Mariana Rodríguez 202421258
Carlos García 202321911
Nicolás Ascencio 202420192

|Documento de Análisis - Proyecto 1

---

1. Introducción

1.1 Descripción

BoletaMaster es una plataforma digital diseñada para la venta y gestión de tiquetes de eventos en diferentes venues. Permite que organizadores, clientes y administradores interactúen para realizar diferentes interacciones compra de un tiquete o la cancelación de un evento dentro del sistema

1.2 Características Principales

- **Gestión de Eventos**: Creación, configuración y cancelación de eventos
- **Múltiples tipos de tiquetes**: Básico, Múltiple, Temporada, Deluxe
- **Sistema de localidades**: Con o sin numeración de asientos
- **Gestión financiera**: Cálculo de ganancias y comisiones
- **Sistema de ofertas**: Descuentos temporales y promociones
- **Reembolsos**: Por cancelación de eventos o solicitud individual
- **Transferencia de tiquetes**: Entre usuarios registrados

2. Objetivos

2.1 Objetivo General

Diseñar e implementar un sistema que permita la gestión de eventos y tiquetes, facilitando la compra, creación y administración de eventos en una plataforma unificada.

2.2 Objetivos Específicos

1. Permitir que los organizadores creen, configuren y administren eventos y localidades
2. Ofrecer a los clientes la posibilidad de comprar, almacenar y transferir tiquetes digitales
3. Permitir a los administradores aprobar venues, aplicar recargos y gestionar cancelaciones
4. Controlar la validez, disponibilidad y transferibilidad de los tiquetes en tiempo real
5. Implementar un módulo financiero para calcular montos, recargos, descuentos y generar reportes

2.3 No Objetivos

- El sistema no gestiona pagos en línea reales ya que este proceso se ha tercerizado

3. Restricciones del Sistema

3.1 Restricciones de Venues y Eventos

- Cada venue solo puede tener un evento activo por fecha
- Los organizadores solo pueden crear eventos en venues aprobados

3.2 Restricciones de Tiquetes

- Los tiquetes deben pertenecer a una localidad válida del evento
- Cada tiquete debe tener un identificador único en el sistema
- Los clientes no pueden superar el límite por transacción

3.3 Restricciones de Usuarios

- Solo el administrador puede aprobar venues y cancelar eventos
- El organizador no puede cancelar eventos directamente, solo solicitar
- Los tiquetes transferidos deben estar activos, sin usar y ser transferibles
- Los paquetes Deluxe no son transferibles

---

4. Elementos de la solución implementada

4.1 Estructura de Paquetes

Para la solución se decidió separar las clases en 4 paquetes principales: Usuarios, el cual contendrá todos los distintos tipos de usuarios que podrán estar registrados en el ecosistema; Tiquete, en el cual se encuentran todas las clases derivadas de tiquete y además se incluye el paquete deluxe; Evento, en el cual se incluye todo lo relacionado con los eventos presentes dentro del sistema, como el lugar de presentación, fecha o localidades disponibles para el evento; y finalmente se generó un último paquete con un vendedorTiquete, el cual está encargado de controlar y generar los tiquetes con todas las características solicitadas en el documento.

Estas decisiones se tomaron ya que en cada uno de estos paquetes se encuentran las clases relacionadas y que tienen una gran similitud entre sí mismas, ya sea porque son subclases de una clase padre o porque están destinadas a estar en el mismo lugar para el mismo fin, como crear un evento.

5. Descripción Detallada de Clases

5.1 PAQUETE USUARIOS

5.1.1 Clase Usuario (Abstracta)

Descripción: Clase base que representa a cualquier usuario del sistema con funcionalidades comunes de autenticación y gestión de tiquetes dependiendo del tipo de usuario.

Atributos:

- login: String - Identificador único del usuario
- password: String - Contraseña de acceso
- saldo: double - Balance virtual para transacciones
- tiquetesActivos: ArrayList<Tiquete> - Lista de tiquetes poseídos

Métodos:

- getLogin(): String
- setLogin(login: String): void
- getPassword(): String
- setPassword(password: String): void
- getSaldo(): double
- setSaldo(saldo: double): void
- getTiquetesActivos(): ArrayList<Tiquete>
- agregarTiquete(tiquete: Tiquete): void
- removerTiquete(tiquete: Tiquete): boolean
- autenticar(password: String): boolean
- actualizarSaldo(monto: double): void
- validarCredenciales(login: String, password: String): boolean
- transferirTiquete(tiquete: Tiquete, destino: Usuario, password: String): void

La clase Usuario se creó como abstracta porque todos los tipos de usuarios dentro del sistema (Cliente, Organizador y Administrador) comparten ciertos atributos y comportamientos básicos, como el login, la contraseña y la posibilidad de autenticarse. Además, esto garantiza que no se puedan crear usuarios “genéricos” sin un rol definido, ya que cada uno debe pertenecer a una categoría específica con funciones concretas dentro de la plataforma.

5.1.2 Clase Cliente

Descripción: Usuario que puede comprar tiquetes para eventos, transferirlos a otros usuarios y solicitar reembolsos.

Atributos (adicionales a Usuario):

- tiquetesComprados: ArrayList<Tiquete> - Historial de tiquetes adquiridos
- historialTransacciones: ArrayList<String> - Registro de operaciones realizadas

Métodos:

- Cliente(login: String, password: String, saldo: double) - Constructor

- getTiquetesComprados(): ArrayList<Tiquete>
- getHistorialTransacciones(): ArrayList<String>
- comprarTiquete(tipo: String, usuario: Usuario, evento: Evento, localidad: Localidad, cantidad: int): void
- transferirTiquete(emisor: Usuario, receptor: Usuario, tiquete: Tiquete): void
- cancelarTiquete(codigoTiquete: String, nombreEvento: String): void

Restricciones:

- Debe respetar el límite máximo de tiquetes por transacción
- Solo puede transferir tiquetes propios y que sean transferibles

La clase Cliente hereda de Usuario para aprovechar las funciones básicas como el manejo del login, la contraseña y el saldo, pero añade su propio comportamiento enfocado en la compra y gestión de tiquetes. El cliente puede comprar diferentes tipos de tiquetes, transferirlos a otros usuarios y mantener un registro de todas sus transacciones, lo cual facilita la trazabilidad y el soporte en caso de reclamos o reembolsos.

Además, al separar los tiquetesComprados del historial completo, se puede diferenciar fácilmente entre los tiquetes que el cliente posee actualmente y los que ya usó o transfirió. Esto ayuda a mantener un mejor control sobre las compras activas y pasadas dentro del sistema.

5.1.3 Clase Organizador

Descripción: Usuario responsable de crear y gestionar eventos, configurar localidades y monitorear las ventas de sus eventos.

Atributos :

- eventosCreados: ArrayList<Evento> - Lista de eventos creados por el organizador
- solicitudesPendientes: ArrayList<Evento> - Solicitudes de cancelación en espera

Métodos:

- Organizador(login: String, password: String, saldo: double) - Constructor
- getEventosCreados(): ArrayList<Evento>
- getEventosActivos(): List<Evento>
- getSolicitudesPendientes(): ArrayList<Evento>
- agregarEvento(evento: Evento): void
- removerEvento(evento: Evento): void
- crearEvento(nombre: String, fecha: Date, cantBasicos: int, cantMultiples: int, cantDeluxe: int, cargoPorcentual: double, cuotaAdicional: double, maxBasicos: int, maxDeluxe: int, maxMultiples: int, venue: Venue): Evento
- solicitarCancelacion(evento: Evento, motivo: String): boolean
- asignarLocalidad(evento: Evento, localidad: Localidad): void
- configurarLocalidades(evento: Evento, localidades: Localidad[]): void

- crearOferta(evento: Evento, localidad: Localidad, descuento: double, fechaInicio: Date, fechaFin: Date): void
- sugerirVenue(venue: Venue): void
- consultarGananciasTotales(): double
- consultarGananciasPorEvento(evento: Evento): double
- consultarGananciasPorLocalidad(evento: Evento, localidad: Localidad): double
- calcularPorcentajeVentas(evento: Evento): double
- comprarTiqueteComoCortesia(evento: Evento, localidad: Localidad, cantidad: int): void

Restricciones:

- Solo puede crear eventos en venues aprobados por el administrador
- No puede cancelar eventos directamente, solo solicitar cancelación

La clase Organizador hereda de Usuario para conservar las funciones básicas de cuenta, pero se especializa en la creación y gestión de eventos dentro de la plataforma.

este tiene listas separadas de eventosCreados y eventosActivos, lo que permite mantener un historial completo de todos los eventos activos y no activos, lo cual facilita el funcionamiento de ciertos requerimientos como la compra de eventos organizados por sí mismo.

Además, dentro de este se generan funciones que permiten una persistencia de lso datos financieros útiles para una analisis de finanzas

---

5.1.4 Clase Administrador

Descripción: Usuario con permisos especiales para el control financiero y administrativo de toda la plataforma.

Atributos (adicionales a Usuario):

- COSTOFIJOEMISION: int = 5000 - Costo fijo por emisión de cualquier tiquete
- porcentajeServicioPorTipo: Map<String, Double> - Porcentajes de servicio según tipo de evento
- solicitudesCancelacion: List<Evento> - Lista de solicitudes pendientes de revisar

Métodos:

- Administrador(login: String, password: String, saldo: double) - Constructor
- getCostoFijoEmision(): int
- setPorcentajeServicioPorTipo(mapa: Map<String, Double>): void
- getPorcentajeServicioPorTipo(): Map<String, Double>
- definirPorcentajeServicio(tipoEvento: String, porcentaje: double): void
- getPorcentajeServicio(tipoEvento: String): double
- aprobarVenue(venue: Venue): void
- agregarSolicitudCancelacion(evento: Evento): void
- revisarSolicitudesCancelacion(): List<Evento>

- autorizarCancelacionOrganizador(evento: Evento): void
- cancelarEvento(evento: Evento): void
- cancelarEventoOrganizador(evento: Evento): void
- Reembolso(usuario: Usuario, tiquete: Tiquete): boolean
- setOfertaLocalidad(evento: Evento, oferta: double): void
- consultarGananciasGenerales(): double
- consultarGananciasPorFecha(fecha: Date): double
- consultarGananciasPorEvento(evento: Evento): double
- consultarGananciasPorOrganizador(organizador: Organizador): double

Restricciones:

- Es el único usuario que puede aprobar venues
- Define los porcentajes de servicio que aplican globalmente
- Puede cancelar cualquier evento sin autorización previa
- Los porcentajes de servicio deben estar entre 0% y 100%

La clase Administrador hereda de Usuario para mantener la coherencia del sistema de autenticación, aunque su papel dentro de la plataforma es principalmente de control y supervisión.

este es el que organiza las tasas de porcentaje de servicio o los costos fijo de impresion y emision de los tiquetes. Además es el unico que puede cancelar eventos sin autorización

Ademas, los métodos cancelarEvento y cancelarEventoOrganizador permite aplicar diferentes políticas de reembolso dependiendo de quién inicie la cancelación, ya sea que el evento se cancele por el organizador o por el mismo administrador lo cual tendra tasas y politicas de devolucion de dinero dependiendo quien cancele el evento

---

5.2 PAQUETE TIQUETES

5.2.1 Clase Tiquete (Abstracta)

Descripción: Clase base para todos los tipos de tiquetes que maneja el sistema, define la estructura y comportamiento común.

Atributos:

- usuario: Usuario - Propietario actual del tiquete
- id: String - Identificador único del tiquete
- transferible: boolean - Indica si puede ser transferido
- silla: String - Número de asiento (null si no aplica)
- usado: boolean - Estado de uso del tiquete
- localidad: Localidad - Localidad a la que pertenece
- eventos: ArrayList<Evento> - Lista de eventos asociados

Métodos:

- Tiquete(id: String, transferible: boolean, silla: String, localidad: Localidad, evento: Evento, usuario: Usuario) - Constructor
- getUsuario(): Usuario
- setUsuario(usuario: Usuario): void
- getId(): String
- setId(id: String): void
- isTransferible(): boolean
- setTransferible(transferible: boolean): void
- getSilla(): String
- setSilla(silla: String): void
- getLocalidad(): Localidad
- setLocalidad(localidad: Localidad): void
- isUsado(): boolean
- setUsado(usado: boolean): void
- getEventos(): ArrayList<Evento>
- setEventos(eventos: ArrayList<Evento>): void
- getPrecioTiquete(): double - **Método abstracto**
- getPrecio(): double
- +agregarEvento(evento: Evento)
- +removerEvento(evento: Evento)

Restricciones:

- El ID debe ser único en todo el sistema
- No puede usarse después de que el evento haya vencido
- En localidades numeradas, el número de silla debe ser único
- Una vez usado, no puede volver a estado no usado

Esta clase se creo abstracta ya que existen tiquetes con los mismos atributos pero distintas características, como el número de entradas que genera un tiquete, cosas adicionales o incluso el derecho a la entrada en distintos eventos.

El atributo transferible permite controlar, de forma individual, si un tiquete puede o no ser transferido, lo cual es clave en casos como los paquetes Deluxe, que nunca son transferibles.

Además, el meterlos una lista de eventos hace posible que un mismo tiquete represente tanto una entrada para un solo evento (como los tiquetes básicos) como varios eventos (como los tiquetes de temporada), dando mayor flexibilidad al sistema y facilitando la implementación de sus métodos.

5.2.2 Clase TiqueteBasico

Descripción: Representa el tiquete más simple del sistema, otorgando acceso a una localidad específica de un evento.

Atributos (adicionales a Tiquete):

- precioTiqueteBasico: double - Precio específico del tiquete

Métodos:

- TiqueteBasico(id: String, transferible: boolean, silla: String, localidad: Localidad, evento: Evento, precio: double, usuario: Usuario) - Constructor
- getPrecioTiquete(): double - Implementación del método abstracto
- setPrecioTiqueteBasico(precio: double): void
- usarTiqueteBasico(): void

Restricciones:

- Debe estar asociado a una localidad válida del evento
- El precio es determinado por la localidad más cargos del evento
- Solo permite acceso a un evento específico

La clase TiqueteBasico representa una entrada individual para un solo evento. se diseñó de la manera más simple posible para facilitar el generarlo.

5.2.3 Clase TiqueteMultiple

Descripción: Paquete que incluye múltiples tiquetes vendidos como una unidad indivisible (ej: palcos familiares).

Atributos (adicionales a Tiquete):

- numeroDeTiquetes: int - Cantidad de accesos incluidos
- tiquetes: ArrayList<Tiquete> - Lista de tiquetes individuales contenidos
- tiquetesUsados: ArrayList<Tiquete> - Registro de tiquetes ya utilizados
- precioTiqueteMultiple: double - Precio total del paquete

Métodos:

- TiqueteMultiple(id: String, transferible: boolean, silla: String, localidad: Localidad, evento: Evento, precio: double, numeroDeTiquetes: int, usuario: Usuario) - Constructor
- añadirTiquete(nuevoTiquete: Tiquete): void
- getNumeroDeTiquetes(): int
- setNumeroDeTiquetes(numeroDeTiquetes: int): void
- getTiquetes(): ArrayList<Tiquete>
- getTiquetesUsados(): ArrayList<Tiquete>
- getPrecioTiquete(): double
- setPrecioTiqueteMultiple(precioTiqueteMultiple: double): void
- usarTiqueteMultiple(): void

Restricciones:

- No es transferible como paquete completo
- Se cuenta como una unidad para límites de transacción

La clase TiqueteMultiple es un conjunto de varios tiquetes básicos que se venden y gestionan como una sola unidad para simplificar su manejo y uso. Este tipo de tiquete no puede transferirse, ya que está pensado para ser usado por un mismo grupo de personas, evitando problemas con transferencias. Finalmente, el sistema permite marcar los tiquetes dentro del paquete como usados de forma independiente, lo que resulta útil para grupos o familias en los que no todos asisten al evento al mismo tiempo.

5.2.4 Clase TiqueteTemporada

Descripción: Tiquete especial que otorga acceso a múltiples eventos con una sola compra (ej: abono temporada deportiva).

Atributos (adicionales a Tiquete):

- eventosUsados: ArrayList<Evento> - Registro de eventos ya asistidos
- precio: double - Precio total del pase de temporada

Métodos:

- TiqueteTemporada(id: String, transferible: boolean, silla: String, localidad: Localidad, evento: Evento, precio: double, usuario: Usuario) - Constructor
- getPrecioTiquete(): double
- setPrecio(precio: double): void
- getUsados(): ArrayList<Evento>
- setUsados(usados: ArrayList<Evento>): void
- size(): int
- usarTiqueteTemporada(nombreEvento: String): void
- usarTiqueteTemporada(tiquete: TiqueteTemporada, nombreEvento: String): void

Restricciones:

- Debe incluir al menos 2 eventos
- El precio debe ofrecer ventaja sobre compra individual
- Una vez usado para un evento, no puede reutilizarse para el mismo
- Se marca como totalmente usado cuando se agotan todos los eventos

La clase TiqueteTemporada permite acceder a múltiples eventos bajo una sola compra, aprovechando la lista de eventos definida en la clase base Tiquete, lo que evita duplicar código. El atributo eventosUsados permite controlar qué eventos del paquete ya fueron utilizados, evitando que el mismo tiquete se use más de una vez para un mismo evento.

5.2.5 Clase PaqueteDeluxe

Descripción: Paquete premium que combina entrada al evento con beneficios adicionales y mercancía exclusiva.

Atributos:

- tiquetesCortesia: ArrayList<Tiquete> - Tiquetes adicionales de cortesía incluidos
- beneficiosAdicionales: String - Descripción de beneficios (meet&greet, backstage, etc.)
- mercanciaAdicional: String - Descripción de mercancía incluida
- tiquete: Tiquete - Tiquete principal del paquete

Métodos:

- PaqueteDeluxe(beneficiosAdicionales: String, mercanciaAdicional: String, tiquete: Tiquete) - Constructor
- getTiquetesCortesia(): ArrayList<Tiquete>
- agregarTiqueteCortesia(tiquete: Tiquete): void
- getBeneficiosAdicionales(): String
- setBeneficiosAdicionales(beneficiosAdicionales: String): void
- getMercanciaAdicional(): String
- setMercanciaAdicional(mercanciaAdicional: String): void
- getTiquete(): Tiquete
- setTiquete(tiquete: Tiquete): void

Restricciones:

- NUNCA es transferible (se fuerza en constructor)
- Debe incluir al menos un tiquete base
- Los beneficios son específicos del paquete y no se pueden modificar post-venta
- Las cortesías incluidas siguen las reglas del tiquete base

La clase PaqueteDeluxe se diseñó para ofrecer una versión premium de los tiquetes, combinando acceso al evento con beneficios y mercancía exclusiva. En lugar de crear una nueva subclase para cada tipo de tiquete, esta clase utiliza composición, lo que significa que puede envolver cualquier tipo de tiquete existente y añadirle ventajas adicionales.

El hecho de que los Paquetes Deluxe no sean transferibles es una medida clave para mantener la exclusividad y evitar la reventa no autorizada. Además, la separación entre los beneficios adicionales y la mercancía ayuda a dejar claro qué incluye exactamente el paquete.

5.3 PAQUETE EVENTO

5.3.1 Clase Evento

Descripción: Representa un evento específico para el cual se venden tiquetes, gestionando toda la configuración, venta y control del mismo.

Atributos:

- nombreEvento: String - Nombre descriptivo del evento
- fecha: Date - Fecha de realización
- hora: String - Hora del evento
- tiquetesVendidos: ArrayList<Tiquete> - Registro de todas las ventas
- cantidadTiquetesBasicos: int - Cantidad inicial de tiquetes básicos
- cantidadTiquetesMultiples: int - Cantidad inicial de tiquetes múltiples
- cantidadTiquetesDelux: int - Cantidad inicial de tiquetes deluxe
- cargoPorcentual: double - Porcentaje adicional sobre precio base
- cuotaAdicional: double - Monto fijo adicional por tiquete
- cancelado: boolean - Estado de cancelación del evento
- vendedorEvento: VendedorTiquetes - Factory para creación de tiquetes
- cantidadMaxTiquetesBasicos: int - Límite máximo de básicos
- cantidadMaxDeluxe: int - Límite máximo de deluxe
- cantidadMaxMultiples: int - Límite máximo de múltiples
- maxTiquetesPorTransaccion: int - Límite por compra individual
- administrador: Administrador - Referencia al administrador
- localidades: HashSet<Localidad> - Conjunto de localidades configuradas
- organizador: Organizador - Creador y responsable del evento
- venue: Venue - Lugar físico del evento
- tipoEvento: String - Categoría (MUSICAL, DEPORTIVO, CULTURAL, RELIGIOSO)
- todosLosEventos: static List<Evento> - Registro global de eventos

Métodos:

- Evento(...) - Múltiples constructores con diferentes parámetros
- Getters y setters para todos los atributos
- agregarLocalidad(localidad: Localidad): void
- removerLocalidad(localidad: Localidad): void
- venderTiquete(tipoTiquete: String, usuario: Usuario, evento: Evento, localidad: Localidad, numeroTiquetes: int): Tiquete
- validarFecha(): boolean
- estaVencido(): boolean
- getTodosLosEventos(): static List<Evento>

Restricciones:

- La fecha debe ser futura al momento de creación
- El venue debe estar aprobado por el administrador
- Solo puede haber un evento por venue en una fecha específica
- Debe tener al menos una localidad configurada
- Los límites máximos no pueden ser menores a las cantidades iniciales

La clase Evento es el núcleo del sistema, ya que conecta y coordina la interacción entre los distintos componentes como el organizador, las localidades, los tiquetes y el vendedor.

El uso de un HashSet para almacenar las localidades evita que se registren duplicados y permite búsquedas rápidas y eficientes, optimizando el rendimiento del sistema.

El atributo vendedorEvento aplica el patrón Factory, delegando la creación de tiquetes a una clase especializada, lo que simplifica el código del evento y mejora su mantenibilidad.

Además, la lista estática todosLosEventos actúa como un registro global de todos los eventos creados, facilitando las consultas del administrador sin necesidad de una base de datos o repositorio adicional.

Por último, el tipo de evento influye directamente en los porcentajes de servicio aplicados, permitiendo definir tarifas diferentes el evento, lo que hace al sistema más flexible y adaptable a distintos contextos.

5.3.2 Clase Localidad

Descripción: Representa una sección específica dentro de un venue con características y precios propios.

Atributos:

- nombreLocalidad: String - Nombre identificador de la sección
- tieneNumeracion: boolean - Si los asientos están numerados
- descuento: double - Descuento aplicado (0-1)
- precioBasico: double - Precio para tiquetes básicos
- precioDelux: double - Precio para tiquetes deluxe
- precioMultiple: double - Precio para tiquetes múltiples
- precioTemporada: double - Precio para tiquetes de temporada
- ofertas: ArrayList<Oferta> - Lista de ofertas temporales asociadas

Métodos:

- getNombreLocalidad(): String
- setNombreLocalidad(nombreLocalidad: String): void
- isTieneNumeracion(): boolean
- setTieneNumeracion(tieneNumeracion: boolean): void
- getDescuento(): double
- setDescuento(descuento: double): void - Aplica descuento a todos los precios
- getPrecioBasico(): double
- setPrecioBasico(precioBasico: double): void
- getPrecioDelux(): double
- setPrecioDelux(precioDelux: double): void
- getPrecioMultiple(): double
- setPrecioMultiple(precioMultiple: double): void
- getPrecioBasicoConDescuento(): double
- getPrecioDeluxConDescuento(): double
- getPrecioMultipleConDescuento(): double
- getPrecioTemporadaConDescuento(): double
- setPrecioTemporada(precioTemporada: double): void
- agregarOferta(oferta: Oferta): void
- removerOferta(oferta: Oferta): void

- getOfertas(): ArrayList<Oferta>

Restricciones:

- Todos los tiquetes de una misma localidad tienen el mismo precio base
- Si tiene numeración, cada asiento debe ser único dentro de la localidad
- Los descuentos deben estar entre 0% y 100%
- Los precios no pueden ser negativos

La clase Localidad se encarga de centralizar la definición de los precios dentro de un evento, asegurando que todos los tiquetes de una misma sección mantengan una coherencia en sus valores.

La separación de precios por tipo de tiquete permite aplicar estrategias de precios diferenciadas, como descuentos para compras múltiples o promociones especiales según el tipo de acceso.

El método setDescuento() aplica el descuento de forma uniforme a todos los precios asociados, manteniendo la proporcionalidad entre los diferentes tipos de tiquetes y evitando inconsistencias.

Además, la posibilidad de gestionar ofertas temporales por localidad le da al sistema la capacidad de aplicar promociones específicas y controladas, sin afectar otras partes del evento.

Por último, la opción de numeración permite adaptar la localidad a distintos tipos de escenarios: desde teatros con asientos asignados hasta festivales o conciertos con acceso general, brindando flexibilidad en la organización de los eventos.

5.3.3 Clase Venue

Descripción: Representa el lugar físico donde se realizan los eventos.

Atributos:

- nombre: String - Nombre del establecimiento
- tipoVenue: String - Categoría (ESTADIO, TEATRO, AUDITORIO, etc.)
- capacidad: int - Aforo máximo permitido
- ubicacion: String - Dirección física
- restriccionesDeUso: String - Limitaciones específicas del lugar
- proximoEvento: String - Nombre del siguiente evento programado
- fechaProximoEvento: Date - Fecha del próximo evento
- aprobado: boolean - Estado de aprobación por administrador
- eventosReservados: Map<Date, Evento> - Calendario de ocupación

Métodos:

- Venue(nombre: String, tipo: String, capacidad: int, ubicacion: String, restricciones: String) - Constructor
- Getters y setters para todos los atributos básicos
- setProximoEvento(evento: Evento): void

- validarDisponibilidad(fecha: Date): boolean
- agregarEvento(evento: Evento): boolean
- removerEvento(evento: Evento): boolean
- mismoDia(f1: Date, f2: Date): boolean - Método auxiliar

Restricciones:

- La capacidad debe ser mayor a 0
- No puede tener dos eventos en la misma fecha
- Los venues sugeridos requieren aprobación antes de poder usarse
- Las restricciones de uso deben respetarse al crear eventos

La clase Venue administra la disponibilidad física de los espacios donde se realizan los eventos, asegurando que no existan conflictos de fechas ni sobreasignaciones.

El uso de un Map<Date, Evento> permite validar la disponibilidad de forma rápida y eficiente (O(1)), lo cual es esencial para evitar la doble reserva de un mismo lugar en una fecha determinada. La aprobación por parte del administrador garantiza que solo los venues que han sido verificados puedan ser utilizados. Además, las restricciones de uso permiten reflejar condiciones del espacio asegurando el desarrollo correcto del evento.

Por último, la capacidad máxima del venue funciona como un límite superior para la suma de todas las localidades del evento, evitando la sobreventa de tiquetes y asegurando que el aforo nunca exceda lo permitido.

5.3.4 Clase Oferta

Descripción: Representa un descuento temporal aplicable a una localidad específica de un evento.

Atributos:

- evento: Evento - Evento al que aplica la oferta
- localidad: Localidad - Localidad beneficiada
- descuento: double - Porcentaje de descuento (0-100)
- fechaInicio: Date - Inicio de vigencia
- fechaFin: Date - Fin de vigencia

Métodos:

- Oferta(evento: Evento, localidad: Localidad, descuento: double, fechaInicio: Date, fechaFin: Date) - Constructor
- getEvento(): Evento
- getLocalidad(): Localidad
- getDescuento(): double
- getFechaInicio(): Date
- getFechaFin(): Date
- estaVigente(fechaActual: Date): boolean

- calcularPrecioConDescuento(precioBase: double, fechaActual: Date): double

Restricciones:

- La fecha de inicio debe ser anterior a la fecha fin
- El descuento debe estar entre 0% y 100%
- La oferta no puede extenderse más allá de la fecha del evento
- Una vez creada, los parámetros no son modificables

La clase Oferta permite implementar descuentos temporales, estos descuentos pueden aplicarse a localidades específicas dentro de un evento. Una vez creada, la oferta es inmutable, lo que evita modificaciones que podrían afectar a los compradores que ya adquirieron tiquetes bajo ciertas condiciones.

El método estaVigente() concentra la lógica de validación temporal, verificando si la fecha actual se encuentra dentro del rango de validez de la oferta. Por su parte, el método calcularPrecioConDescuento() calcula el precio final, aplicando el descuento únicamente cuando la oferta está activa.

Gracias a esta separación de responsabilidades, el sistema puede manejar múltiples ofertas con distintas ventanas de tiempo para una misma localidad.

5.4 PAQUETE COMPRA

5.4.1 Clase VendedorTiquetes

Descripción: Es el responsable de la creación de todos los tiquetes del sistema, garantizando códigos únicos y aplicando reglas de negocio.

Atributos:

- codigos: static HashSet<String> - Registro global de códigos generados para garantizar unicidad

Métodos:

- VendedorTiquetes() - Constructor
- generarCodigoUnico()
- venderTiquete(tipoTiquete: String, precioBase: double, cargoPorcentual: double, cuotaAdicional: double, usuario: Usuario, evento: Evento, localidad: Localidad, numeroTiquetes: int): Tiquete

Funcionalidades internas del método venderTiquete:

- Genera códigos únicos de 7 dígitos
- Valida unicidad contra el HashSet estático
- Asigna asientos aleatorios en localidades numeradas (A-J + 1-20)

- Calcula precio final según tipo de tiquete y usuario
- Aplica precio cero para cortesías del organizador
- Crea la instancia apropiada según tipo solicitado
- Para tiquetes múltiples, crea los tiquetes individuales contenidos

Restricciones:

- Los códigos generados deben ser únicos en todo el sistema
- El tipo de tiquete debe ser válido (basico, temporada, multiple)
- En localidades numeradas debe asignar asiento
- Precio cero solo para organizador en su propio evento

La clase VendedorTiquetes se encarga de crear todos los tipos de tiquetes del sistema en un solo lugar. Esto significa que, en lugar de que cada clase tenga que preocuparse por cómo se construye un tiquete, toda la lógica de creación está centralizada aquí. De esta forma, cuando un cliente o un organizador necesita generar un tiquete, simplemente llama a venderTiquete() la cual se encarga de asignar todos los datos pertinentes al tiquete y construirlo según la elección.

El uso de un HashSet se uso para guardar los códigos asegura que ningún tiquete se repita, con validaciones instantáneas. También, la asignación aleatoria de asientos simula la experiencia real de compra en venues numerados. Además, el cálculo del precio considera si el comprador es el organizador (caso en el que el tiquete se considera una cortesía), lo que simplifica el trabajo de otras clases.

5.5 PAQUETE PERSISTENCIA

5.5.1 Clase CentralPersistencia

Descripción: Factoría central que gestiona la creación de instancias de persistencia según el tipo de archivo requerido.

Atributos:

- JSON: String = "JSON" - Constante para tipo de archivo JSON

Métodos:

- getPersistenciaFinanzas(tipoArchivo: String): PersistenciaFinanzas
- getPersistenciaUsuarios(tipoArchivo: String): PersistenciaUsuarios
- getPersistenciaTiquetes(tipoArchivo: String): PersistenciaTiquetes

Restricciones:

- Solo soporta tipo JSON en la implementación actual
- Debe retornar la implementación correcta según tipo solicitado

La clase CentralPersistencia funciona como una central para manejar toda la persistencia de datos del sistema. Su objetivo es centralizar la forma en que se guardan y cargan los datos, evitando que otras clases tengan que preocuparse por los detalles del almacenamiento. Gracias a esta abstracción, el sistema puede cambiar fácilmente el formato de persistencia sin modificar el código del resto del programa.

Además, la separación por tipo de entidad (como finanzas, usuarios y tiquetes) permite aplicar optimizaciones específicas según la naturaleza de cada dato, así como realizar cargas selectivas cuando no es necesario traer toda la información a memoria. Esta clase mejora la flexibilidad, mantenibilidad y escalabilidad del sistema, facilitando la evolución futura de la plataforma sin afectar su funcionamiento actual.

5.5.2 Interfaces de Persistencia

Interface PersistenciaUsuarios:

- cargarUsuarios(archivo: String, administrador: Administrador): void
- salvarUsuarios(archivo: String, administrador: Administrador): void

Interface PersistenciaFinanzas:

- cargarFinanzas(archivo: String, administrador: Administrador): void
- salvarFinanzas(archivo: String, administrador: Administrador): void

Interface PersistenciaTiquetes:

- cargarTiquetes(archivo: String, administrador: Administrador): void
- salvarTiquetes(archivo: String, administrador: Administrador): void

Las interfaces de persistencia se utilizan para definir contratos que especifican cómo deben guardarse y cargarse los datos del sistema, sin imponer una forma única de hacerlo. Gracias a esto, es posible tener múltiples implementaciones sin alterar el funcionamiento del resto del código.

El Administrador se pasa como parámetro en los métodos de carga y guardado para que las operaciones de persistencia tengan acceso al contexto completo del sistema, incluyendo usuarios, eventos, finanzas y tiquetes.

De esta manera, se garantiza que la información se mantenga consistente y sincronizada entre las distintas capas del sistema, manteniendo la separación entre la lógica y el almacenamiento.

5.5.3 Implementaciones JSON

Clases de Implementación:

- PersistenciaUsuariosJson - Maneja serialización de usuarios
- PersistenciaFinanzasJson - Gestiona datos financieros

- PersistenciaTiquetesJson - Persiste tiquetes y relaciones

Atributos comunes para keys JSON:

- Constantes String para nombres de campos JSON
- Métodos privados de conversión objeto-JSON y viceversa

El formato JSON fue elegido para la persistencia de datos porque es fácil de leer y entender por humanos, lo que facilita el debugging y las pruebas durante el desarrollo. Además, su estructura ligera y ampliamente soportada lo convierte en una opción práctica para sistemas en crecimiento.

La separación en múltiples archivos permite evitar posibles conflictos de escritura concurrente en el futuro y facilita la realización de copias de seguridad específicas sin afectar el resto de la información.

---

6. Relaciones entre Clases

6.1 Herencia

- Usuario *(abstracta)*
  - Cliente *(extends Usuario)*
  - Organizador *(extends Usuario)*
  - Administrador *(extends Usuario)*
- Tiquete *(abstracta)*
  - TiqueteBasico *(extends Tiquete)*
  - TiqueteMultiple *(extends Tiquete)*
  - TiqueteDeTemporada *(extends Tiquete)*
  - PaqueteDeluxe *(extends Tiquete)*

PaqueteDeluxe (composición con Tiquete)

6.2 Asociaciones

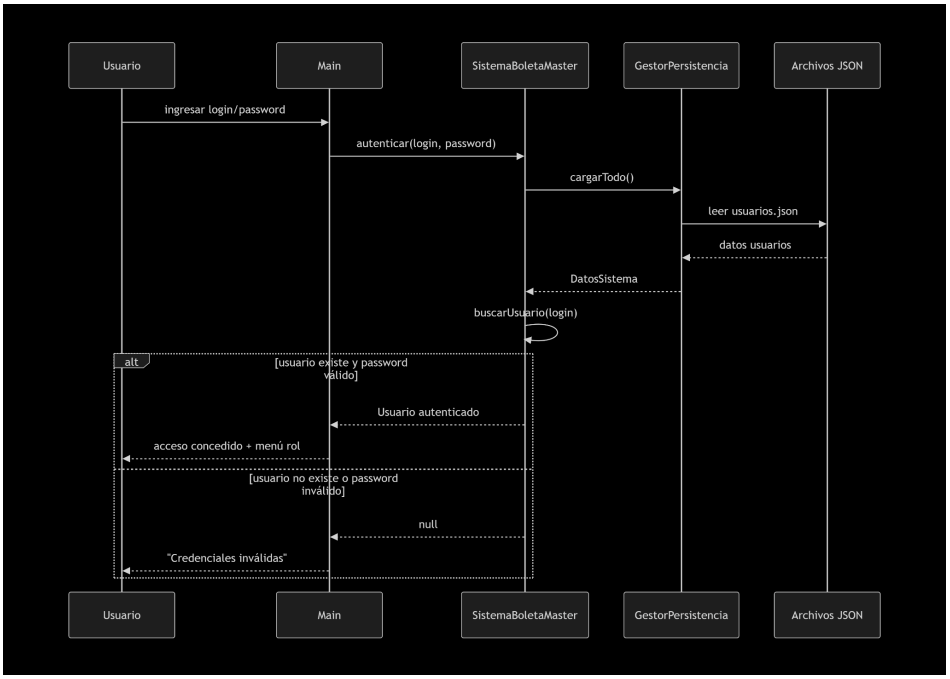
- Cliente 1 ⟷ 0..* Tiquete:Un cliente puede poseer múltiples tiquetes; un tiquete puede estar asignado a un cliente (o disponible).
- Organizador 1 ⟷ 0..* Evento: Un organizador puede crear y gestionar múltiples eventos.
- Evento * ⟷ 1 Venue: Un Venue solo puede tener un evento por fecha.
- Evento 1 ⟷ 1..* Localidad: Todo evento debe tener al menos una localidad.
- Localidad 1 ⟷ 0..* TiqueteBasico: Los tiquetes básicos pertenecen a una localidad específica y heredan su precio y condiciones.
- Cliente 1 ⟷ 0..* Transacción: Un cliente puede hacer diferentes transacciones.
- Transacción 1 ⟷ 1..* Tiquete: Una transacción debe incluir al menos un tiquete.

---

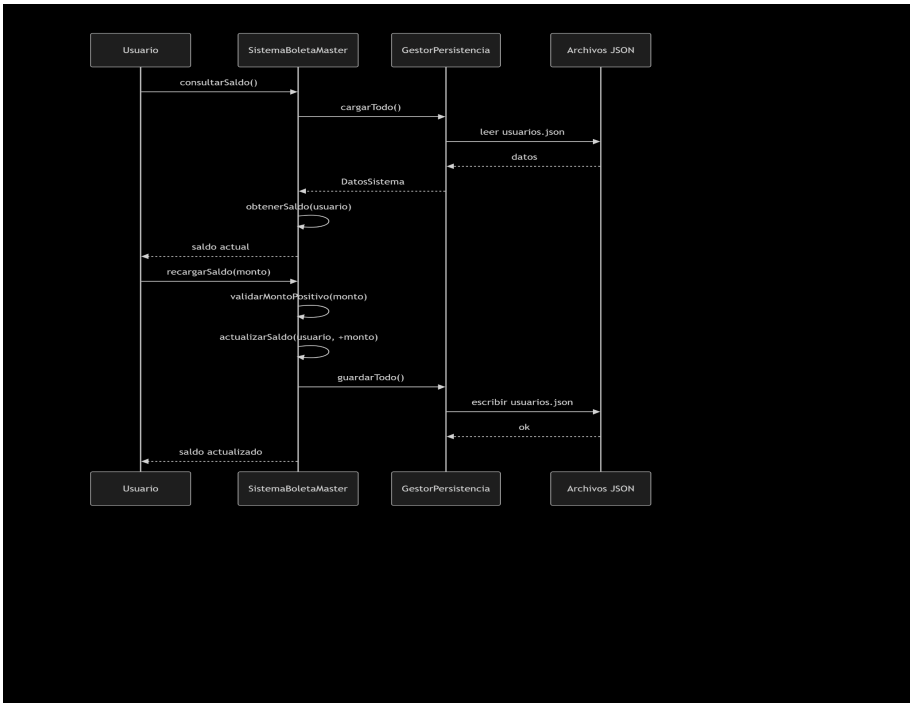
7. Requerimientos Funcionales

1. AUTENTICACIÓN Y USUARIOS

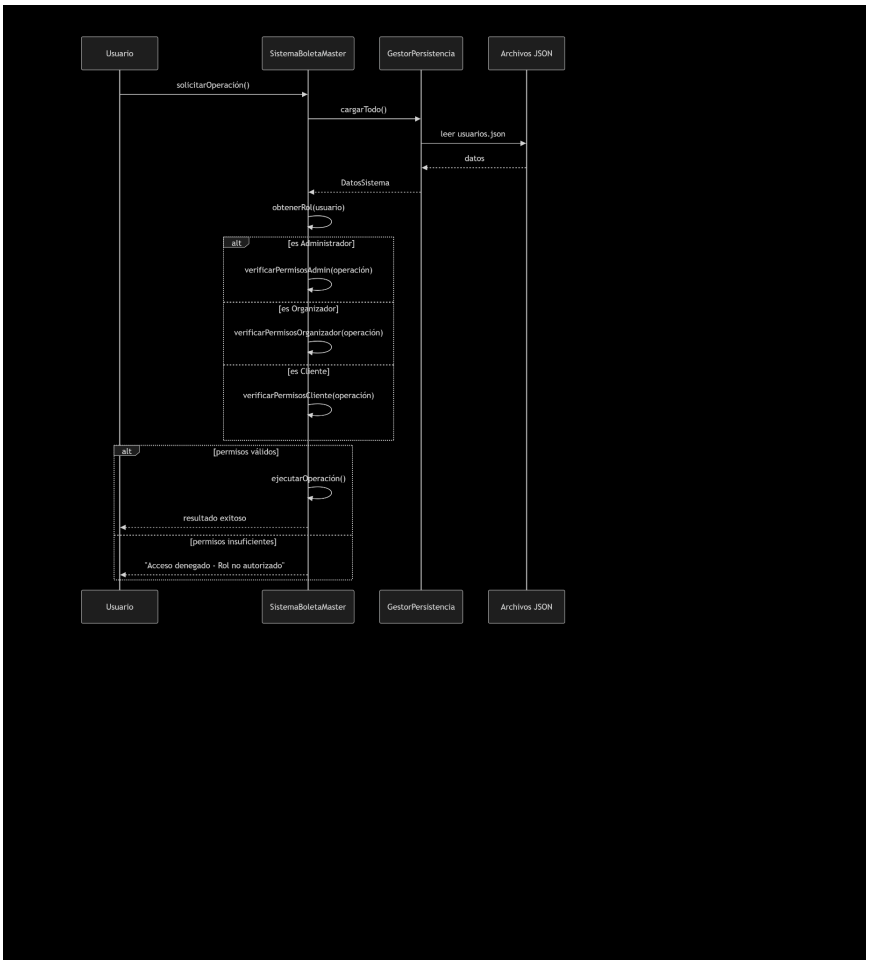
RF1 – Permitir la autenticación de usuarios mediante credenciales únicas (login y contraseña).



RF2 – Gestionar el saldo virtual de cada usuario para realizar compras, transferencias y reembolsos.

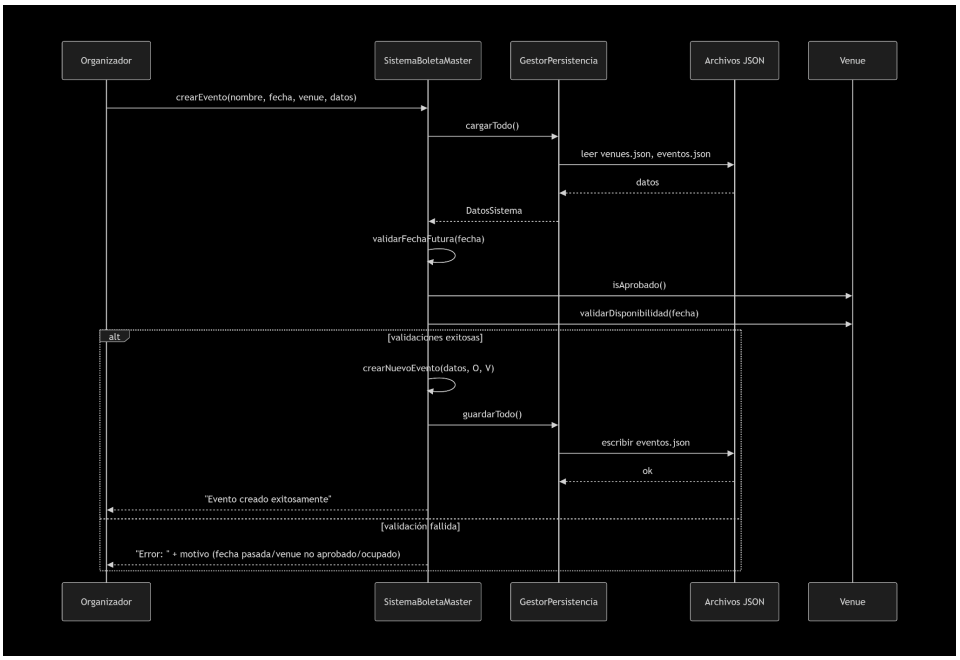


RF3 – Diferenciar los roles de **Cliente**, **Organizador** y **Administrador**, asignando permisos y funcionalidades específicas según el tipo de usuario.

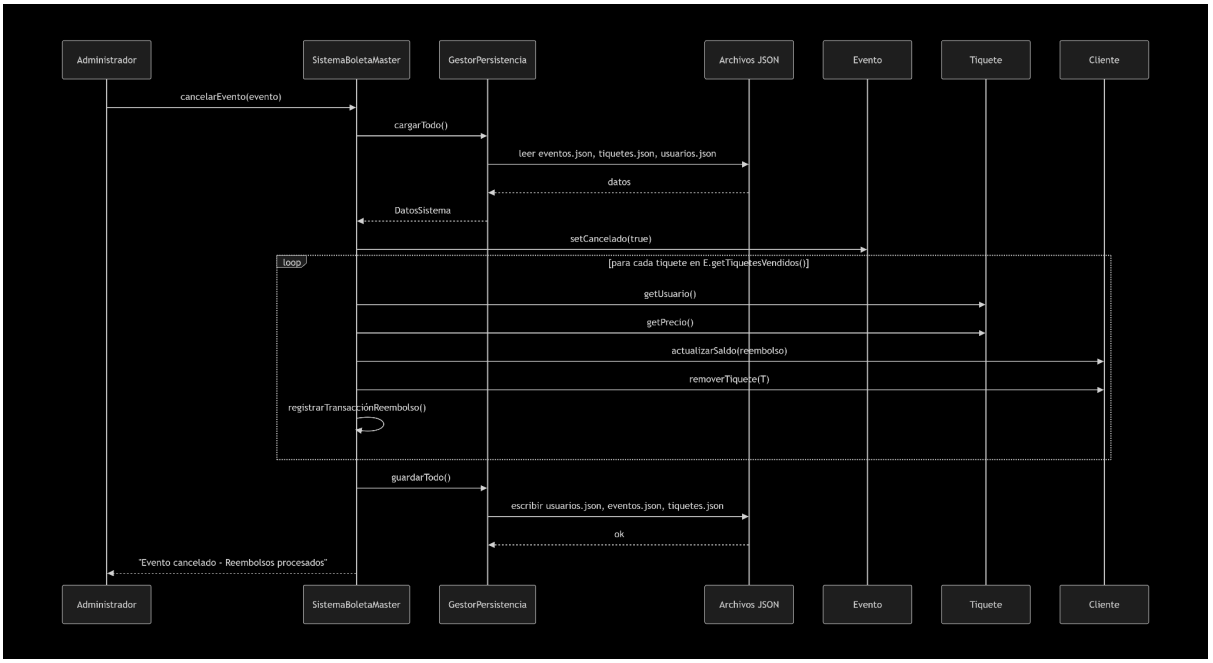


2. GESTIÓN DE EVENTOS

RF4 – Permitir la creación de eventos con validaciones de fecha futura y asociación obligatoria a un *venue* aprobado.

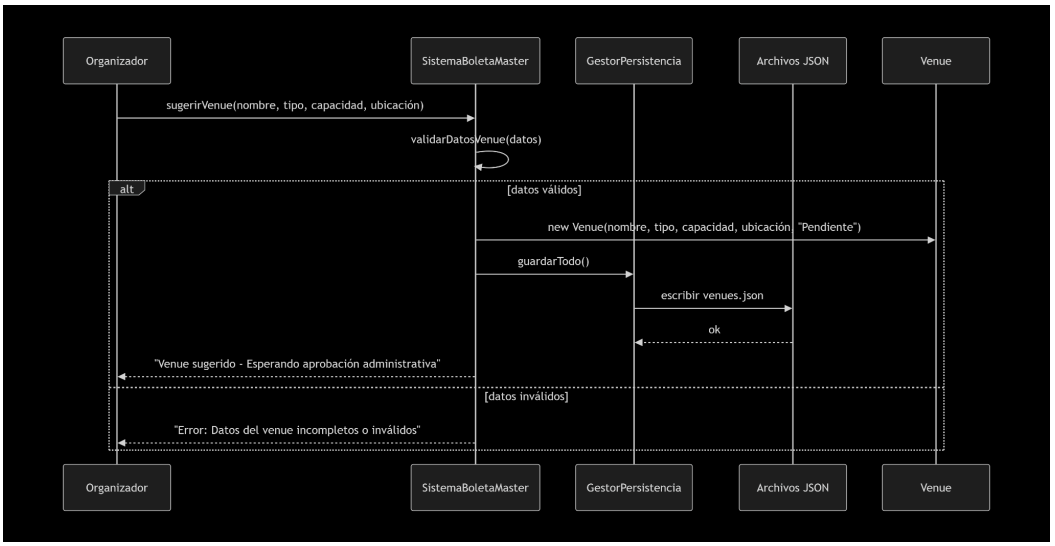


RF5 – Permitir la cancelación de eventos con ejecución del sistema de reembolsos correspondiente según el tipo de cancelación (administrativa o por organizador).

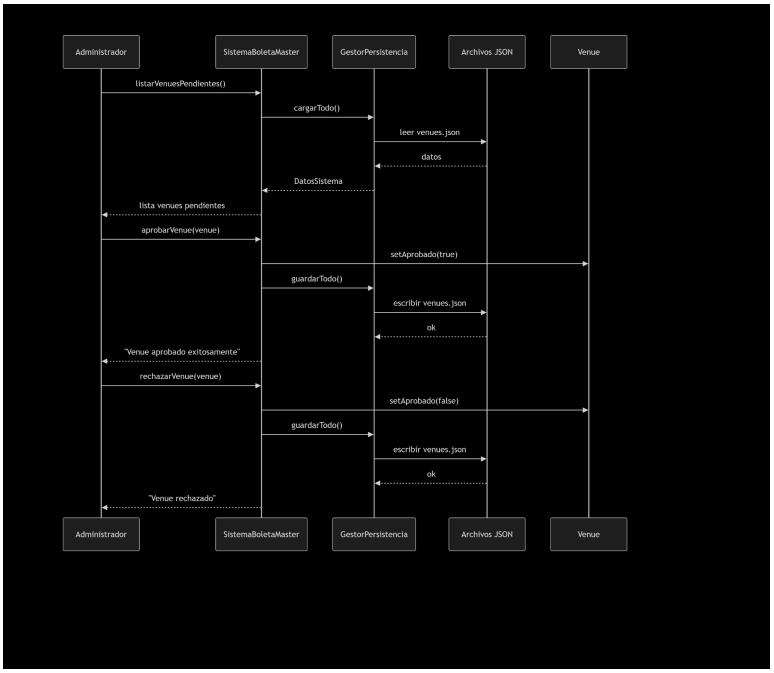


3. GESTIÓN DE VENUES

RF6 – Permitir a los organizadores sugerir nuevos *venues* al sistema con información básica (nombre, tipo, capacidad, ubicación).

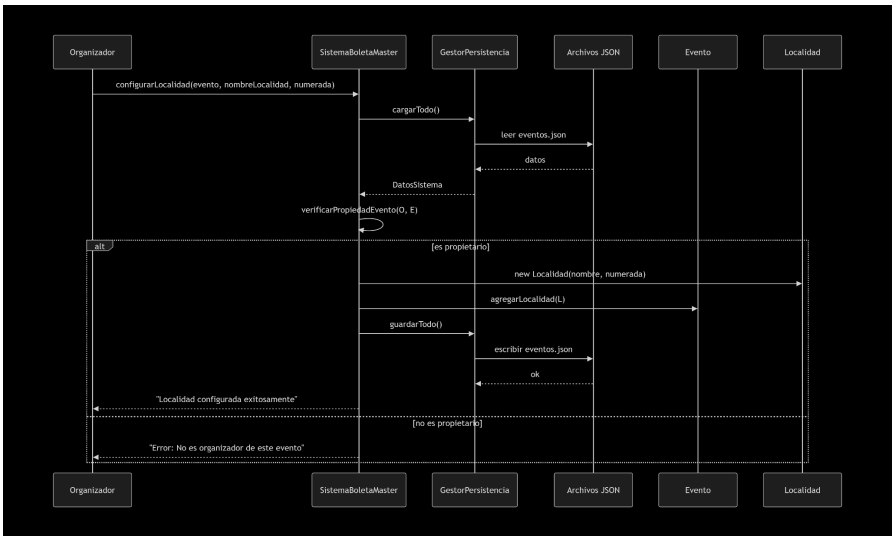


RF7 – Permitir al administrador aprobar o rechazar *venues* sugeridos, garantizando que no existan eventos simultáneos en el mismo lugar y fecha.

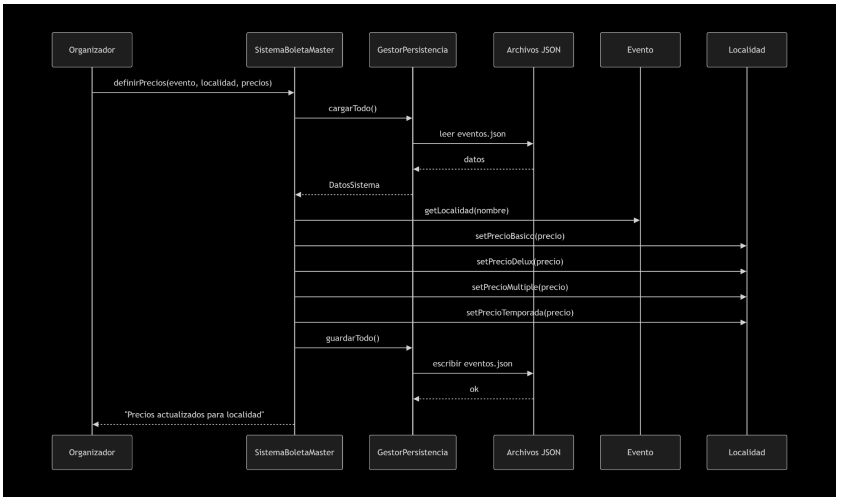


4. CONFIGURACIÓN DE LOCALIDADES

RF8 – Permitir la configuración de localidades dentro de un evento, especificando si son numeradas o de acceso general.

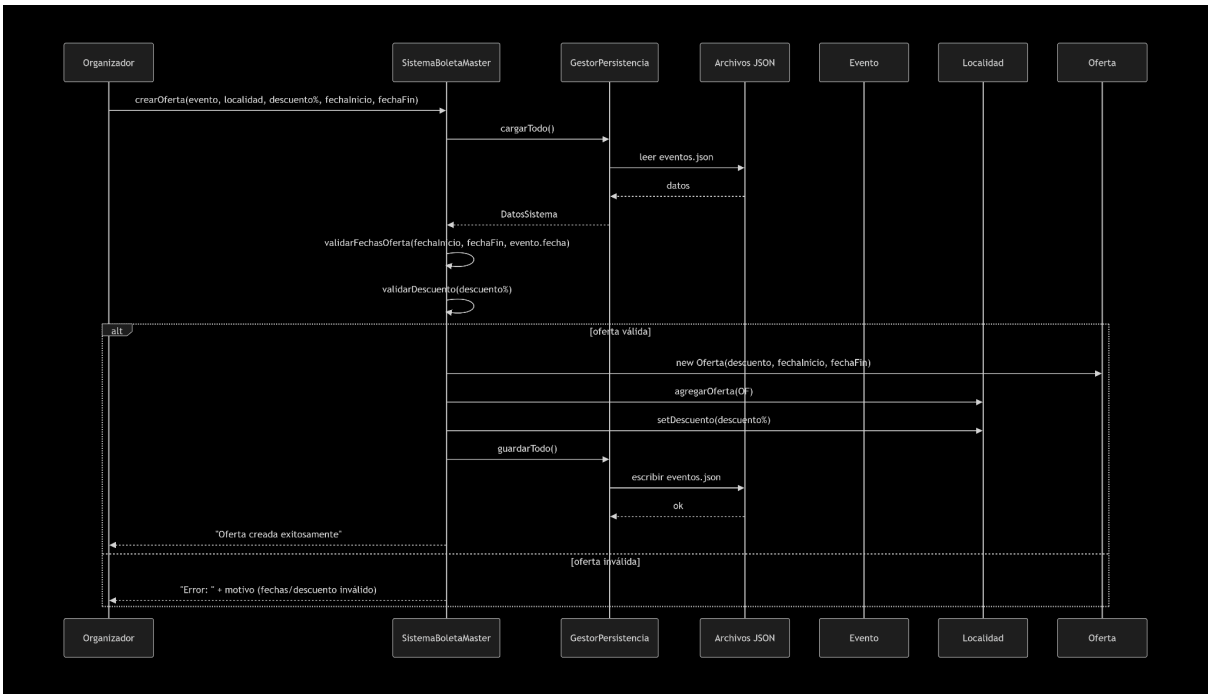


RF9 – Definir los precios de los tiquetes de acuerdo con el tipo de localidad.

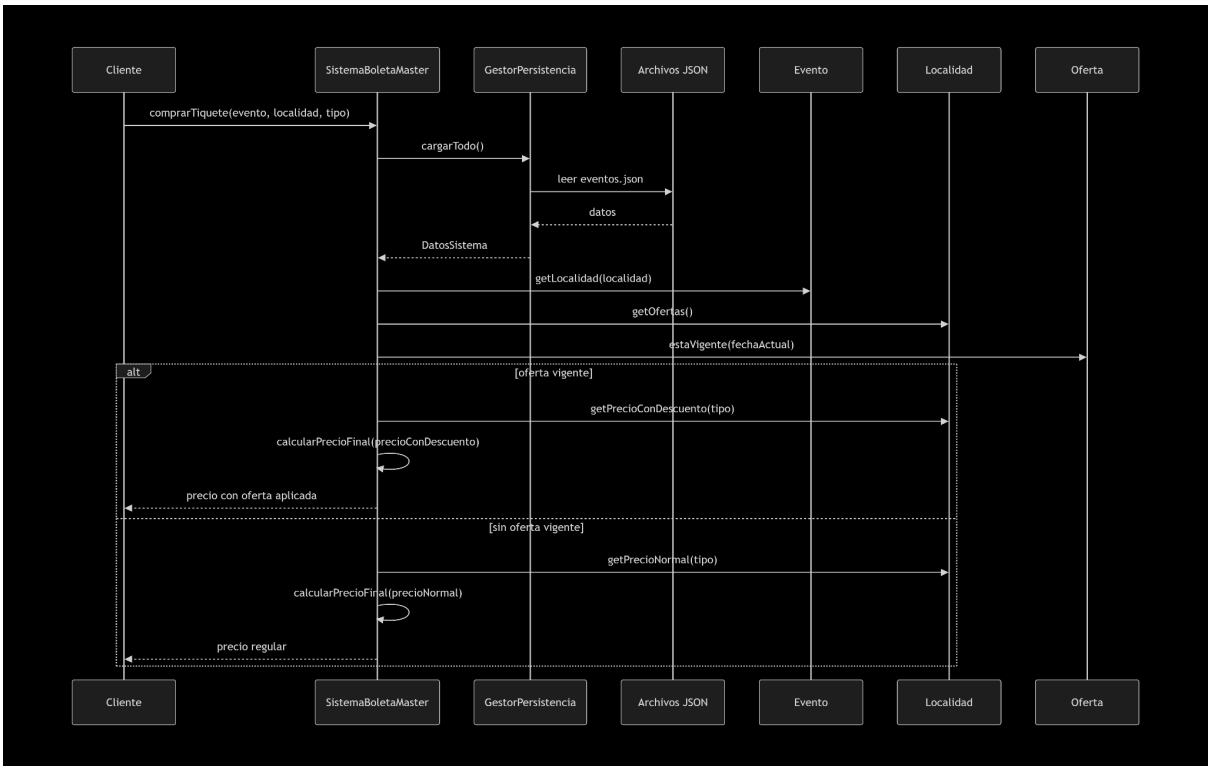


5. OFERTAS Y PROMOCIONES

RF10 – Permitir a los organizadores crear ofertas temporales con descuentos porcentuales y rangos de vigencia definidos.

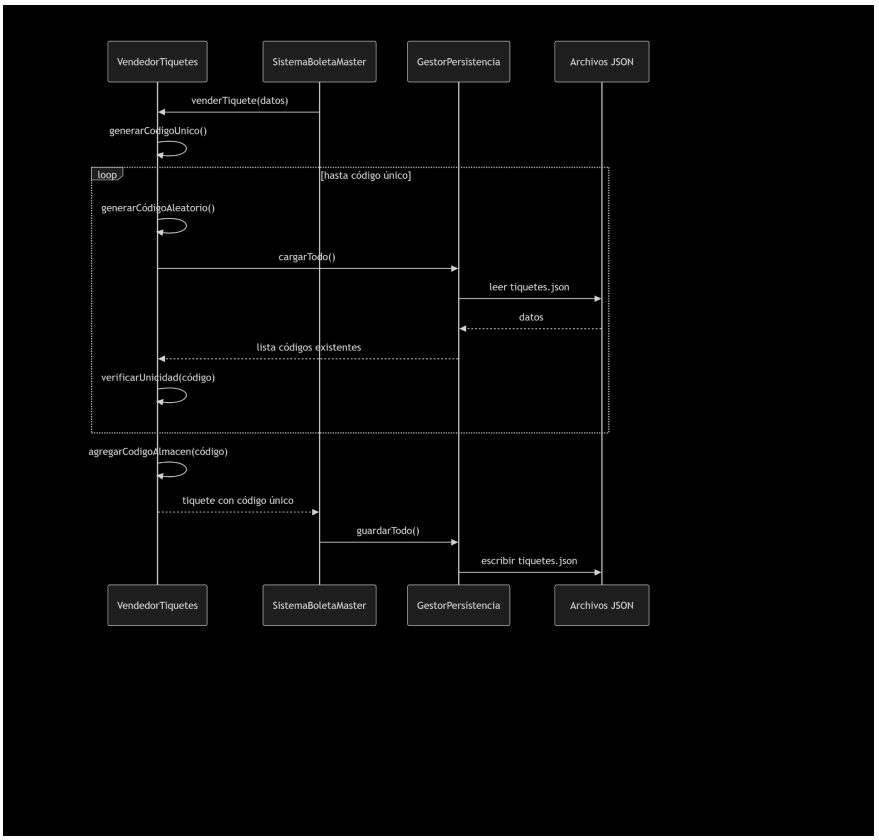


RF11 – Aplicar automáticamente las ofertas vigentes durante el proceso de compra, afectando el precio final de los tiquetes.

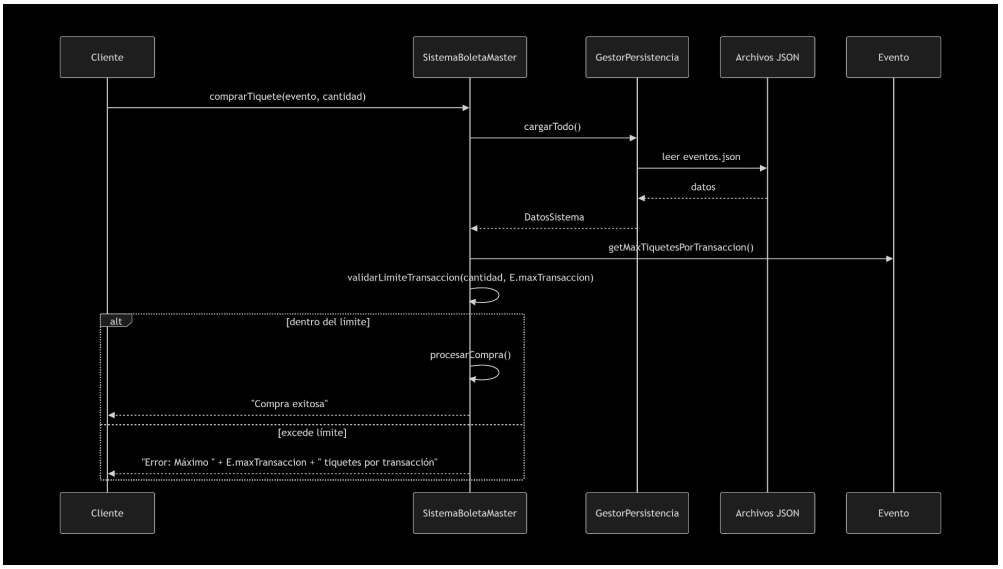


6. VENTA DE TIQUETES

RF12 – Permitir la venta de tiquetes generando un código único por unidad vendida.

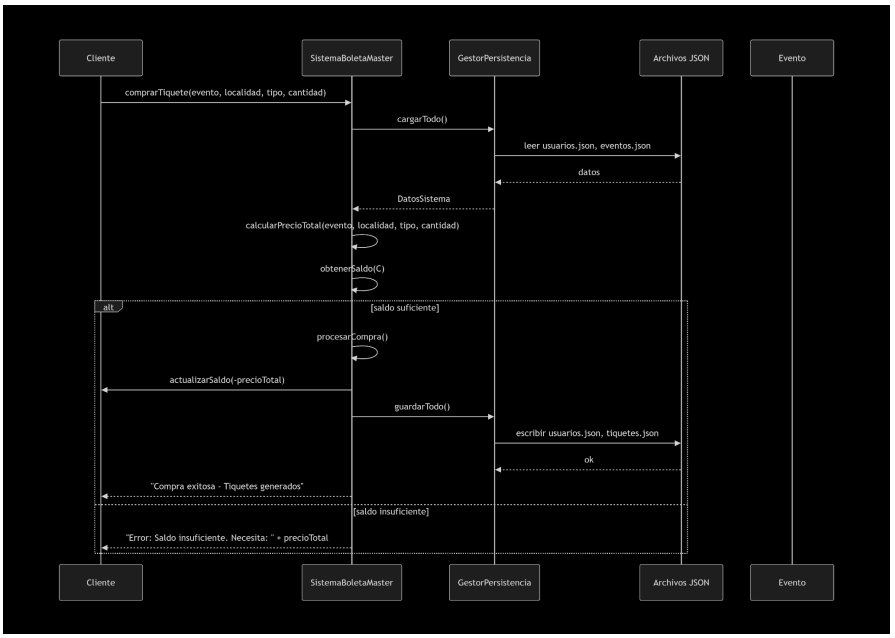


RF13 – Validar que no se supere el límite máximo de tiquetes por transacción establecido por cada evento.

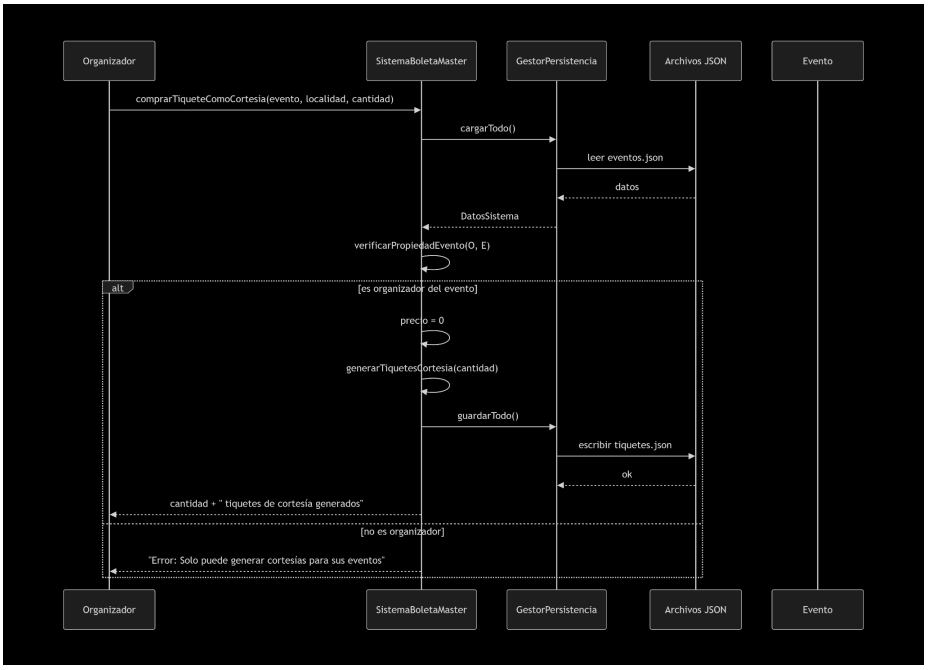


7. COMPRAS DE CLIENTES

RF14 – Permitir a los clientes comprar tiquetes de diferentes tipos, verificando el saldo disponible y las restricciones del evento.

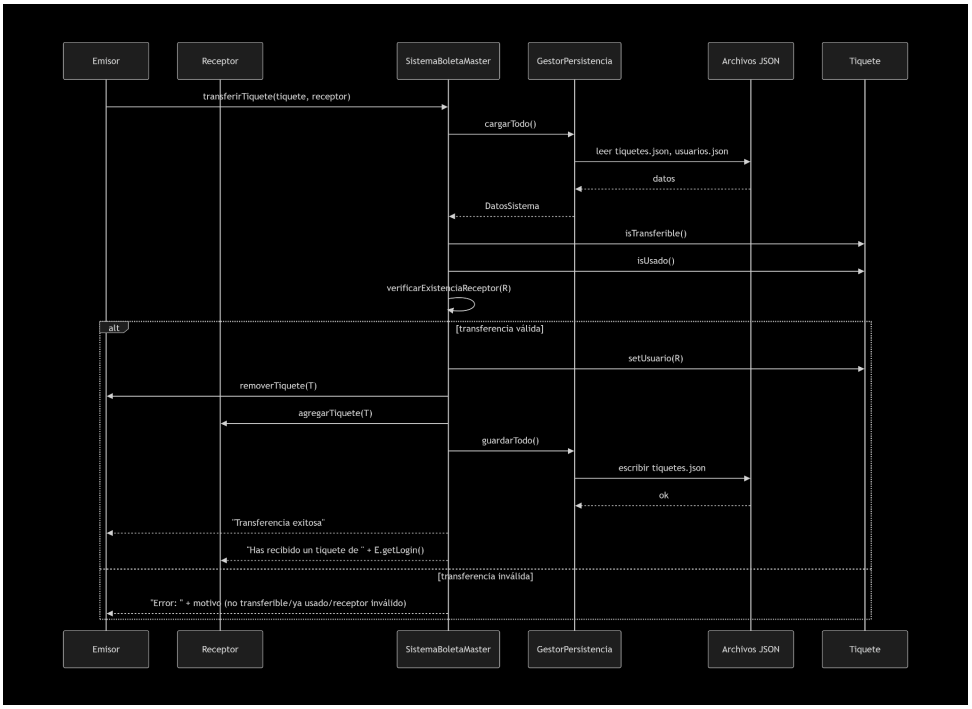


RF15 – Permitir a los organizadores adquirir tiquetes de sus propios eventos como cortesías, sin costo y dentro de los límites definidos.

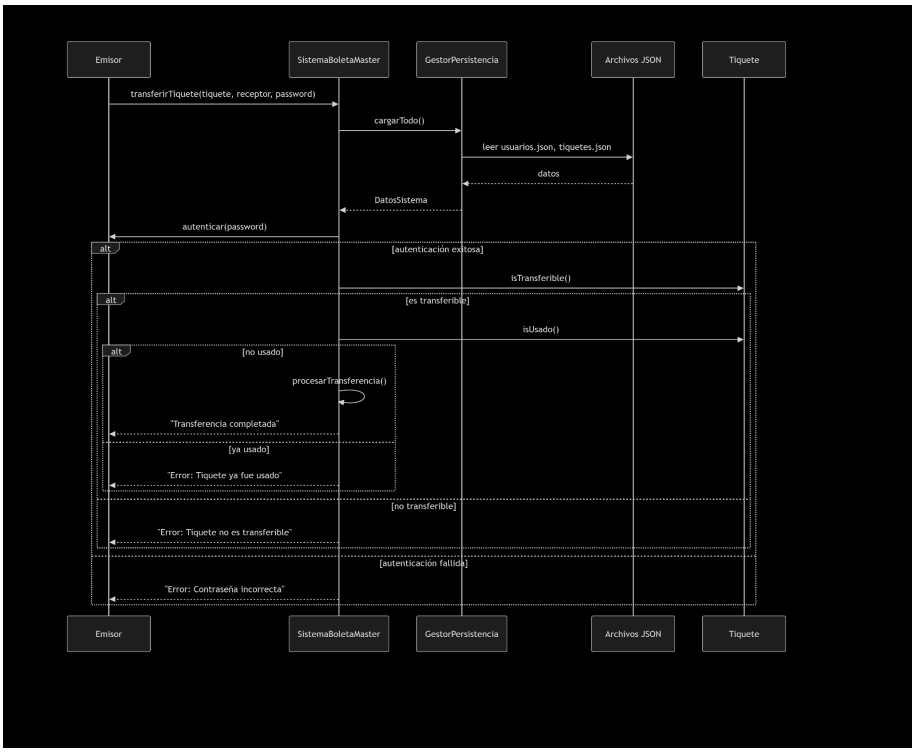


8. TRANSFERENCIAS

RF16 – Permitir la transferencia de tiquetes entre usuarios registrados

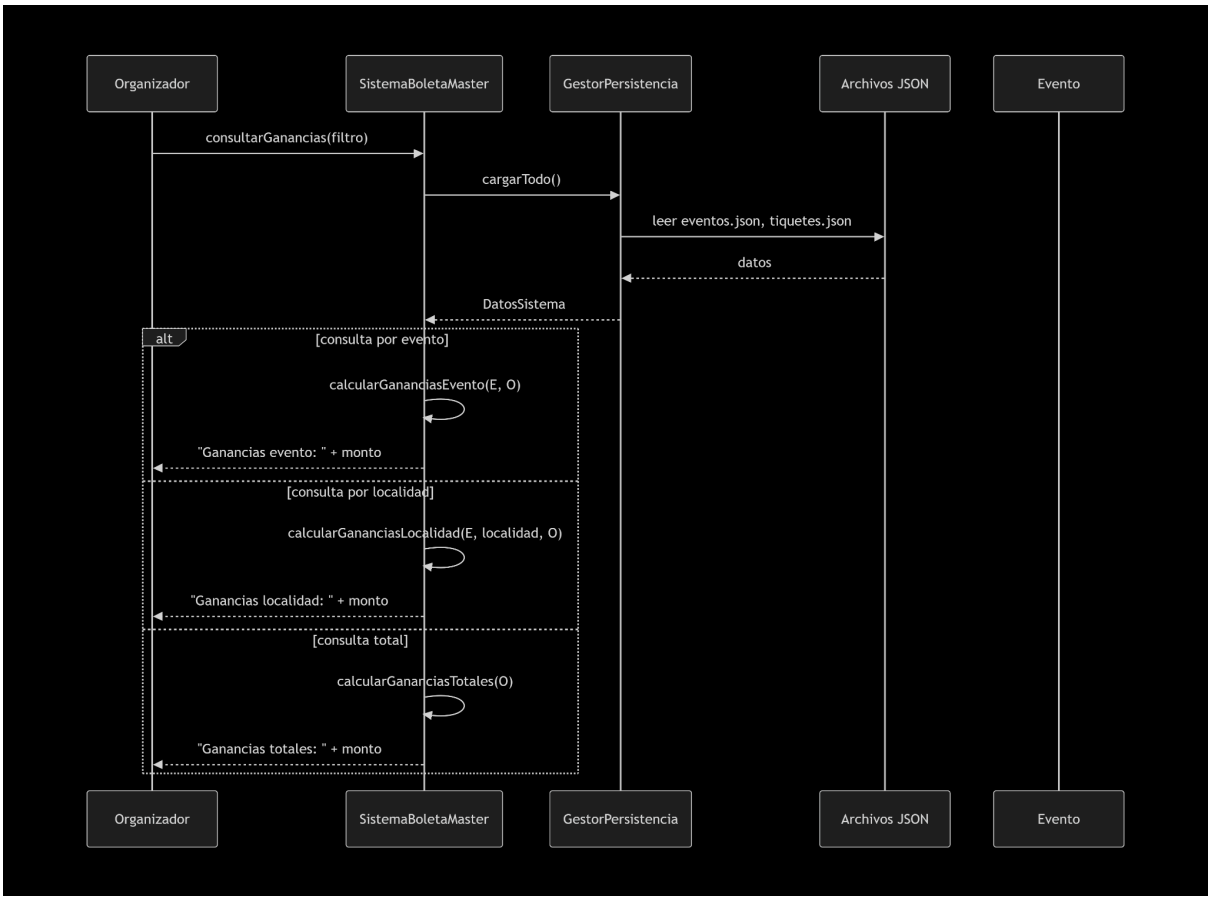


RF17 – Validar la transferibilidad, el estado del tiquete y las credenciales del usuario emisor para garantizar la seguridad de la operación.

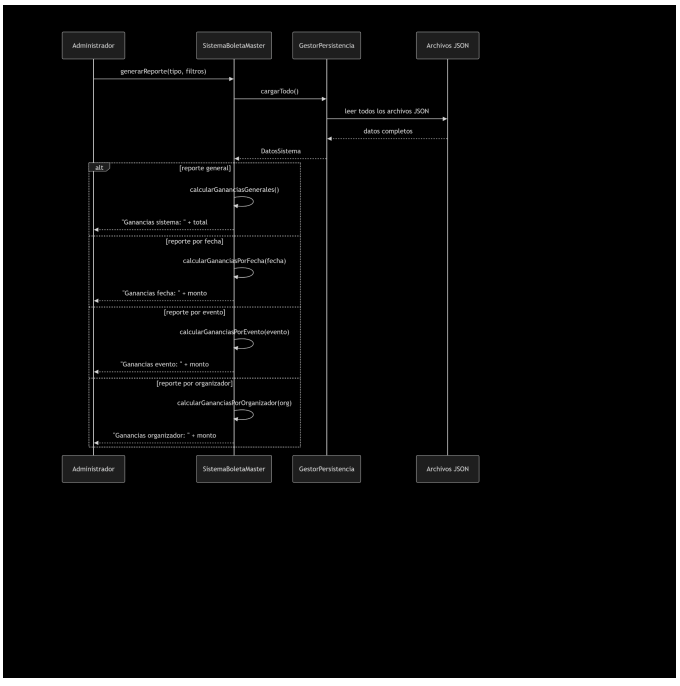


9. CONSULTAS Y REPORTES

RF18 – Permitir a los organizadores consultar las ganancias obtenidas por evento y por localidad.

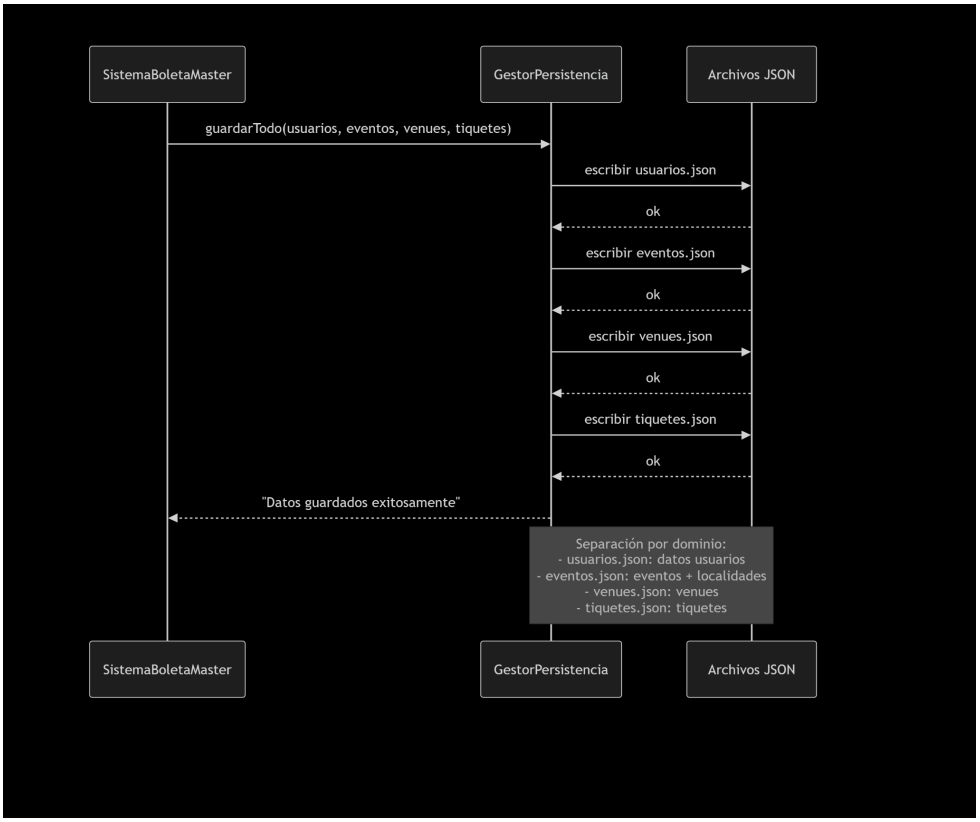


RF19 – Permitir al administrador generar reportes financieros globales, filtrados por fecha, evento u organizador.

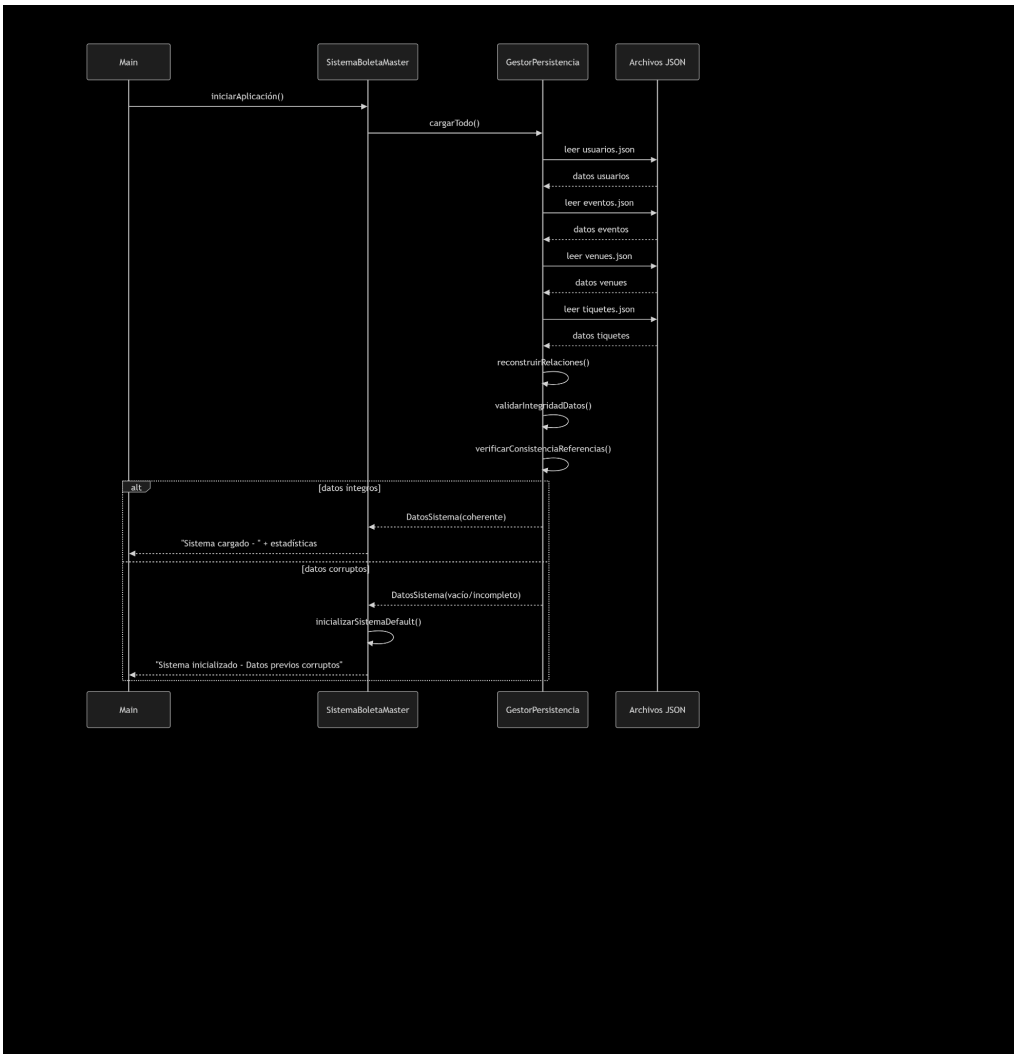


10. PERSISTENCIA

RF20 – Guardar todos los datos del sistema (usuarios, eventos, tiquetes y finanzas) en archivos con formato **JSON**, separados por dominio.



RF21 – Cargar los datos desde los archivos JSON al iniciar el sistema, garantizando integridad y coherencia de la información almacenada.



9. Pruebas del Sistema

Prueba 1 — Creación de tiquete

- Objetivo: Verificar la correcta creación de tiquetes
- Entradas: datos necesarios como usuario que compra el tiquete, el evento localidad etc
- Esperado:un tiquete con código de identificación único.

-

Prueba 2 — Comprar tiquete

- Objetivo: verificar que n cliente pueda comprar un tiquete
- Entradas: todos los datos necesarios para la compra del tiquiete incluyendo el usuario
- Esperado: que el usuario reciba un tiquete dentro asociado a su perfil dentro del sistema

Prueba 3 — crear Evento

- Objetivo: verificar la correcta creacion de un evento
- Entradas: se crea un evento por parte del organizador
- Esperado: un nuevo evento el cual sea añadido a los eventos del organizador
-

Prueba 4 — Compra en localidad numerada

- Objetivo: Reservar asientos únicos.
- Entradas: Cliente compra Platea con asientos 10, 11 y 12; luego otro cliente intenta el 11.
- Esperado: El segundo intento es rechazado por asiento ocupado.

Prueba 5 — Transferencia permitida

- Objetivo: Verificar que se pueda transferir un tiquete simple.
- Entradas: Cliente A transfiere un tiquete válido a Cliente B.
- Esperado: El tiquete aparece ahora en la cuenta del Cliente B; no es posible si el evento ya pasó.

-

Prueba 6 — Compra de cortesías por organizador

- Objetivo: Comprobar que si el organizador compra en su propio evento, sea marcado como cortesía.
- Entradas: Organizador compra 2 tiquetes en su evento.
- Esperado: Los tiquetes quedan registrados, pero no generan ingresos ni para él ni para la tiquetera.

Prueba 7 — Cancelación por administrador

- Objetivo: Validar los reembolsos de cancelación administrativa.
- Entradas: Admin cancela el Concierto A.
- Esperado: A los clientes se les devuelve el valor pagado menos la cuota de emisión, en su saldo virtual.

Prueba 8 — Cancelación solicitada por organizador

- Objetivo: Validar reglas de reembolso cuando cancela el organizador.
- Entradas: Organizador pide cancelar su evento; administrador lo aprueba.
- Esperado: A los clientes se les devuelve solo el precio base; los recargos se los queda la tiquetera.

Prueba 9 — Reembolso por calamidad

- Objetivo: Permitir reembolsos especiales por decisión del administrador.
- Entradas: Admin autoriza reembolso a un cliente por calamidad.
- Esperado: El dinero aparece en el saldo virtual del cliente.

Prueba 10 — Reportes del organizador

- Objetivo: Generar estadísticas de ventas y ocupación.
- Entradas: Organizador consulta reporte general, por evento y por localidad.
- Esperado: Se muestran ventas totales y % ocupación sin contar los recargos.

Prueba 11 — Reportes del administrador

- Objetivo: Calcular ingresos de la tiquetera.
- Entradas: Admin consulta reportes por fecha, evento y organizador.
- Esperado: Los reportes muestran solo lo ganado por cuotas de emisión y servicio.

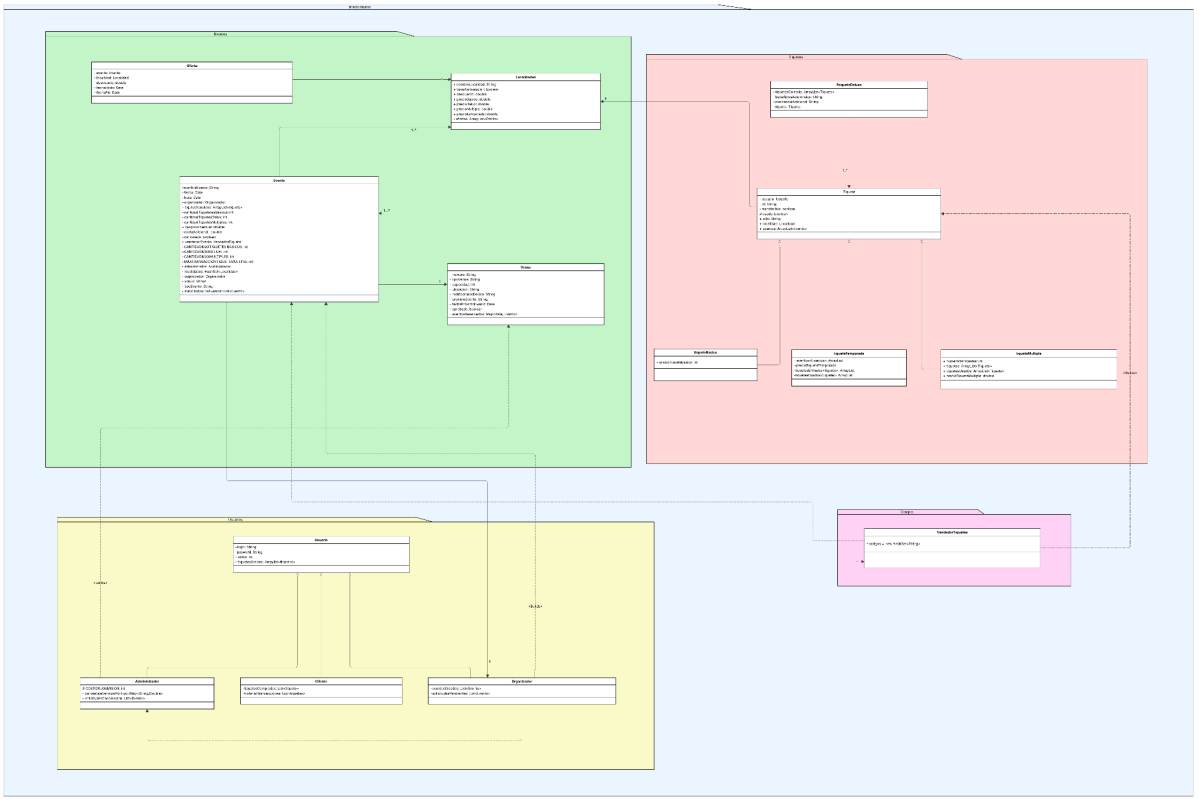
Prueba 12 — Persistencia

- Objetivo: Asegurar que los datos se guarden y se recarguen al reiniciar.
- Entradas: Crear usuarios, eventos y compras; cerrar y volver a abrir la aplicación.
- Esperado: Los datos persisten y al cargar aparecen iguales a como estaban antes.

---

10. Diagrama de Clases

10.1 Diagrama de Alto Nivel



10.2 Diagrama Detallado