202423975 – Manuel Mejía

202420898 – Daniel Rincón

202420166 – Isabela Archbold

Documento de Diseño

Índice – Proyecto 1

9. **Diagramas de secuencia**
    9.1. Compra de tiquetes
    9.2. Transferencia de tiquetes
        9.3. Creación de eventos
        9.4. Cancelación de eventos
        9.5. Generación de reportes
10. **Información sobre la persistencia**
        10.1. Estructura de carpetas
        10.2. Formato de los archivos
        10.3. Esquema de datos
        10.4. Ejemplo de persistencia
11. **Alternativas de solución consideradas**
        11.1. Opciones evaluadas
        11.2. Razones para no implementarlas
12. **Preocupaciones y riesgos de la solución**
        12.1. Riesgos técnicos
        12.2. Riesgos funcionales
        12.3. Riesgos de mantenimiento
13. **Conclusiones**
        13.1. Evaluación general del diseño
        13.2. Proyecciones para la siguiente entrega

1. Contexto del Problema

1.1 Descripción General del Problema

El proyecto consiste en el diseño e implementación de una plataforma para la compra, gestión y administración de eventos de diferentes tipos. El sistema debe permitir la interacción de tres actores principales:

- Clientes naturales, quienes ingresan al sistema para buscar, cotizar y comprar tiquetes de eventos específicos.
- Organizadores, responsables de la creación y administración de eventos, manejo de venues, definición de precios y configuración de ofertas.
- Administradores, encargados de la gestión general del sistema, incluyendo control financiero, sobrecargos, cancelaciones y supervisión global de las operaciones.

La función principal del sistema es la venta de tiquetes, ya que constituye la base operativa y la funcionalidad más utilizada. El sistema debe manejar diferentes tipos de tiquetes y paquetes, adaptándose a los distintos eventos. Toda la información del sistema debe ser persistente, almacenándose en archivos planos (JSON) para garantizar la recuperación de datos y trazabilidad entre ejecuciones.

1.2 Problemática Principal

El principal reto es construir una plataforma integral y flexible capaz de manejar la compra de distintos tipos de tiquetes para eventos diversos, administrados por diferentes organizadores, y bajo la supervisión centralizada de un administrador.

Además, la solución debe garantizar:

- Una estructura lógica y modular para las relaciones entre eventos, usuarios y tiquetes.
- Persistencia eficiente y confiable sin uso de bases de datos externas.
- Capacidad de escalar a nuevas funcionalidades sin comprometer el diseño base.
- Mantenimiento sencillo y extensible para futuras versiones.

1.3 Actores Principales

- **Cliente:** actor principal del sistema. Compra, visualiza y gestiona sus tiquetes dentro de su cuenta personal.
- **Organizador:** genera los eventos, define precios, localidades y administra el venue.
- **Administrador:** supervisa la plataforma, aprueba eventos, gestiona finanzas y ejecuta cancelaciones o reembolsos.

- **Programadores:** responsables de la arquitectura, desarrollo y mantenimiento técnico del sistema.

---

2. Alcance de la Solución

2.1 Funcionalidades incluidas

La versión actual del sistema permite:

- Registro de usuarios (clientes, organizadores, administradores).
- Compra directa de tiquetes desde la plataforma, sin redirecciones externas.
- Historial de tiquetes pasados y actuales para cada usuario.
- Gestión de saldo virtual, que funciona como un sistema de puntos y reembolsos.
- Reembolsos automáticos o manuales, según tres escenarios:
    a) Cancelación del evento por parte del administrador.
    b) Cancelación por parte del organizador (por insolvencia).
    c) Solicitud individual del usuario por calamidad.

Cada tiquete contiene los siguientes atributos:

- Precio, cargo por servicio y cuota de impresión.
- Evento al que pertenece.
- Fecha, hora, número ID y tipo de tiquete.

2.2 Funcionalidades Futuras

La solución está diseñada para ser escalable en términos de persistencia, interfaz, seguridad y tipos de usuario.

En el futuro se espera:

- Integrar bases de datos y autenticación externa (por ejemplo, con Google).
- Incorporar una interfaz gráfica moderna (UI/UX).
- Implementar seguridad avanzada para evitar accesos no autorizados.
- Crear nuevos tipos de tiquetes y usuarios (Premium, VIP, etc.) con beneficios adicionales.

2.3 Supuestos Generales

- Solo la aplicación accede a los archivos de datos.
- Todas las fechas son válidas y dentro del mismo huso horario.
- Los usuarios ingresan credenciales correctas.

- No existe concurrencia ni edición simultánea.
- Los archivos de persistencia siempre existen o se crean al iniciar.
- Toda la información financiera se maneja en una sola moneda (COP).

---

3. Restricciones y limitaciones

3.1 Restricciones/ limitaciones técnicas y operativas

Restricciones técnicas y operativas

1. Lenguaje y entorno de desarrollo: La solución está completamente desarrollada en Java, sin dependencias externas ni integración con APIs. Esto garantiza independencia del entorno y control total sobre la estructura del código.
2. Persistencia basada en JSON: El sistema no utiliza una base de datos tradicional. La información se almacena en archivos planos con formato JSON, lo que permite mantener la simplicidad, legibilidad y dominio total sobre los datos registrados.
3. Ejecución en entorno local: El backend opera únicamente en modo local, sin conexión a servicios web o servidores en la nube. Por tanto, la plataforma no puede ejecutarse en línea ni realizar comunicación externa.
4. Interfaz de usuario mínima: El proyecto se centra en la lógica de negocio y la arquitectura de clases. No incorpora una interfaz gráfica avanzada ni elementos visuales de interacción. La funcionalidad es completamente operativa desde la lógica interna.
5. Sin integración de pagos ni manejo de divisas: La plataforma no incluye pasarelas de pago ni gestión de monedas. El proceso de compra de tiquetes es puramente funcional y no contempla transacciones económicas reales, lo que elimina la necesidad de integrar APIs externas.

Limitaciones Funcionales

1. Tipos de usuarios: El sistema admite únicamente tres roles: Cliente, Organizador y Administrador. Todos los clientes se tratan de manera uniforme; no se contemplan perfiles diferenciados por edad, condición o accesibilidad.
2. Eventos estáticos: Los eventos son simulados y no están vinculados con plataformas o proveedores reales. Las actualizaciones solo se reflejan al recargar los datos, sin sincronización en tiempo real.
3. Ausencia de control físico de acceso: El sistema no administra inventarios físicos ni mecanismos de validación en espacios reales (como torniquetes o códigos QR). Se asume

que el ID del tiquete garantiza su autenticidad dentro del entorno lógico. No se contemplan medidas contra la reventa o duplicación fuera del sistema.

---

4. Objetivos

4.1 Objetivo General

Desarrollar una plataforma que permita crear, gestionar y vender tiquetes de eventos, asegurando la persistencia de la información y la correcta administración por parte de diferentes tipos de usuario.

4.2 Objetivos Especificos

- Implementar un modelo de dominio orientado a objetos que represente de forma precisa las entidades del sistema de boletería (usuarios, eventos, tiquetes, localidades y venues).
- Desarrollar una lógica de negocio robusta y modular, capaz de manejar los procesos principales del sistema: creación de eventos, venta de tiquetes, transferencias, cancelaciones y reembolsos, cumpliendo las restricciones y reglas establecidas para cada tipo de usuario.
- Diseñar un mecanismo de persistencia estructurado y seguro que permita almacenar la información del sistema en archivos locales (planos o binarios), garantizando la integridad de los datos y su recuperación en futuras ejecuciones del programa.
- Construir programas de demostración funcional que validen el correcto comportamiento del sistema mediante casos de prueba controlados, sin depender de interfaces gráficas, y mostrando el estado de la aplicación a través de consola de manera clara y comprensible.
- Establecer una base escalable para futuras ampliaciones, permitiendo que la estructura actual sirva como punto de partida para incorporar en próximas versiones una interfaz de usuario completa, un sistema de persistencia avanzado (bases de datos o APIs) y funcionalidades extendidas como reportes o análisis de ventas.

---

5. No objetivos

El presente proyecto se enfoca en la implementación funcional del sistema de boletería y gestión de eventos, por lo cual se han definido las características que no forman parte del alcance actual. Estas exclusiones permiten mantener los objetivos realistas y priorizar el desarrollo de la lógica central del sistema.

1. No se busca integrar pasarelas de pago reales: El sistema no contempla la conexión con plataformas externas de pago (como PayU, Stripe o PayPal). Las transacciones son simuladas de manera interna, sin involucrar dinero real ni validación bancaria.
2. No se implementan sistemas avanzados de autenticación y seguridad: El sistema no incluye cifrado de contraseñas, verificación en dos pasos, ni manejo de sesiones seguras. Se asume que los usuarios que ingresan son legítimos y que las credenciales son válidas.
3. No se desarrolla una interfaz gráfica profesional (UI/UX): La interacción con el sistema se realiza únicamente por consola. No se consideran elementos visuales, diseño de experiencia de usuario o componentes gráficos.
4. No se contempla el manejo de múltiples divisas, idiomas o regiones: Todas las operaciones se ejecutan en una sola moneda (COP) y en idioma español. No se incluye conversión automática de valores ni localización del contenido.
5. No se integra control físico o validación presencial de tiquetes.: La verificación de acceso a eventos no depende de códigos QR, torniquetes ni dispositivos externos.

---

6. Elementos de la Solucion

6.1 Descripcion general del diseño

El sistema prioriza la organización y claridad del código. En total cuenta con 15 clases principales dedicadas a la lógica y 6 clases auxiliares para la persistencia.

Entre las más importantes están Usuario, Organizador, Evento, Tiquete y Administrador, que representan los elementos centrales del sistema. Estas clases se comunican mediante un diseño de control delegado, donde cada una asume responsabilidades específicas: por ejemplo, el organizador administra los eventos, el usuario compra tiquetes y el administrador supervisa el funcionamiento general.

La información se guarda en archivos JSON, lo que permite mantener los datos entre ejecuciones sin requerir una base de datos compleja. En conjunto, el diseño logra un sistema modular, escalable y fácil de mantener.

6.2 Clases Principales del proyecto

Cada clase una cumple una función específica dentro de la lógica del programa, permitiendo una interacción ordenada y eficiente entre los diferentes componentes, estas son:

- **Usuario:** Representa al cliente del sistema. Puede registrarse, iniciar sesión, comprar tiquetes y consultar sus compras.

- **Organizador:** Es el encargado de crear y administrar los eventos. Puede definir la información de cada evento, modificarla o eliminarla.
- **Evento:** Contiene todos los datos relacionados con un evento, como nombre, fecha, lugar, capacidad y tiquetes disponibles.
- **Tiquete:** Registra la información de una entrada específica, incluyendo su número, precio, estado y el evento al que pertenece.
- **Administrador:** Supervisa el funcionamiento general del sistema. Puede revisar la información de usuarios, eventos y tiquetes, y realizar ajustes si es necesario.
- **PersistenciaJSON:** Se encarga de guardar y cargar la información de las clases principales en archivos con formato JSON, asegurando que los datos se mantengan entre ejecuciones.

6.3 Metodos y atributos Relevantes

Cada clase del sistema cuenta con un conjunto de atributos y métodos que permiten representar su comportamiento y las relaciones con las demás entidades. A continuación, se describen los más importantes:

Clase Usuario

- Atributos:
  - idUsuario: Identificador único del usuario.
  - nombre: Nombre completo del usuario.
  - correo: Correo electrónico asociado a la cuenta.
  - saldoVirtual: Saldo disponible para compras o reembolsos.
  - tiquetesComprados: Lista de tiquetes adquiridos por el usuario.
- Métodos:
  - comprarTiquete(Evento evento, TipoTiquete tipo): Permite al usuario adquirir un tiquete.
  - consultarHistorial(): Muestra el historial de compras del usuario.
  - solicitarReembolso(Tiquete tiquete): Envía una solicitud de devolución al administrador.

Clase Organizador

- Atributos:
  - idOrganizador: Identificador único.
  - nombre: Nombre del organizador.
  - eventosCreados: Lista de eventos que administra.
- Métodos:
  - crearEvento(String nombre, Date fecha, Venue venue): Registra un nuevo evento en el sistema.

    - cancelarEvento(Evento evento): Permite cancelar eventos propios.
    - modificarEvento(Evento evento): Actualiza información de un evento existente.

Clase Evento

- Atributos:
    - idEvento: Identificador del evento.
    - nombreEvento: Nombre o título del evento.
    - fecha: Fecha del evento.
    - hora: Hora de inicio.
    - venue: Lugar donde se realizará.
    - tiquetesDisponibles: Lista de tiquetes que pueden ser adquiridos.
- Métodos:
    - generarTiquetes(int cantidad, double precio): Crea los tiquetes asociados al evento.
    - obtenerDisponibilidad(): Retorna el número de tiquetes aún disponibles.
    - calcularIngresos(): Calcula el total recaudado por el evento.

Clase Tiquete

- Atributos:
    - idTiquete: Código único del tiquete.
    - precioBase: Valor original del tiquete.
    - cargoServicio: Tarifa adicional por emisión.
    - evento: Referencia al evento asociado.
    - estado: Indica si el tiquete está activo, usado o reembolsado.
- Métodos:
    - calcularPrecioFinal(): Retorna el precio total (base + cargos).
    - transferir(Usuario nuevoPropietario): Transfiere el tiquete a otro usuario.
    - reembolsar(): Gestiona el proceso de devolución del valor del tiquete.

Clase Administrador

- Atributos:
    - idAdministrador: Identificador del administrador.
    - nombre: Nombre del administrador.
    - eventosSupervisados: Lista de eventos bajo su control.
- Métodos:
    - aprobarReembolso(Tiquete tiquete): Autoriza o rechaza una devolución.
    - cancelarEvento(Evento evento): Cancela un evento y gestiona reembolsos.
    - generarReporteGeneral(): Produce informes de ventas, cancelaciones y reembolsos.

Clase PersistenciaJSON

- Atributos:
  - rutaArchivo: Ubicación donde se guarda la información.
- Métodos:
  - guardarDatos(Object objeto, String tipo): Almacena los datos de una clase en formato JSON.
  - cargarDatos(String tipo): Recupera la información almacenada en los archivos.
  - actualizarArchivo(): Reescribe la información para mantenerla sincronizada.

6.4 Relaciones entre Clases

El diseño del sistema establece relaciones entre las clases principales que reflejan la dinámica real de una plataforma de boletería.

- **Relación Usuario – Tiquete:** Un Usuario puede poseer varios Tiquetes (relación uno a muchos). Cada tiquete pertenece a un único usuario, lo que permite rastrear las compras y gestionar transferencias o reembolsos individuales.
- **Relación Organizador – Evento:** Cada Organizador puede crear y administrar múltiples Eventos. A su vez, un evento solo puede tener un organizador responsable. Esta relación es clave para la gestión de la oferta de eventos dentro de la plataforma.
- **Relación Evento – Tiquete:** Un Evento contiene una lista de Tiquetes asociados (uno a muchos). Cada tiquete hace referencia directa al evento al que pertenece, lo que permite conocer disponibilidad, ingresos y control de acceso.
- **Relación Administrador – Evento / Tiquete / Organizador:** El Administrador mantiene una relación jerárquica con las demás clases. Supervisa los Eventos, aprueba o rechaza reembolsos de tiquetes, y puede cancelar eventos creados por los organizadores. Su rol garantiza el cumplimiento de las políticas del sistema.
- **Relación PersistenciaJSON – Todas las clases:** La clase PersistenciaJSON actúa como un módulo independiente de almacenamiento. No participa en la lógica del negocio directamente, pero se comunica con todas las demás clases para guardar y recuperar su información en archivos locales.

---

7. Diagrama de Clases de Alto Nivel

7.1 Estructura General

El diagrama de clases tiene una separación clara entre las capas de lógica de negocio y persistencia de datos.

En la capa lógica se encuentran las clases que gestionan las operaciones principales:

- Usuario, clase base de la cual derivan los roles específicos del sistema.
- Cliente, Organizador y Administrador, que heredan de *Usuario* y definen las acciones particulares de cada tipo de actor dentro de la plataforma.
- Evento y Tiquete, encargadas de manejar la información relacionada con la oferta y compra de entradas.

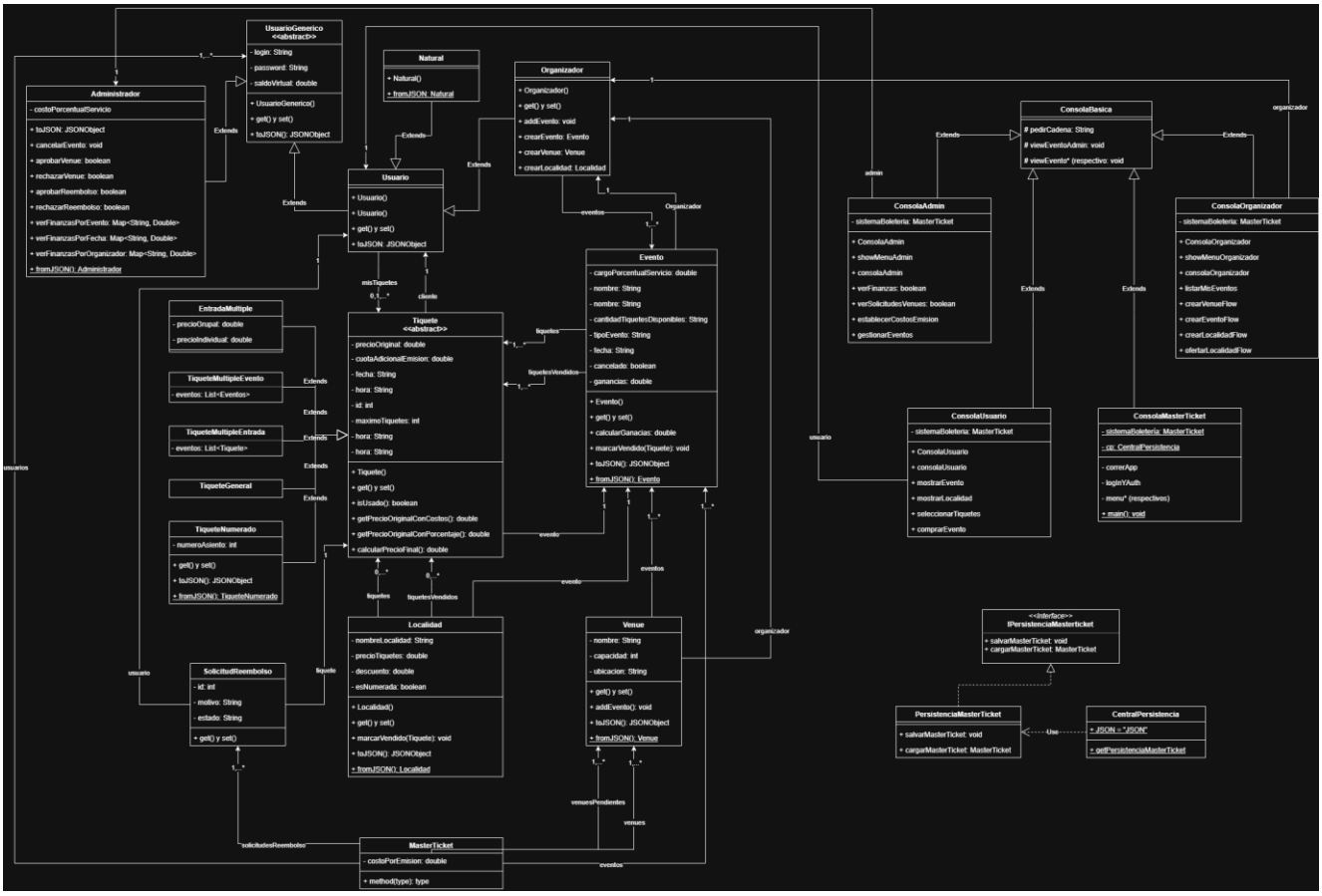
Por su parte, la capa de persistencia está compuesta por la clase PersistenciaJSON, responsable de almacenar y recuperar la información de los objetos en archivos locales.

7.2 Relaciones Principales

Las relaciones entre las clases reflejan el flujo natural de interacción dentro de una plataforma de gestión de eventos y tiquetes:

- **Herencia:** Las clases Cliente, Organizador y Administrador heredan de Usuario, compartiendo atributos básicos como nombre, correo y contraseña, pero implementando métodos distintos según sus funciones.
- **Asociación Usuario–Tiquete:** Un Cliente puede tener múltiples Tiquetes asociados (relación uno a muchos). Cada tiquete pertenece a un solo cliente, lo que permite rastrear su historial de compras.
- **Asociación Organizador–Evento:** Un Organizador puede crear y administrar varios Eventos, mientras que cada evento tiene un único organizador responsable.
- **Asociación Evento–Tiquete:** Cada Evento posee una colección de Tiquetes vinculados, que representan las entradas disponibles para dicho evento.
- **Asociación Administrador–Sistema:** El Administrador tiene acceso a la supervisión global del sistema, gestionando la información de usuarios, eventos y tiquetes para asegurar la coherencia operativa.
- **Dependencia PersistenciaJSON–Clases principales:** La clase PersistenciaJSON mantiene una relación de dependencia con todas las clases del dominio, ya que es la encargada de guardar y recuperar sus datos en formato JSON.

8. UML



El UML anterior muestra el funcionamiento total de la solución, incluyendo las clases más importantes y sus relaciones (1), atributos y métodos empleados (2), el modelo de la consola/interacción con el usuario (3) y el manejo de la persistencia (4).

9. Diagramas de Secuencia

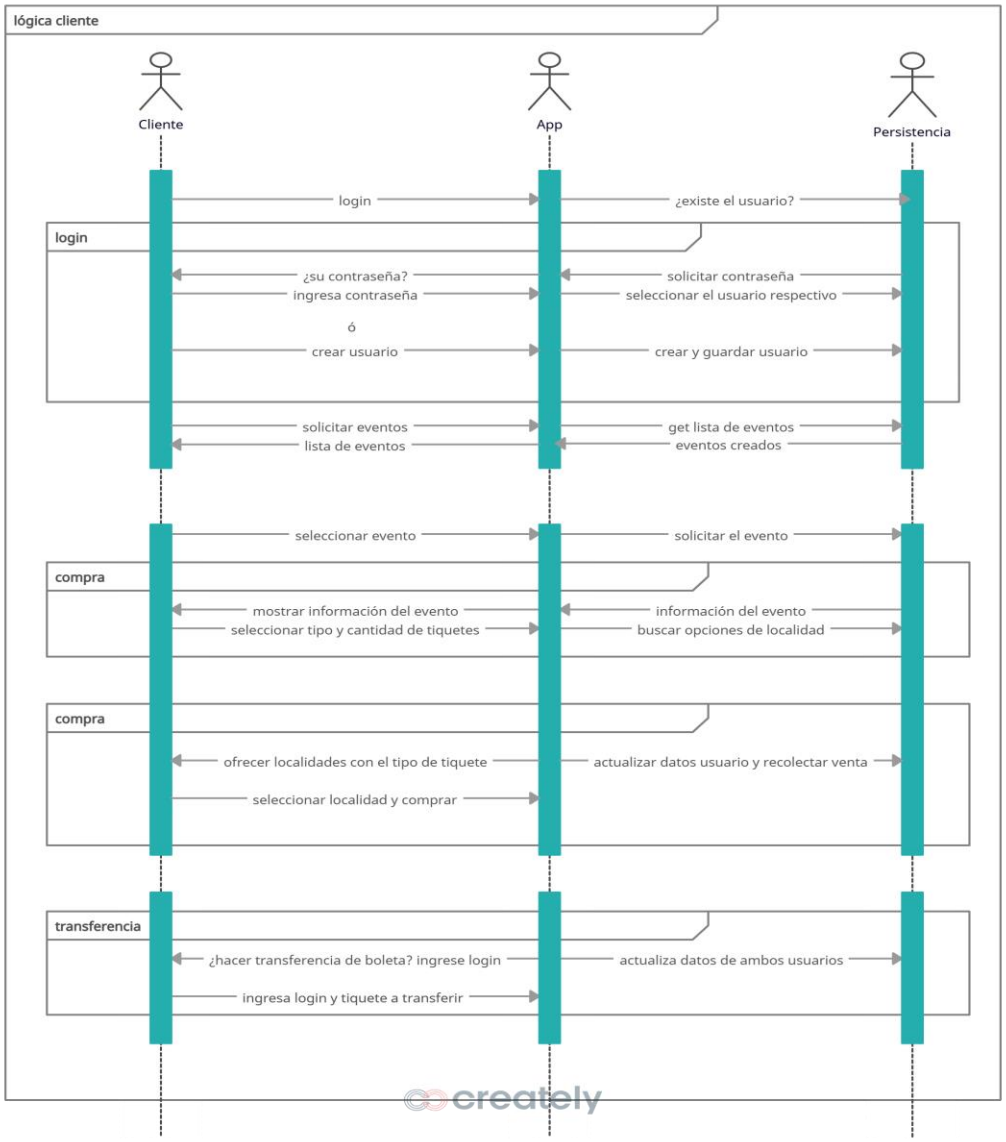


Figura 1: Diagrama de Secuencia del Cliente

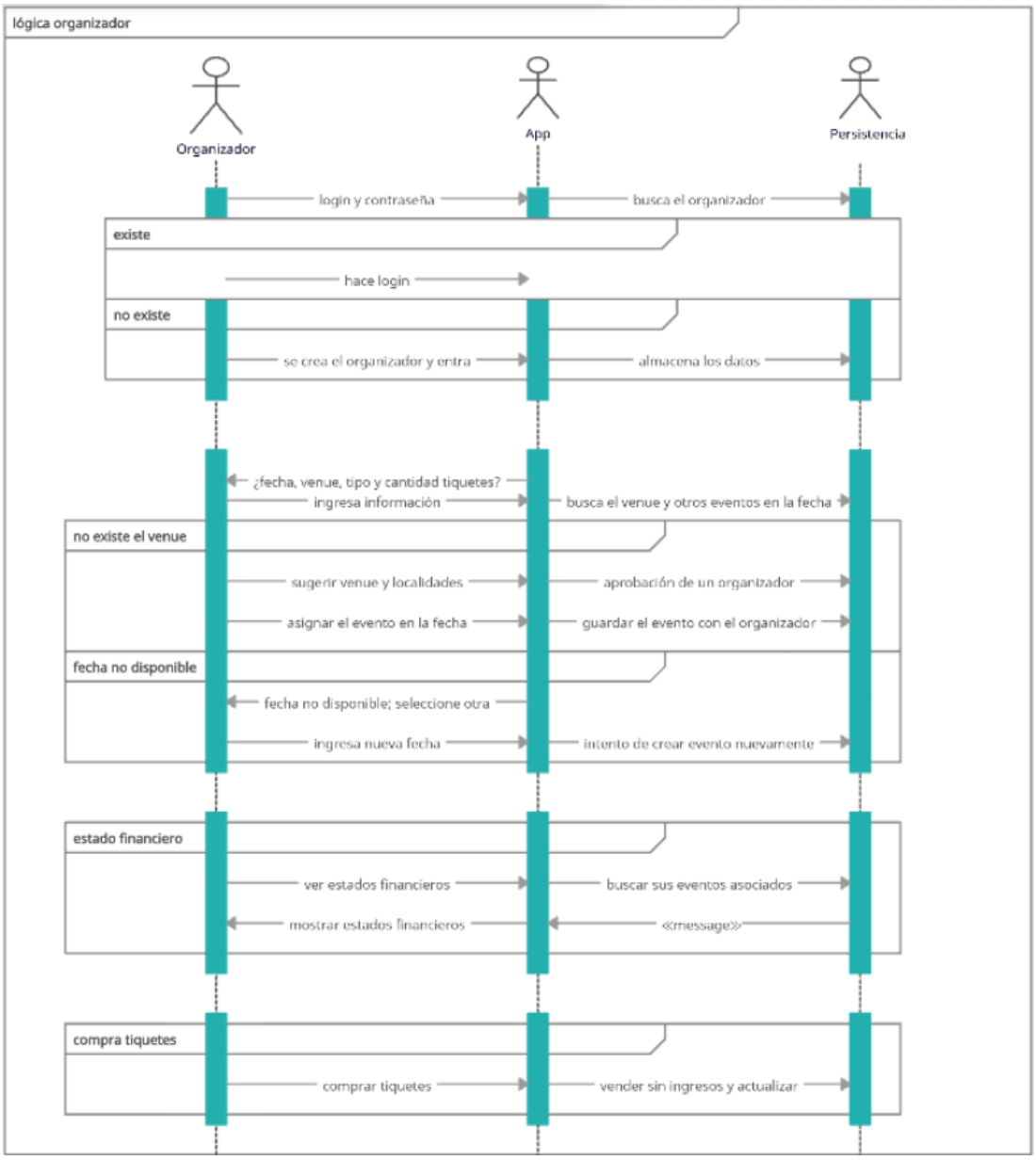


Figura 2: Diagrama de Secuencia del Organizador

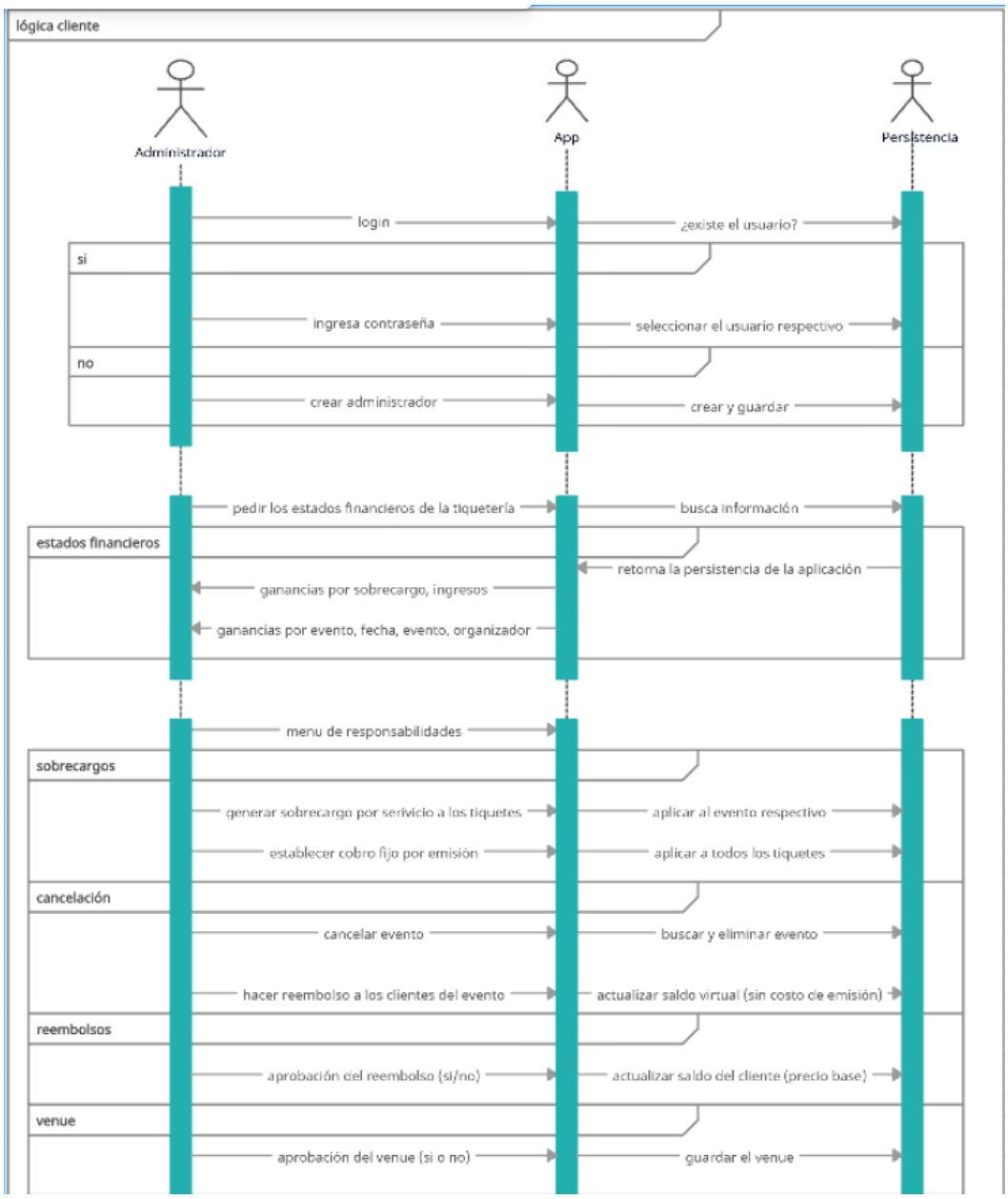


Figura 3: Diagrama de Secuencia del Adinistrador

10. Información sobre la persistencia.

10.1. Estructura de carpetas

La aplicación utiliza un esquema de persistencia local basado en archivos JSON. Toda la información se almacena en una carpeta denominada **`/datos`**, ubicada al mismo nivel del código fuente pero **fuera de la carpeta `/src`**, con el fin de mantener una separación clara entre la lógica del programa y los datos persistentes.

Dentro de esta carpeta se genera automáticamente el archivo **`masterticket.json`**, que actúa como punto central de persistencia del sistema. Si la carpeta o el archivo no existen al momento de ejecutar el programa, el método `saveDefault()` de la clase `CentralPersistencia` los crea de forma automática.

10.2. Formato de los archivos

El sistema utiliza el formato **JSON** (JavaScript Object Notation) para almacenar toda la información del dominio. La serialización se realiza mediante la biblioteca **`org.json`**, utilizando las clases `JSONObject` y `JSONArray`.

El formato elegido permite representar jerárquicamente todos los objetos del sistema (usuarios, eventos, venues, localidades, tiquetes, etc.) en un único documento. Los datos se escriben con **sangría ("pretty print")**, facilitando la lectura manual y el versionamiento en Git.

Los principales tipos de datos almacenados son:

- **Strings:** para nombres, logins, contraseñas, identificadores y fechas (ISO 8601, ej. `"2025-10-22T18:30"`).
- **Numbers:** enteros para identificadores y contadores; `double` para precios, porcentajes y saldos.
- **Booleans:** para campos de estado (`cancelado`, `usado`, `esNumerada`, etc.).

El uso de JSON permite mantener una estructura autoexplicativa y portable, compatible con otros lenguajes o servicios externos en futuras version

}

11. Alternativas de solución consideradas

11.1 Opciones Evaluadas

Durante la fase de diseño se consideraron varias alternativas para implementar la persistencia, la estructura del sistema y la gestión de datos, con el fin de seleccionar la opción más equilibrada entre complejidad y funcionalidad. En general, durante la fase de ideación, la principal alternativa a la solución dada fue una base de datos relacional; se consideró su uso para almacenar usuarios, eventos y tiquetes. Esta opción ofrece robustez y escalabilidad, pero fue descartada debido a que el alcance del proyecto exige ejecución completamente local, sin dependencias externas ni configuración de servidores.

12. Preocupaciones y riesgos de la solución

12.1 Riesgos Técnicos

- Corrupción de archivos JSON: Si el archivo de persistencia se modifica manualmente o no se cierra correctamente durante la escritura, el sistema podría no poder cargar los datos.
- Escalabilidad limitada: Al no usar una base de datos, el sistema podría ralentizarse si el volumen de datos crece significativamente.

12.2 Riesgos Funcionales

- Errores en transferencias o reembolsos: Si no se validan correctamente las reglas de negocio, podría haber pérdidas de coherencia en los saldos o en la propiedad de tiquetes.
- Dependencia en la precisión del usuario: El sistema asume que los datos ingresados son válidos, por lo que un error humano puede generar resultados inconsistentes.

12.3 Riesgos de Mantenimiento

- Dependencia de archivos locales: El manejo manual de archivos puede complicar las pruebas y la depuración en entornos distribuidos.
- Falta de automatización en pruebas: La validación manual desde consola dificulta la integración continua y el aseguramiento de calidad.

13. Conclusiones

El diseño cumple el "core" del dominio y los objetivos del proyecto:

El modelo orientado a objetos captura correctamente actores, eventos, venues y tiquetes (incluyendo numerados, múltiples y paquetes), junto con reglas de compra, topes por transacción, transferencias y reembolsos. Esto alinea la solución con el objetivo de practicar análisis→diseño→implementación con foco en la lógica funcional, no en UI.

La arquitectura es coherente con el alcance de Entrega 1/2 y facilita demostración por consola:

Se cubren los requerimientos de persistencia, programas de demostración y documentación de diseño (diagramas de clases y secuencia). La separación entre P1 (lógica y pruebas por consola) y P2 (interacción completa en consola) se respeta y deja el sistema listo para ampliar interacción sin rehacer la base.

La persistencia en JSON es suficiente para el contexto y mantiene trazabilidad:

Guardar el estado completo en archivos fuera de `/src` cumple con la exigencia técnica, mantiene los datos legibles/versionables y permite reproducibilidad de casos de prueba. Es una decisión consciente para priorizar el diseño OO antes que infraestructura de datos.

Gobernanza y políticas están correctamente centralizadas en el Administrador:

Las reglas de sobrecargos, cuotas fijas, aprobación de venues y cancelaciones (con reembolsos parciales/total según causal) están concentradas en el rol de Administrador, lo que simplifica control y auditoría financiera inicial del sistema.

El modelo de negocio del Organizador está bien soportado:

El Organizador puede crear eventos, definir localidades y ofertas temporales; además, consulta estados financieros por evento/localidad, lo que permite iterar sobre pricing y ocupación desde el día uno.