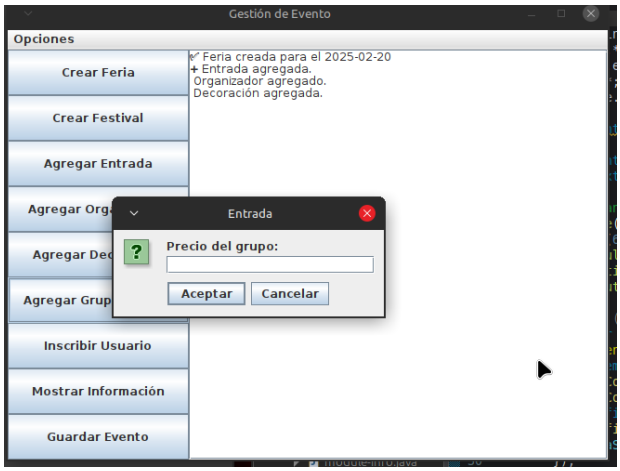
CP04: Agregar Decoracion
- **Entrada**:Tipo de Decoración("SILLAS", 100Bs)
- **Salida Esperada**Decoración agregada





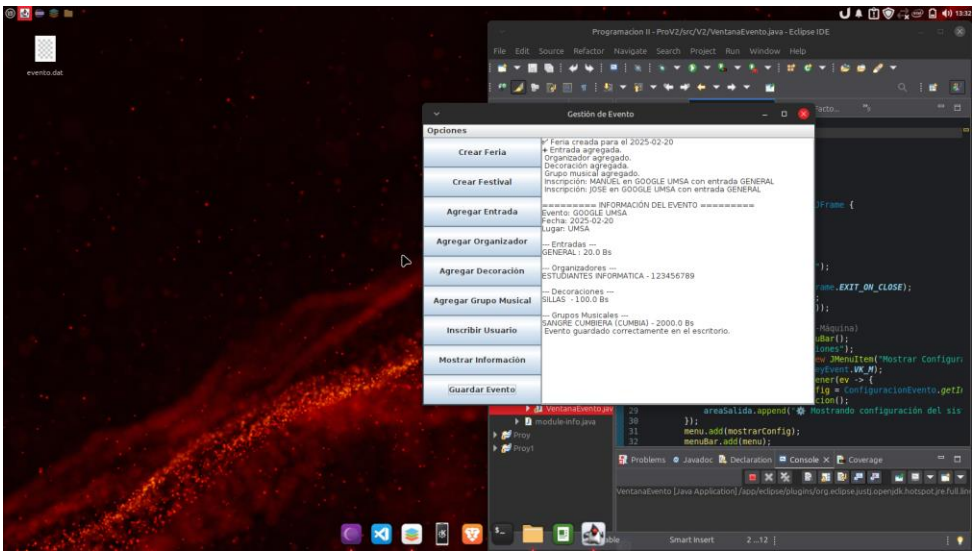
CP05: Agregación Grupo Musical
- Entrada**:Grupo Musica("SANGRE CUMBIERA(CUMBIA) ",2000.0Bs)
- Salida Esperada**:





P06 Mostrar Información

CP07: Guardando info en un .dat



9. Conclusiones

Reflexión sobre el Uso de Patrones de Diseño en el Proyecto

Basándose en la documentación proporcionada, el Sistema de Gestión de Eventos Sociales implementó específicamente **dos patrones de diseño principales** junto con conceptos fundamentales de POO. La implementación se centró en patrones que resuelven problemas concretos del dominio de gestión de eventos.

Ventajas Observadas en la Estructura del Sistema

Arquitectura Simplificada y Efectiva

Los patrones implementados han creado una arquitectura clara y bien estructurada:

Separación de responsabilidades:

- ConfiguracionEvento (Singleton): Maneja la configuración global del sistema

- EventoFactory (Factory): Se encarga exclusivamente de la creación de eventos

- Jerarquía de Eventos: Implementa herencia y polimorfismo para diferentes tipos de eventos

- VentanaEvento: Maneja la interfaz gráfica y su relación con el modelo

- Serializable: Proporciona persistencia de datos

Aplicación de Contenidos de la Materia

Principios de POO Implementados

1. Encapsulamiento Avanzado:

```
public class ConfiguracionEvento {
    private static ConfiguracionEvento instancia;
    private ConfiguracionEvento() {} // Constructor privado

    public static ConfiguracionEvento getInstancia() {
        if (instancia == null) {
            instancia = new ConfiguracionEvento();
        }
        return instancia;
    }
}
```

2. Herencia Estratégica:

La jerarquía de eventos (`Evento` → `Festival`, `Feria`) demuestra uso apropiado de herencia para compartir comportamientos comunes mientras permite especialización.

3. Polimorfismo Efectivo:

```
Evento evento1 = new Festival(...);
Evento evento2 = new Feria(...);
```

4.Patrones de Diseño Clásicos:

- Singleton: Para gestión de configuración global

- Factory: Para encapsular creación de objetos complejos

5. Arquitectura en Capas:

- Capa de Presentación: VentanaEvento (GUI)

- Capa de Lógica de Negocio: Clases de eventos

- Capa de Configuración: ConfiguracionEvento

- Capa de Persistencia: Serializables

6.Persistencia:

La implementación de `Serializable` demuestra comprensión de conceptos de persistencia de objetos.

Mejoras en la Interfaz

1. MVC Más Explícito:

Separar claramente el controlador de la vista para mejor organización.

2. Event Handling:

Implementar manejo de eventos de GUI más robusto.

10. Distribución de Roles del Equipo

Rol / Integrante	Responsabilidad principal
Chuquimia Limachi Cristhian José	Jefe de grupo – código principal
Pacha Choque Angel	Conexión base de datos con código principal
Aruquipa Aguilar Rimer Diego	Interfaz Grafica
Albarracin Flores Wanda Esnider	Informe completo

11. Anexos

 Código fuente con clases organizadas por paquete (ej. modelo, servicio, util, factory, etc.)

```
src/
├── module-info.java
└── V2/
    ├── ConfiguracionEvento.java
    ├── Decoracion.java
    ├── Entrada.java
    ├── Evento.java
    ├── EventoFactory.java
    ├── Feria.java
    ├── Festival.java
    ├── GrupoMusical.java
    ├── Inscripcion.java
    ├── Organizador.java
    ├── Usuario.java
    └── VentanaEvento.java
```

Diagrama UML