Universidad Argentina De La Empresa

Licenciatura en Gestión de Tecnología de la Información

Trabajo Práctico Obligatorio

Paradigma Orientado a Objetos

Sistema de Gestión de Eventos

Estudiante: Lucas Rodas

Legajo: 1199266

Introducción

El Trabajo Práctico Obligatorio (TPO) de la materia Paradigma Orientado a Objetos consiste en desarrollar un sistema de gestión de eventos utilizando Java y la biblioteca Swing.

El objetivo principal del TPO es sintetizar las habilidades adquiridas durante la cursada de la materia. En línea con este propósito, decidí ampliar el alcance del proyecto, con la intención de que el resultado final no solo cumpliera con los requerimientos solicitados, sino que además pudiera integrarse a mi portfolio profesional como desarrollador de software.

Considero fundamental contar con un portfolio que refleje no solo mis habilidades técnicas, sino también mi capacidad para abordar proyectos de forma integral, ya que es un aspecto clave para la inserción y crecimiento en el mercado laboral. Este trabajo representó una oportunidad valiosa para enfrentar el desarrollo completo de una aplicación, desde el diseño de la arquitectura hasta la implementación de una interfaz de usuario.

Además, esta instancia me permitió combinar mi experiencia previa en desarrollo con el conocimiento específico adquirido en esta materia, particularmente en el uso del lenguaje Java y la biblioteca Swing. Incorporar un proyecto así en mi portfolio era una oportunidad que consideré imprescindible aprovechar.

Para llevar adelante la implementación del proyecto de manera eficaz, eficiente y en tiempos razonables, opté por utilizar patrones de diseño. Si bien estos patrones son abordados en materias más avanzadas de la carrera, me resultaron fundamentales para mantener una organización clara, facilitar la iteración del código y simplificar la resolución de errores. Esta decisión surgió especialmente luego de enfrentar dificultades organizativas en trabajos previos que involucraban Swing. Asimismo, la adopción de principios SOLID resultó esencial para asegurar la calidad del trabajo dentro del período asignado.

A diferencia de entregas anteriores, decidí realizar esta entrega mediante un repositorio en GitHub. Este enfoque me permitió reforzar aspectos clave del desarrollo profesional como el control de versiones, la documentación clara y la visibilidad pública de mi trabajo. Considero que estas habilidades son esenciales, y hasta el momento no hemos tenido muchas oportunidades para practicarlas durante la carrera.

En definitiva, abordé este trabajo práctico con la visión de que sea más que una simple entrega de la materia, buscando que sea representativo de mi forma de trabajar y que me permita abrir puertas profesionales.

Marco conceptual

Propósito general

La aplicación es un sistema de gestión de eventos que permite a diferentes tipos de usuarios (organizadores y asistentes) crear, visualizar y registrar eventos con sus respectivos datos relevantes (fecha, lugar, asistentes, etc.).

El sistema diferencia funcionalidades según el rol del usuario y mantiene un historial de eventos pasados y futuros. Para poder manejar esta funcionalidad, cuenta con un sistema de autentificación.

Adicionalmente, la aplicación cuenta con un sistema de notificaciones que alerta a los usuarios cuando la fecha de un evento se aproxima, cuando hubo un cambio en algún detalle o cuando se suspende.

Aspectos funcionales

- **Gestión de eventos**
  - Crear y editar eventos con título, fecha, ubicación, descripción y otros detalles.
- **Listado de eventos**
  - Ver eventos futuros y pasados.
- **Registro de asistentes**
  - Asociar personas a eventos como asistentes.
- **Roles de usuario**
  - Distinguir entre asistentes y organizadores, y administrar permisos en base a los roles.
- **Interfaz dinámica**
  - Mostrar contenido según el tipo de usuario.
- **Escalabilidad**
  - Permitir agregar funciones nuevas sin romper las ya implementadas.

Entidades clave del dominio

- **Evento**: unidad central de planificación, tiene una fecha, lugar, título, descripción y lista de asistentes.
- **Usuario**: persona que interactúa con el sistema. Puede ser asistente u organizador.
- **Asistente**: usuario que puede ver eventos y registrarse como participante.
- **Organizador**: usuario que puede crear, editar y gestionar eventos.
- **Registro**: asociación entre un usuario asistente y un evento al que asistió.

Capas

En términos conceptuales, el funcionamiento de la aplicación está dividido en cuatro capas:

1. **Capa de dominio**: maneja la lógica de negocio y los modelos.
2. **Capa de persistencia**: maneja el almacenamiento de archivos e información.
3. **Capa de aplicación/servicio**: procesa las acciones solicitadas por el usuario.
4. **Capa de presentación**: lee inputs del usuario y muestra la información.

Aspectos no funcionales

- **Modularidad**: el sistema debe estar dividido en módulos claramente definidos, con una única función asignada. Los módulos deben comunicarse entre sí, sin la necesidad de conocer detalles del otro.
- **Escalabilidad**: el sistema debe seguir el open/closed principle, manteniéndose abierto a extensiones pero cerrado a modificaciones.
- **Accesibilidad**: la aplicación debe ser fácil de usar, clara e intuitiva por los diferentes usuarios.
- **Separación de responsabilidades**: cada parte del sistema debe ocuparse de una funcionalidad única, sin mezclar otras. Es decir, el modelo de datos, la interfaz gráfica y las reglas de negocio no deben tener interdependencia.

Requerimientos y casos de uso

Requerimientos funcionales

ID	Descripción
RF01	El sistema debe permitir crear eventos con título, fecha, ubicación y descripción.
RF02	El sistema debe permitir modificar los datos de un evento existente.
RF03	El sistema debe permitir eliminar eventos existentes.
RF04	El sistema debe mostrar una lista de eventos futuros y pasados.
RF05	El sistema debe registrar qué personas asistieron a cada evento.
RF06	El sistema debe distinguir entre usuarios asistentes y organizadores.
RF07	El sistema debe mostrar funcionalidades y vistas según el rol de usuario (asistente u organizador).
RF08	El sistema debe autenticar usuarios mediante login con nombre de usuario y contraseña.
RF09	El sistema debe cargar al iniciar la información de usuarios, eventos y registros de asistencia.
RF10	El sistema debe persistir cualquier cambio en usuarios, eventos y registros de asistencia.
RF11	El sistema debe validar los datos ingresados al crear o modificar un evento (campos obligatorios, fecha válida).
RF12	El sistema debe impedir que un asistente modifique o elimine un evento (solo organizadores pueden hacerlo).
RF13	El sistema debe permitir al asistente ver únicamente los eventos en los que está registrado.
RF14	El sistema debe enviar notificaciones a los asistentes registrados cuando un evento cambie (fecha, datos) o se cancele.

Requerimientos no funcionales

ID	Descripción
RNF01	El sistema debe estar organizado en módulos desacoplados para facilitar la extensión y el mantenimiento.
RNF02	El sistema debe poder escalar para soportar la incorporación de nuevas funcionalidades (nuevos tipos de eventos, roles o fuentes de datos).
RNF03	El sistema debe ofrecer una interfaz clara e intuitiva, con feedback inmediato ante las acciones del usuario.
RNF04	El sistema debe garantizar tiempos de respuesta adecuados.
RNF05	El sistema debe contar con un mecanismo de notificaciones modular y extensible (p. ej. pop-ups, correo, mensajes internos) para futuros canales.
RNF06	El sistema debe entregar las notificaciones de cambios de evento de forma oportuna.

Casos de uso

ID	Título
CU01	Crear evento
CU02	Modificar evento
CU03	Eliminar evento
CU04	Listar eventos futuros y pasados
CU05	Registrar asistencia de usuario a evento
CU06	Autenticar usuario (login)
CU07	Gestionar roles de usuario
CU08	Mostrar interfaz según rol de usuario
CU09	Cargar datos iniciales de usuarios y eventos
CU10	Persistir cambios en usuarios y eventos
CU11	Validar datos al crear o modificar evento
CU12	Controlar permisos de edición y eliminación
CU13	Filtrar eventos según el usuario asistente
CU14	Notificar cambios o cancelaciones de eventos

Arquitectura

La aplicación se basa en un esquema de capas y está apoyada por dos patrones de diseño: MVP (Model–View–Presenter) para la interfaz, y un adaptador de persistencia (Pattern Adapter/Strategy) que centraliza la lectura y escritura de datos (JSON ahora, base de datos en el futuro).

Capas de la aplicación

1. Capa de presentación (UI)
    a. Contiene ventanas y paneles Swing.
    b. Se encarga únicamente de mostrar datos y leer acciones del usuario.
    c. No realiza lógica de negocio ni maneja datos.
2. Capa de aplicación / servicio
    a. Coordina los casos de uso, como puede ser crear un evento, listar eventos futuros o passados, registrar asistencia, etc.
    b. Invoca o llama validaciones y reglas de negocio definidas en el dominio.
    c. No conoce detalles de la UI ni del manejo de datos.
3. Capa de dominio
    a. Define las entidades clave (Evento, Usuario, Asistente, Organizador, etc) y sus comportamientos.
    b. Contiene las reglas de negocio puras, totalmente desacopladas de la UI y el almacenamiento de datos.
4. Capa de persistencia
    a. Implementa la lectura y escritura de información en archivos JSON.
    b. Proporciona una interfaz común para el resto del sistema, de forma que otros componentes no necesiten conocer el formato de los archivos.

Patrón MVP (Model View Presenter)

Dado que estamos utilizando Swing, utilizar el patrón MVP es la mejor opción. Dada la complejidad de la tarea a realizar, elegir un patrón de diseño es fundamental para el correcto cumplimiento de los requerimientos.

Roles MVP

- **Model**: son las interfaces de acceso a datos y objetos de dominio que representan los datos a mostrar o modificar durante la ejecución del programa.
- **View**: son las clases de Swing, que utilizan eventos y callbacks.
- **Presenter**: se suscribe a los eventos de la View, la conecta con la capa de aplicación para ejecutar los casos de uso y recibe los resultados de la modificación de la información para pasarlos a la View.

Adaptador de persistencia

El patrón de adaptador para la persistencia actúa como una capa intermedia que desacopla la lógica de negocio de los detalles concretos de almacenamiento. Gracias a él, la aplicación puede escalar sin modificaciones profundas: hoy lee y escribe datos en archivos JSON locales, pero mañana puede conectar con MongoDB u otro servicio simplemente intercambiando la implementación del adaptador. De este modo, se preserva la coherencia del modelo de

dominio, se facilita el mantenimiento y se garantiza que el formato elegido (JSON) sea ya compatible con una futura base de datos documental.

Implementación

Diseño de clases

Clases del Dominio

- Evento
- Usuario (abstracta)
- Organizador (extiende Usuario)
- Asistente (extiende Usuario)

Clases de Aplicación / Servicio

- AuthService
- CancelarAsistenciaService
- CrearEventoService
- EliminarEventoService
- ListarUsuariosService
- ListarEventosService
- ModificarEventoService
- NotificarService
- RegistrarAsistenciaService

Clases de Infraestructura

- StorageAdapter (interfaz)
- JsonStorageAdapter (implementa StorageAdapter)
- EmailAdapter (interfaz)
- SmtpEmailAdapter (implementa EmailAdapter)

Clases Interfaz de Usuario (MVP)

View:

- LoginView
- DashboardView
- CrearEventoView
- DetalleEventoView

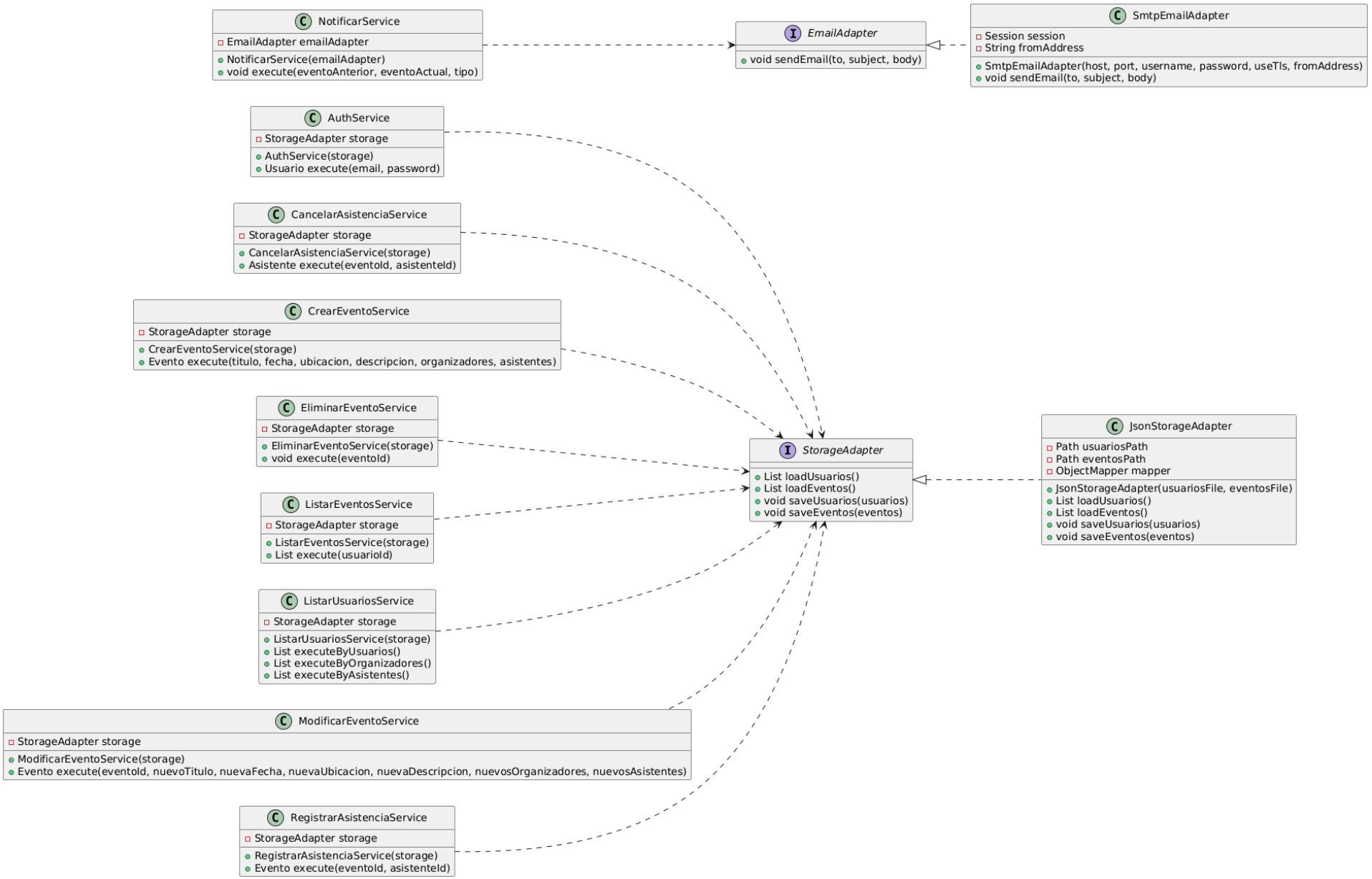
Presenter:

- LoginPresenter
- DashboardPresenter
- CrearEventoView
- DetalleEventoPresenter

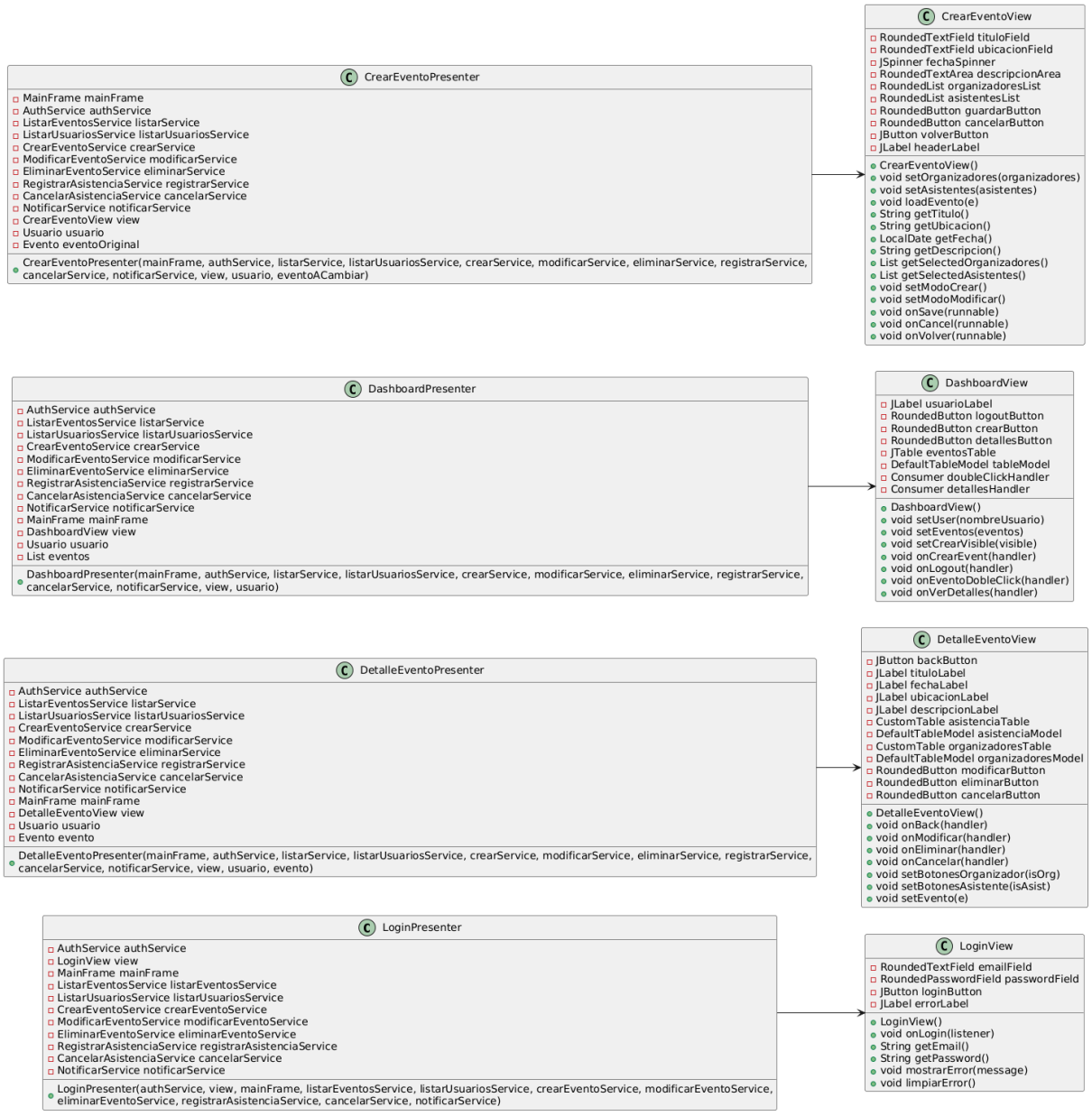
Diagrama de clases

Dominio

Aplicación / Servicio e Infraestructura



Interfaz de usuario



Clases de utilidades y elementos personalizados

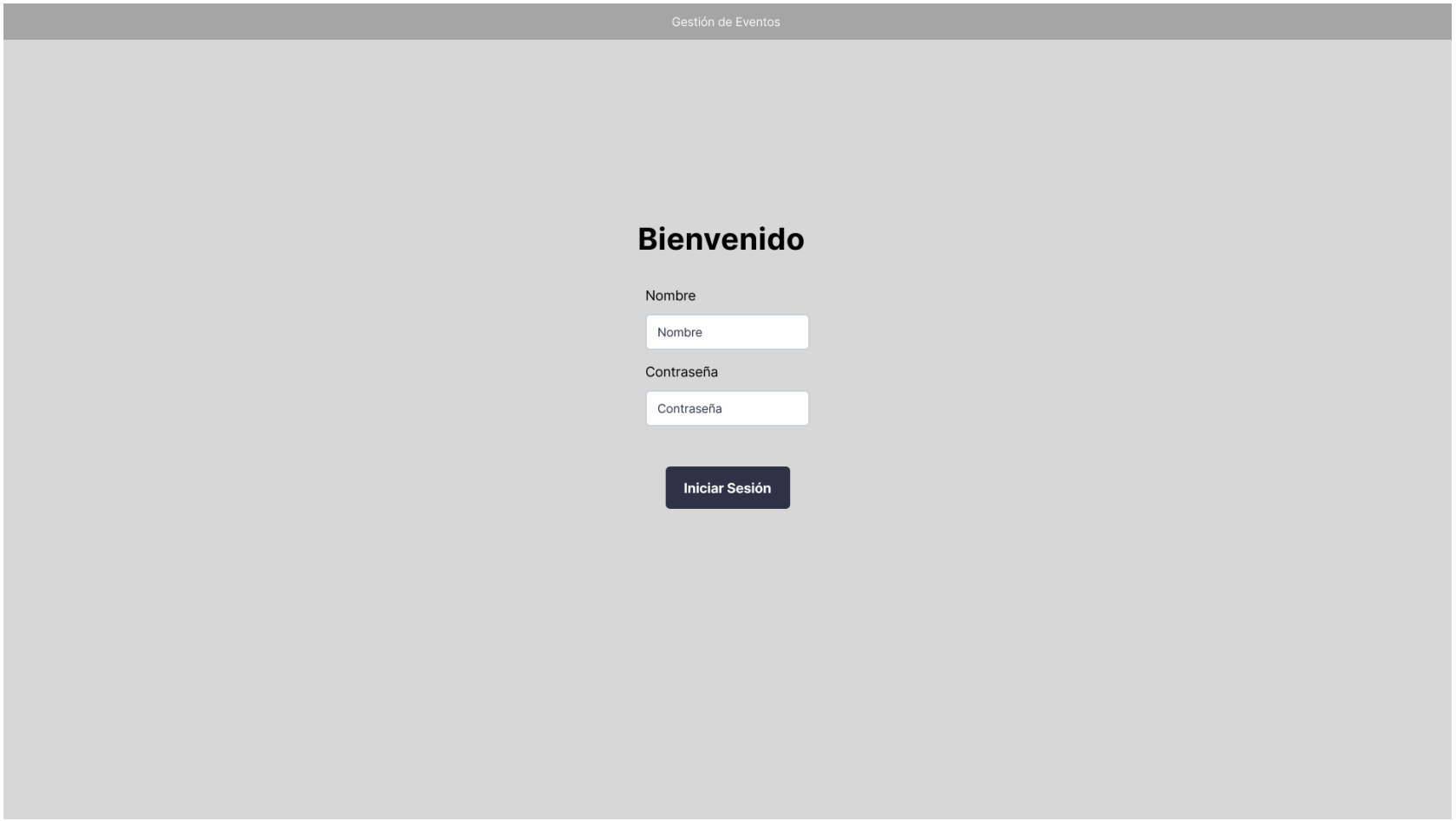
Para implementar correctamente las funcionalidades del sistema, fue necesario incorporar una serie de clases auxiliares que no fueron incluidas en los diagramas de clases. Algunas de ellas se encargan de aspectos funcionales como el manejo de constantes o la transferencia de datos entre capas, mientras que otras están orientadas a mejorar la interfaz gráfica, extendiendo componentes de Swing para lograr una estética más interesante que el estándar. A continuación se describen brevemente estas clases:

- **MainFrame**: clase principal de la interfaz gráfica. Administra la ventana principal de la aplicación y permite mostrar dinámicamente los distintos paneles que componen la vista mediante un CardLayout.
- **TipoNotificacion**: enum utilizado para distinguir los tipos de notificaciones generadas ante acciones sobre eventos, siendo éstas: creación, modificación o eliminación.
- **UsuarioDto**: objeto de transferencia de datos (DTO) utilizado para adaptar la entidad *Usuario* a un formato compatible con la serialización en JSON. Contiene únicamente los campos necesarios para persistir y recuperar correctamente los datos sin acoplarse a la lógica de negocio de la entidad.
- **EventoDto**: objeto de transferencia de datos (DTO) utilizado para representar la información de un evento en una estructura apta para ser serializada en JSON.
- **CustomTable**: extensión de JTable que adapta la apariencia visual de las tablas para entonar con el resto de elementos.
- **RoundedButton**: extensión de JButton para crear botones con bordes redondeados, alineados con el estilo visual general de la aplicación.
- **RoundedList**: extensión de JList, con modificaciones estéticas y de comportamiento para ajustarse a las necesidades visuales y funcionales del sistema.
- **RoundedPasswordField**: extensión de JPasswordField con bordes redondeados, utilizada para ingresar contraseñas con caracteres censurados.
- **RoundedSpinner**: extensión de JSpinner que aplica bordes redondeados para mantener la coherencia visual en los formularios. Utilizada para la selección de fecha al momento de crear o editar un evento.
- **RoundedTextArea**: extensión de JTextArea que redondea los bordes y modifica el relleno interior para que se adapte a la estética de la aplicación.
- **RoundedTextField**: extensión de JTextField con bordes redondeados para una presentación uniforme.
- **ToggleSelectionModel**: modelo de selección para listas (ListSelectionModel) que permite seleccionar y deseleccionar ítems con un solo click, haciendo más intuitiva la interacción del usuario.
- **UIConstants**: clase que contiene constantes de configuración visual como colores, tamaños de fuente y dimensiones generales de los componentes, facilitando la consistencia estética, el mantenimiento del código y futuras iteraciones estéticas.

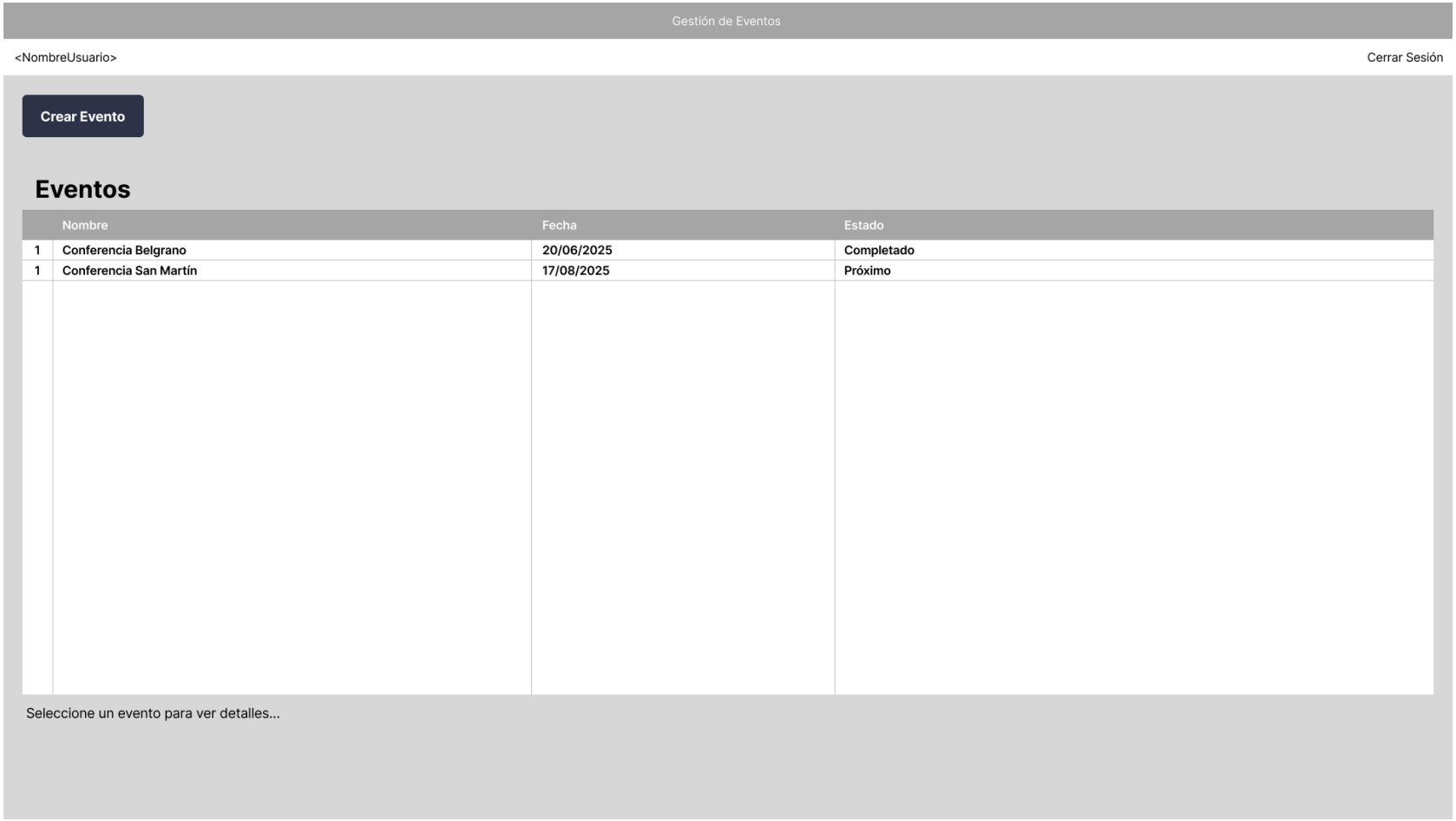
Prototipos

Los prototipos están diseñados a modo de wireframe, el objetivo principal es determinar la estructura de cada pantalla sin entrar en detalles estéticos. Debido a detalles de implementación, algunas disposiciones terminaron cambiando.

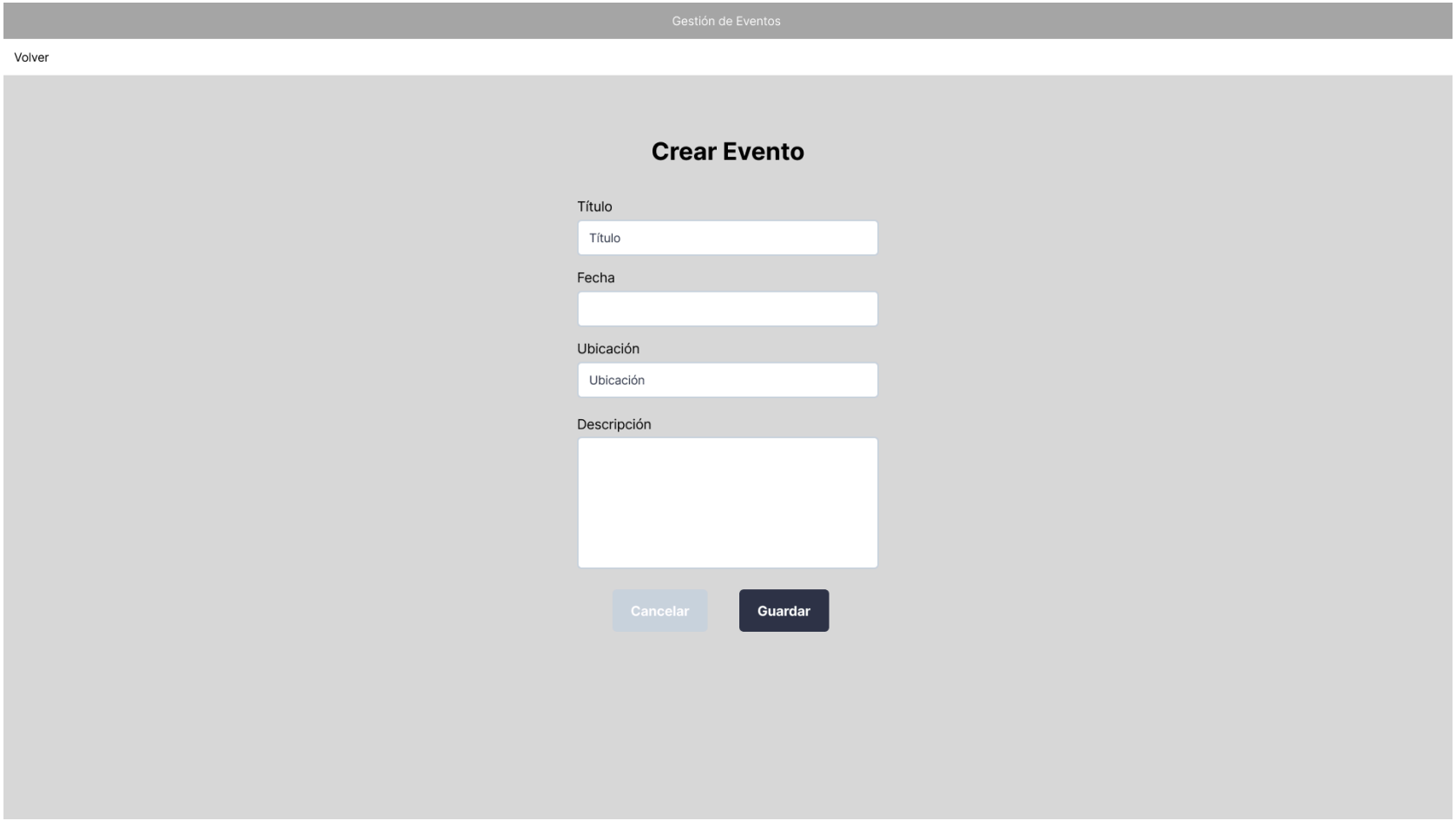
Login



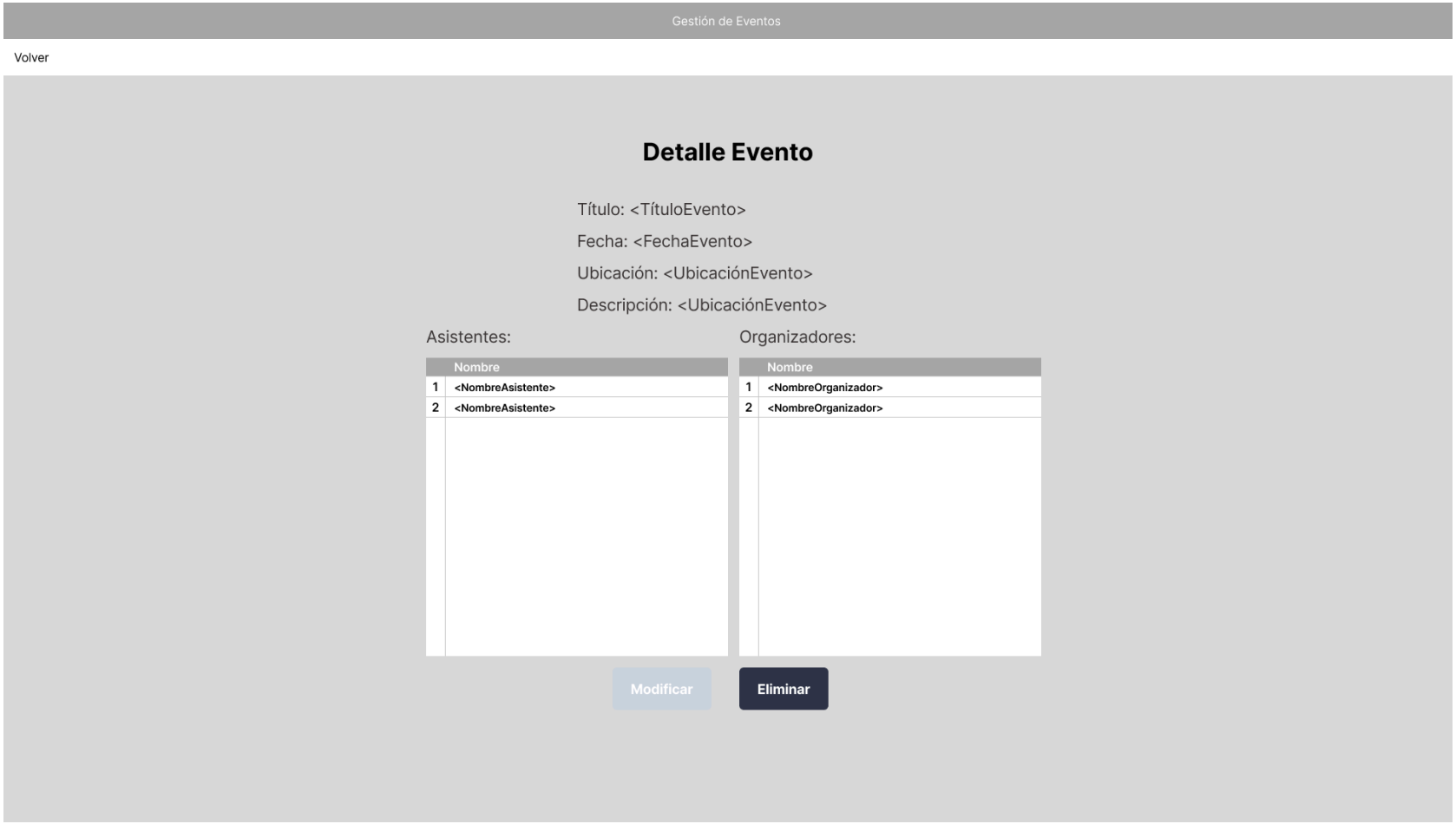
Dashboard


Gestión de Eventos
<NombreUsuario>
Cerrar Sesión
Crear Evento
Eventos
Nombre
Fecha
Estado
1
Conferencia Belgrano
20/06/2025
Completado
1
Conferencia San Martín
17/08/2025
Próximo
Seleccione un evento para ver detalles...

Crear Evento



Detalle Evento



Guía de prueba de la aplicación

La aplicación sigue un flujo definido: el usuario siempre inicia en la pantalla de inicio de sesión y, una vez autenticado, navega por distintas secciones según su rol. Para facilitar la evaluación del trabajo, a continuación se detalla un breve instructivo de uso y las credenciales necesarias para ingresar.

Al iniciar, se presenta la pantalla de login. Para acceder al sistema, el usuario debe ingresar un par válido de credenciales. Estas credenciales están almacenadas en el archivo *usuarios.json,* ubicado en el directorio *data*. Allí se encuentra el hash de la contraseña, no la contraseña en texto plano.

Para realizar pruebas con distintos tipos de usuarios, se pueden utilizar los siguientes datos:

Usuario Organizador

- **Email**: ana.perez@example.com
- **Contraseña**: hola123

Usuario Asistente

- **Email**: juan.gomez@example.com
- **Contraseña**: hola123

Ciertamente, las contraseñas de todos los usuarios son *hola123*.

Una vez validadas las credenciales, la navegación dentro de la aplicación es lo suficientemente intuitiva como para no requerir un instructivo detallado. Sin embargo, es importante destacar que los distintos tipos de usuarios cuentan con funcionalidades diferenciadas.

Puntos de mejoras futuras

La aplicación fue desarrollada con una arquitectura que prioriza la **escalabilidad** y **mantenibilidad**, permitiendo su evolución sin necesidad de reescrituras profundas.

Algunas de las mejoras a futuro que se podrían implementar son:

- **Persistencia con base de datos**: actualmente, la información se almacena en archivos JSON. La estructura del sistema permite reemplazar esta lógica por una basada en base de datos relacional o NoSQL mediante la implementación de un nuevo *StorageAdapter*, sin necesidad de modificar el resto de la lógica de negocio.
- **Envío de notificaciones reales**: el sistema de notificaciones puede adaptarse fácilmente para implementar notificaciones reales por correo electrónico utilizando

SmtpEmailAdapter. Actualmente esta funcionalidad no está aplicada ya que todos los usuarios son ficticios para poder hacer pruebas.

- **Mejoras estéticas en la interfaz**: si bien la interfaz gráfica cumple su función, su diseño visual es básico. La implementación actual, sin embargo, permite realizar mejoras estéticas (paleta de colores, estilos, tipografías) de forma localizada, sin afectar la funcionalidad general.
- **Control de permisos dinámico**: a futuro podría ser una opción implementar los roles de usuario en relación a un evento en particular, de forma que un usuario pueda ser *asistente* en un evento y *organizador* en otro.

Conclusión

Este trabajo práctico me permitió integrar y consolidar conocimientos adquiridos a lo largo de la cursada de la materia, combinándolos con mi experiencia previa. Desde el principio, decidí encarar el proyecto como algo más que una simple entrega académica, con la intención clara de que el resultado pudiera incorporarse a mi portfolio y reflejar mi manera de trabajar.

Elegir una arquitectura escalable y aplicar principios SOLID fue una decisión consciente y necesaria. Aprendí de experiencias anteriores que mantener una buena organización del código es fundamental para evitar problemas durante el desarrollo y facilitar la resolución de errores. Aunque entiendo que patrones de diseño y arquitecturas complejas suelen abordarse en etapas más avanzadas de la carrera, no implementarlas habría complicado significativamente completar un trabajo con esta calidad en un plazo razonable.

Además, integrar herramientas profesionales como GitHub, no solo para la entrega del proyecto sino también para versionar y documentar mi trabajo, fue clave. Esta decisión me permitió sentirme más cerca de un entorno profesional real, algo que no siempre se logra en contextos académicos.

Durante el desarrollo enfrenté varios desafíos técnicos y organizativos, especialmente al utilizar Swing en profundidad sin amplia experiencia. En mi entorno profesional utilizo Unity para desarrollar videojuegos, noté rápidamente diferencias y similitudes entre Unity y Swing, de manera que pude ir adaptando mi flujo de trabajo y ampliando mi comprensión de Swing.

También encontré similitudes entre Swing y bibliotecas modernas de desarrollo frontend como React, particularmente en el manejo de estados y eventos en las interfaces gráficas. Aunque Swing maneja estos aspectos de forma más manual y verbosa, la experiencia previa con React me ayudó a entender más fácilmente cómo debía estructurar correctamente la interacción entre componentes visuales y la lógica interna.

Finalmente, ver el sistema funcionando correctamente, con una separación clara entre capas, funcionalidades robustas y un potencial concreto de escalabilidad y mejora, resultó muy gratificante tanto en lo técnico como en lo personal.

En conclusión, este proyecto fue más que un requisito académico: representó una oportunidad concreta para ampliar mis conocimientos, comparar diferentes paradigmas de desarrollo, integrar experiencias previas, ocuparme con algo que me apasiona hacer y sumar una pieza significativa a mi portfolio profesional.