ENTREGA 2 DPOO

Samantha Puentes Pinzón - 202410295
Jefferson Orozco Bolaños - 202415866
David Santiago Ruiz Corredor - 202414922

Repositorios proyecto 1:

https://github.com/DPOO-20242-SPP/Proyecto-1-Dpoo.git

Repositorios proyecto 2:

https://github.com/DPOO-20242-SPP/Entrega-2-DPOO.git

Contexto y alcance:

La solución propuesta busca dar respuesta a la necesidad de gestionar de manera integral la venta y administración de tiquetes para eventos, ofreciendo una plataforma estructurada que permita coordinar las operaciones de compra, organización y control financiero dentro de un mismo sistema. La complejidad de este dominio radica en la coexistencia de múltiples actores —compradores, organizadores y administradores—, así como en la diversidad de reglas asociadas a la creación de eventos, la definición de localidades, los límites de venta y los procesos de reembolso o transferencia de entradas.

El sistema aborda estos desafíos mediante una arquitectura orientada a objetos, que modela con precisión las entidades fundamentales del dominio, sus relaciones y comportamientos. A través de un diseño modular, se promueve un bajo acoplamiento entre componentes, una alta cohesión funcional y una clara distribución de responsabilidades, lo que garantiza una estructura flexible, mantenible y escalable. Esta aproximación facilita futuras extensiones del sistema sin comprometer la estabilidad del núcleo funcional.

En términos funcionales, la solución contempla la administración de tres roles principales:

- Usuarios compradores, quienes pueden registrarse, adquirir tiquetes, visualizar sus compras y transferir entradas a otros usuarios.

- Organizadores de eventos, responsables de crear y configurar eventos, asignar precios, tipos de tiquetes y localidades, así como ofrecer descuentos o promociones.

- Administrador, encargado de fijar los cargos por servicio y emisión, aprobar venues propuestos, gestionar cancelaciones, autorizar reembolsos y supervisar el desempeño financiero del sistema.

El alcance de la solución incluye la creación, almacenamiento y consulta persistente de información relacionada con eventos, usuarios, tiquetes y transacciones. Se garantiza la integridad y trazabilidad de los datos, además de la capacidad para realizar operaciones críticas como la compra, transferencia y cancelación de entradas, con las restricciones y condiciones establecidas por las reglas del negocio.

La arquitectura está diseñada para permitir la incorporación posterior de interfaces de usuario o mecanismos externos de pago, manteniendo la independencia entre la lógica del negocio y los componentes de presentación o persistencia. De esta forma, la solución establece una base sólida para la automatización completa del proceso de venta de tiquetes, ofreciendo una herramienta confiable, extensible y alineada con los principios del diseño orientado a objetos.

Objetivos:

La solución tiene como propósito principal modelar e implementar un sistema orientado a objetos que permita gestionar de forma integral el ciclo de vida de los tiquetes de eventos, desde su creación hasta su venta, transferencia y control administrativo.
Para cumplir con este propósito, se plantean los siguientes objetivos específicos:

- Gestionar los actores principales del sistema, incluyendo compradores, organizadores y administradores, garantizando que cada uno cuente con las funcionalidades y restricciones asociadas a su rol.

- Modelar los eventos y sus componentes, permitiendo la creación de eventos con localidades, tipos de tiquetes, precios y capacidad definida.

- Administrar la venta y transferencia de tiquetes, asegurando la unicidad de los identificadores, el control de límites de compra y las condiciones de reembolso o restricción establecidas.

- Implementar mecanismos de persistencia, que aseguren la conservación, recuperación y consistencia de la información sobre usuarios, eventos, tiquetes y transacciones.

- Estructurar una arquitectura modular y escalable, basada en principios de bajo acoplamiento y alta cohesión, que facilite la evolución del sistema sin comprometer su estabilidad.

- Proporcionar una base sólida para futuras extensiones, incluyendo mecanismos de interacción con usuarios, integraciones con pasarelas de pago o visualización de reportes financieros.

No-Objetivos:

Para delimitar el alcance de la solución y mantener el enfoque en la lógica del negocio, se establecen los siguientes no-objetivos:

- No se implementan interfaces gráficas ni web, dado que el propósito se centra en la estructura interna y la lógica del sistema.
- No se considera la seguridad a nivel de encriptación o autenticación avanzada, más allá del manejo básico de credenciales de usuario para evitar complicar mas la ejecución.

Diseño

Elementos que forman la solución:

La aplicación está organizada en módulos/cohesiones funcionales claramente diferenciadas:

- Clases que representan las entidades del negocio (Evento, Venue, Localidad, Tiquete y sus subtipos, Usuario y subtipos, Transacción, Reembolso, Oferta, etc.).

- Clase *App* que implementa los flujos de uso (comprar, transferir, consultar, etc.) y orquesta llamadas entre entidades y persistencia.

- Ficheros JSON ubicados en la carpeta *data/* que contienen la información persistente (eventos, tiquetes, usuarios, transacciones, reembolsos, venues, etc.).

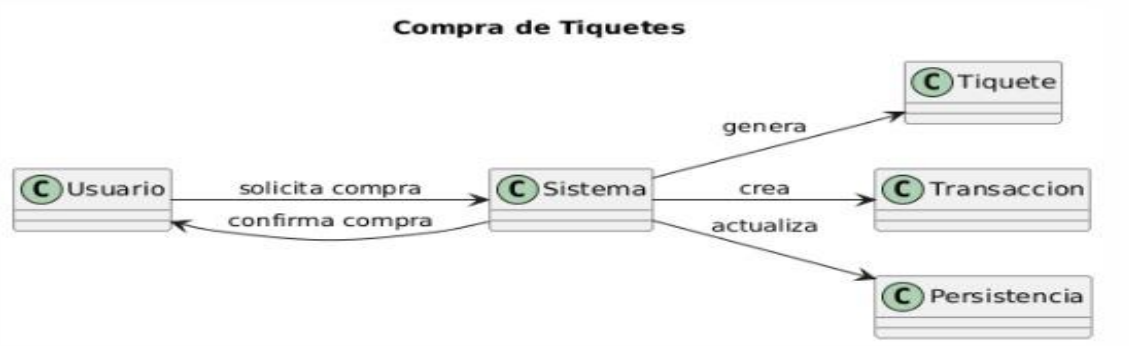
- Clases de apoyo como *IdGenerator* para generación de identificadores únicos.

- Enumeraciones para tipos de evento, estado de tiquete, tipo de localidad, tipos de pago, etc.

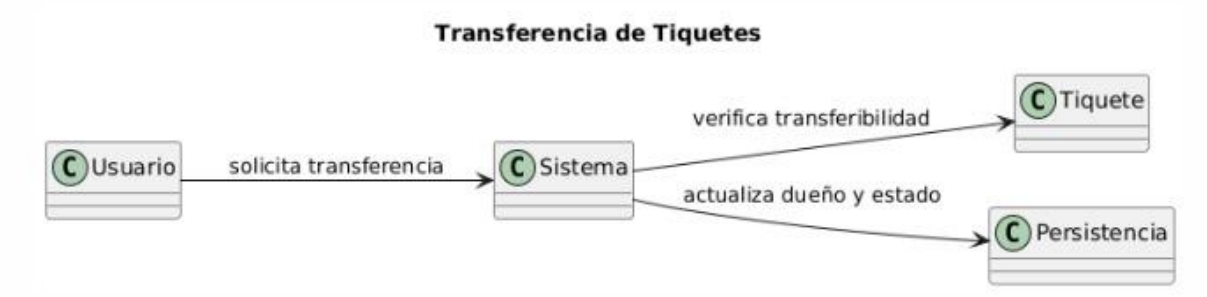
- Clases específicas para manejar transacciones y reembolsos (*Transacción, Reembolso*).

Diagrama de contexto:

El Usuario solicita comprar → el sistema valida límite/capacidad, genera `Tiquete(s)`, crea `Transacción`, actualiza persistencia.

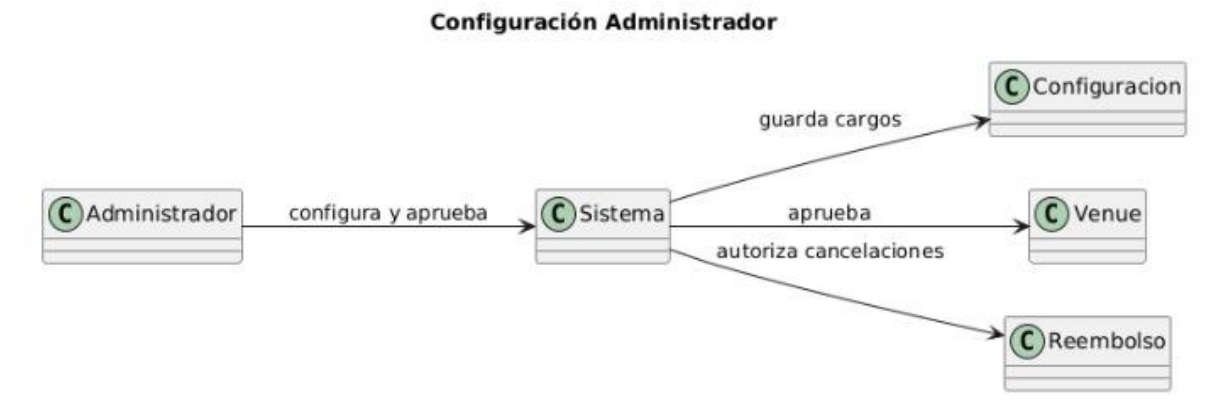


El Usuario solicita transferir → el sistema verifica transferibilidad del tiquete (ej. *`TiqueteDeluxe`* no transferible), valida credenciales y actualiza dueño y estado.



El Organizador crea/edita un Evento ↔ define *`Localidad(es)`*, precios, ofertas.

El Administrador configura cargos (porcentaje por servicio, cuota fija), aprueba venues propuestos, y autoriza reembolsos/cancelaciones.



Persistencia:

Para el caso de la persistencia se usaron dos(2) diferentes las cuales son:

- **Json:** Estos archivos por entidad o lote se encuentran en la carpeta *`data/`* (ej.: *`eventos.json`, `tiquetes.json`, `usuarios.json`, `transacciones.json`, `reembolsos.json`, `venues.json`*). Son responsables de dar o quitar serie a los objetos en puntos de cambio como el inicio y el cierre además de mantener la integridad básica asignando IDs únicos y referencias consistentes, por ejemplo, un *`Tiquete`* referencia *`eventoId`* que debe existir).

  Es ventajoso en el momento de inspeccionar con simplicidad y es suficiente para el alcance actual basado en pruebas y demostraciones, Aunque, no implementa el control de concurrencia y consistencia simultáneos al completo.

- **Base de datos:** Para poder realizar el proyecto de boleta master con el uso de base de datos, requerimos utilizar Xampp, MySQL y el driver MySQL connector. Cada uno tiene una funcionalidad específica para que programa.

    - ***Xampp:***
      El programa Xampp es una herramienta fundamental para poder ejecutar MySQL, debido a que funciona como un servidor local en la computadora. Este guarda todas las tablas de la base de datos.
    - ***JDBC(MySQL connector):***
      El código lo utiliza como " puente" para ejecutar consultas SQL, como el insert.
    - ***MySQL:***
      Para poder implementar las bases de datos, utilizamos MySQL, siendo el sistema de la base de datos para guardar la información.

  El cliente interactúa con la base de datos cada vez que este interactúa con el programa, este último gestiona la base de datos para leer, insertar, actualizar, etc

  La base de datos se conecta MySQL se gestiona por XAMPP, pero para ello, MySQL tiene un modelo de tablas(modelo de datos) siendo un script donde se guarda la

información como Usuarios, Evento, Localidad, Tiquete, transacción

```sql
USE evento_venue;
select * from usuario;

-- VENUE
CREATE TABLE venue (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(150) NOT NULL,
    direccion VARCHAR(255)
);

-- USUARIO
CREATE TABLE usuario (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(100) NOT NULL,
    tipo ENUM('Cliente','Organizador','Administrador') NOT NULL
);

-- EVENTO
CREATE TABLE evento (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    descripcion VARCHAR(255),
    fecha DATE NOT NULL,
    organizador_id INT NOT NULL,
    venue_id INT NOT NULL,
    FOREIGN KEY (organizador_id) REFERENCES usuario(id),
    FOREIGN KEY (venue_id) REFERENCES venue(id)
);

-- LOCALIDAD (cada localidad tiene capacidad y precio unitario)
CREATE TABLE localidad (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100),
    capacidad INT NOT NULL,
    precio DECIMAL(10,2) NOT NULL,
    evento_id INT NOT NULL,
    FOREIGN KEY (evento_id) REFERENCES evento(id)
);

-- TIQUETE
CREATE TABLE tiquete (
    id INT AUTO_INCREMENT PRIMARY KEY,
    tipo ENUM('Normal','Multiple','Deluxe') NOT NULL,
    codigo VARCHAR(100) NOT NULL UNIQUE,
    localidad_id INT NOT NULL,
    asiento_num INT NULL, -- si la localidad es numerada, opcional
    paquete_id INT NULL,  -- para tiquetes múltiples que comparten paquete
    vendido BOOLEAN DEFAULT FALSE,
    propietario_id INT NULL, -- usuario que actualmente posee (null = sin vender)
    FOREIGN KEY (localidad_id) REFERENCES localidad(id),
    FOREIGN KEY (propietario_id) REFERENCES usuario(id)
);
```

La estructura de la tabla de datos tiene un relación entre ella, mediante *foreign keys*, siendo algo similar al single linked list de java, porque cada tabla de datos está vinculada con los de la otra, se hace así para poder evitar datos duplicados.

Las relaciones de las tablas:

Evento→ usuario y venue
localidad→ evento
tiquete → localidad y usuario
transacción → tiquete y usuario

Codigo en conjunto con la base de datos:
Para que Java pudiera interactuar con la base de datos, tuvimos que utilizar JDBC, donde se establece una conexión con MySQL utilizando un usuario y contraseña , siendo root como usuario y 1234 como contraseña.

```
conn = DriverManager.getConnection(URL, USER, PASSWORD);
```

Luego para insertar los datos al momento de que un usuario se registre, se insertan los datos en la tabla del usuario, se hace conn el prepaRedStatement que ejecuta una consulta SQL

```
String sql = "INSERT INTO usuario (nombre, correo, password, tipo) VALUES (?, ?, ?, ?)";
```

Para leer los datos del cliente cuando este quiere ver las transacciones por ejemplo, se consulta a la base de datos para obtener los registros de transacciones asociadas al id de ese cliente en específico y la base de datos devuelve los resultados correspondientes a ese ID.

```
String sql = "SELECT * FROM usuario WHERE id = ?";
```

En el caso que un tiquete sea comprado, se actualiza el estado del tiquete en la base de datos para como vendido
Para esto sirve el SQL que actualiza la columna de vendido de un tiquete, si es vendido a true, y asigna el id del propietario al cliente que lo compro

```
String sql = "SELECT * FROM usuario WHERE correo = ?";
```

¿Cómo funciona?

El primer paso del programa comienza cuando el usuario ingresa sus datos, luego se realiza una consulta en la base de datos para verificar que el tiquete esté disponible, si el tiquete no se ha venido y disponible, se inserta una nueva transacción en la tabla de transacciones, marcando el tiquete como vendido en la tabla tiquete y se asocia el cliente con ese tiquete.

Algoritmos críticos:

Compra de tiquetes – clase *`Transaccion`*

Este método asigna la propiedad del tiquete al comprador y cambia su estado a "Vendido".
Es una parte fundamental del proceso de compra de tiquetes, ya que asegura la unicidad y trazabilidad del propietario.

```
public void marcarVendido(Usuario comprador) {
    this.estado = TipoTiquetes.Vendido;
    this.propietarioActual = comprador;
}
```

Transferencia de tiquetes – clase *`Tiquete`*

El método *transferir()* valida que el tiquete pueda transferirse (según su tipo y estado).
La subclase *TiqueteDeluxe* sobreescribe esta lógica:

```
public boolean transferir(Usuario nuevoPropietario) {
    if (puedeTransferirse() && estado == TipoTiquetes.Vendido) {
        this.propietarioAsignado = nuevoPropietario;
        return true;
    }
    return false;
}
@Override
public boolean puedeTransferirse() { return false; }
```

Esto garantiza que los tiquetes Deluxe no sean transferibles, cumpliendo la regla de negocio indicada en el enunciado del proyecto.

Paquete múltiple – clase *TiqueteMultiple*

Permite distribuir un paquete múltiple de entradas entre varios usuarios, garantizando la consistencia del sistema al verificar la correspondencia entre la cantidad de tiquetes y usuarios.

```
public void asignarTiquetesADistintosUsuarios(List<Usuario> listaUsuarios) {
    if (listaUsuarios.size() != entradas.size()) {
        System.out.println("Error: la cantidad de usuarios no coincide con la cantidad de
tiquetes.");
        return;
    }

    for (int i = 0; i < entradas.size(); i++) {
        TiqueteNormal t = entradas.get(i);
        Usuario u = listaUsuarios.get(i);
        t.setPropietarioAsignado(u);
    }

    System.out.println("Tiquetes asignados correctamente a cada usuario.");
}
```

Creación y validación de eventos – clase *Venue*

Este método valida que un Venue no tenga eventos simultáneos en la misma fecha y hora,
garantizando la integridad de la programación de eventos y evitando solapamientos.

```
public boolean estaDisponible(LocalDate fecha, LocalTime hora) {
    for (Evento e : eventosProgramados) {
        if (e.getFecha().equals(fecha) && e.getHora().equals(hora)) {
            return false;
        }
    }
    return true;
}
```

Cálculo de precio final – clase *Administrador*

Calcula el precio total al cliente, combinando el precio base del tiquete con el recargo porcentual por servicio y el costo fijo de emisión.
Este cálculo se usa en cada compra para reflejar las políticas financieras del sistema.

```
public double calcularCostoTotal(double precioTiquete, TipoDeEvento tipoEvento) {
    double porcentajeServicio = calcularPorcentajeServicio(tipoEvento);
    double recargo = precioTiquete * porcentajeServicio;
    return precioTiquete + recargo + costoEmisionFijo;
}
```

Reembolso – clase *Reembolso*

Define la lógica para aprobar un reembolso: cambia el estado del proceso, registra al aprobador,
y acredita el valor correspondiente al saldo virtual del usuario. Cumple el comportamiento exigido por el administrador.

```
public void aprobar(Administrador aprobador, double valorAcreditado) {
    this.aprobador = aprobador;
    this.estado = "APROBADO";
    this.valorAcreditado = valorAcreditado;
    this.fechaResolucion = LocalDateTime.now();
    double nuevoSaldo = solicitante.getSaldoVirtual() + valorAcreditado;
    solicitante.setSaldoVirtual(nuevoSaldo);
}
```

Alternativas:

Durante el desarrollo del diseño se analizaron diferentes posibilidades técnicas y estructurales que finalmente no fueron adoptadas. Entre ellas, se consideró centralizar la lógica de compra dentro del Administrador, pero se optó por distribuir las responsabilidades mediante la clase Transacción para mantener bajo acoplamiento.

En cuanto a la transferencia de tiquetes, se estudió la opción de permitirla para todos los tipos de tiquete, aunque se desestimó para preservar las reglas de negocio específicas, como la no transferibilidad de los tiquetes Deluxe. En el caso de la creación de eventos, se pensó en un servicio intermedio para validar Venues y fechas, pero se descartó para simplificar el flujo y mantener la lógica cerca de los datos.

Del mismo modo, se consideró incluir el cálculo del precio dentro de la clase Tiquete, aunque se prefirió delegarlo al Administrador para concentrar en un solo punto las políticas económicas. Finalmente, se valoró manejar los reembolsos directamente desde el Administrador o como transacciones inversas, pero se optó por una clase dedicada que ofreciera mayor claridad y trazabilidad del proceso.

En conjunto, las alternativas no implementadas fueron descartadas principalmente por aumentar la complejidad, generar acoplamiento innecesario o dispersar la lógica.

A considerar:

Durante el diseño se tuvieron en cuenta varios aspectos transversales que afectan el funcionamiento general del sistema, la calidad del código y su mantenimiento a largo plazo.

En cuanto a la seguridad, se definió que todos los usuarios deben autenticarse mediante un login y una contraseña. Aunque el alcance del proyecto no contempla mecanismos avanzados de cifrado o recuperación de credenciales, se estructuró el sistema para que en futuras versiones pueda integrarse una capa de seguridad más robusta sin modificar la lógica principal. Además, se restringió el acceso a ciertas operaciones, como la aprobación de eventos o la gestión de tarifas, exclusivamente al Administrador.

Respecto a la persistencia, se buscó mantener la consistencia de los datos entre ejecuciones del programa. Se adoptó principalmente la idea de usar Json como forma de manejo de la información de los usuarios sumado a una segunda versión que usa bases de datos con SQL para verificar el panorama que ambas ofrecen.

En términos de transaccionalidad, se procuró que las operaciones críticas, como la compra o el reembolso de tiquetes, sean atómicas dentro del flujo del programa. Cada transacción modifica los estados de los objetos involucrados de manera controlada, evitando inconsistencias o pérdidas de información. La clase *Transaccion* centraliza estas operaciones, actuando como punto de control del proceso.

Por último, el sistema fue diseñado con un enfoque modular. Se emplea herencia para los distintos tipos de tiquetes y composición para la relación entre eventos, localidades y venues. Esto facilita futuras ampliaciones, como nuevos métodos de pago, tipos de eventos o mecanismos de promoción, sin necesidad de alterar significativamente la estructura existente.

Algoritmos críticos:

Rol: Administrador

HU1. Aprobar nuevos promotores

Como administrador,
quiero revisar y aprobar las solicitudes de registro de nuevos promotores,
para garantizar que solo usuarios válidos puedan crear eventos.
Entradas: listado de promotores pendientes, acción “aprobar” o “rechazar”.
Salidas: cambio de estado del promotor en usuarios.json a “aprobado”; mensaje de confirmación.

HU2. Gestionar eventos del sistema

Como administrador,
quiero ver, modificar o eliminar eventos activos,
para mantener la integridad de los datos en el sistema.
Entradas: selección de evento, acción (editar/eliminar).
Salidas: actualización de eventos.json; confirmación de la acción.

HU3. Revisar transacciones y reembolsos

Como administrador,
quiero acceder al historial de transacciones y solicitudes de reembolso,
para auditar la actividad del sistema.
Entradas: consulta a transacciones.json o reembolsos.json.
Salidas: reporte detallado con ID de usuario, evento, monto, estado.

HU4. Monitorear actividad del sistema

Como administrador,
quiero consultar registros de log del sistema,
para identificar errores o actividades sospechosas.
Entradas: comando o menú para acceder a log/EntradaLog.
Salidas: listado de entradas con fecha, usuario y acción registrada.

Rol: Organizador

HU5. Registro de promotor

Como promotor,
quiero registrarme en el sistema con mis datos personales,
para poder solicitar aprobación del administrador y comenzar a crear eventos.
Entradas: nombre, correo, contraseña, tipoUsuario=“promotor”.
Salidas: usuario creado con estado “pendiente de aprobación”; mensaje de espera.

HU6. Crear evento

Como promotor aprobado,
quiero crear un nuevo evento especificando lugar, fecha, localidades y tipo de tiquetes,
para ofrecerlo a los clientes.
Entradas: nombre del evento, tipo (concierto, obra, etc.), fecha, venue, localidades, precios.
Salidas: registro nuevo en eventos.json; confirmación del sistema.

HU7. Definir localidades y tipos de tiquetes

Como promotor,
quiero configurar las localidades y tipos de tiquetes (normal, deluxe, múltiple),
para personalizar la venta del evento.

Entradas: tipoLocalidad, capacidad, precio, tipoTiquete.
Salidas: actualización en eventos.json y tiquetes.json.

HU8. Consultar ventas de mis eventos

Como promotor,
quiero ver las ventas y transacciones asociadas a mis eventos,
para analizar el rendimiento comercial.
Entradas: ID del evento o nombre.
Salidas: resumen de ventas (entradas vendidas, ingresos totales) desde transacciones.json.

HU9. Solicitar reembolsos o cancelación

Como promotor,
quiero cancelar un evento o tramitar reembolsos cuando sea necesario,
para mantener la satisfacción de los clientes.
Entradas: evento, motivo de cancelación.
Salidas: registros en reembolsos.json, notificación a los clientes afectados.

Rol: Cliente

HU10. Registro de cliente

Como cliente,
quiero crear una cuenta con mis datos personales,
para poder comprar tiquetes a eventos.
Entradas: nombre, correo, contraseña, tipoUsuario=“cliente”.
Salidas: usuario registrado con acceso inmediato; mensaje de confirmación.

HU11. Consultar eventos disponibles

Como cliente,
quiero ver los eventos disponibles en la plataforma,
para elegir cuál quiero asistir.
Entradas: acción “ver eventos”.
Salidas: lista de eventos activos con nombre, fecha, lugar, y precio.

HU12. Comprar tiquete

Como cliente,
quiero seleccionar un evento, una localidad y un tipo de tiquete,
para completar mi compra.
Entradas: selección de evento, localidad, tipo de tiquete, medio de pago.
Salidas: registro de compra en tiquetes.json y transacciones.json; confirmación con ID de compra.

HU13. Solicitar reembolso

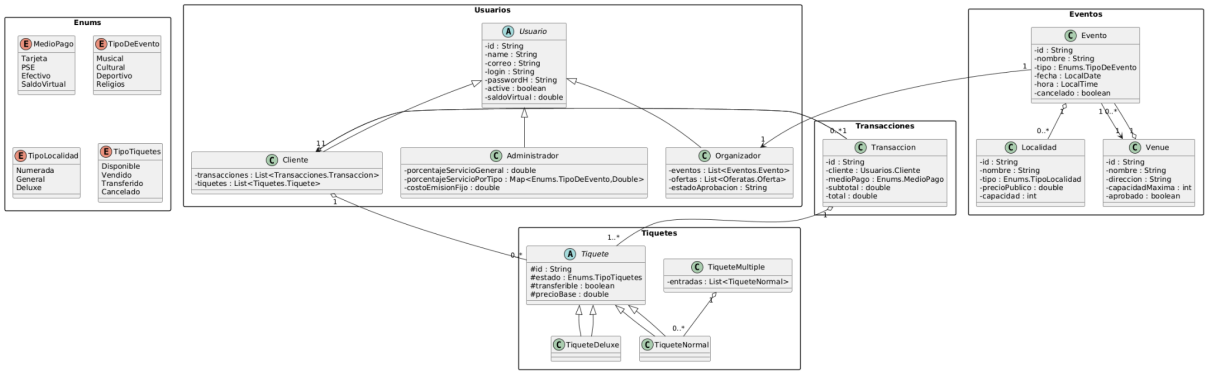
Como cliente,
quiero solicitar el reembolso de un tiquete comprado,
para recuperar mi dinero en caso de cancelación o error.
Entradas: ID de transacción, motivo del reembolso.
Salidas: registro en reembolsos.json, mensaje de recepción de solicitud.

HU14. Consultar historial de compras

Como cliente,
quiero revisar mis compras anteriores,
para verificar mis tiquetes y eventos asistidos.
Entradas: acción “ver historial”.
Salidas: listado con ID de tiquete, evento, fecha, monto.

Diagramas de funcionamiento:

Diagrama nucleo del sistema



Paquete Enums

MedioPago: Tarjeta, PSE, Efectivo, SaldoVirtual → se usa en Transaccion.

TipoDeEvento: Musical, Cultural, Deportivo, Religios → se usa en Evento y en el cálculo de recargos del admin.

TipoLocalidad: Numerada, General, Deluxe → se usa en Localidad.

TipoTiquetes: Disponible, Vendido, Transferido, Cancelado → estado de cada tiquete.

Paquete Usuarios

Clases:

Usuario (abstracta):

Es la base. Tiene:

id, name, correo, login, passwordH

active, saldoVirtual

Métodos típicos como validarPassword, getters…

Cliente:

Hereda de Usuario.

Tiene:

transacciones : List<Transaccion>

tiquetes : List<Tiquete>

Conecta directo con:

HU10 (registro de cliente)

HU12 (comprar tiquete)

HU14 (historial de compras)

Organizador:

Hereda de Usuario.

Tiene:

eventos : List<Evento>

ofertas : List<Oferta>

estadoAprobacion : String (“pendiente”, “aprobado”, “rechazado”)

Relacionado con:

HU5 (registro de promotor, estado “pendiente”)

HU1 (admin aprueba/rechaza al promotor)

HU6, HU7, HU8, HU9 (eventos, localidades, ventas, reembolsos)

Administrador:

Hereda de Usuario.

Tiene todo lo de recargos y administración:

porcentajeServicioGeneral

porcentajeServicioPorTipo

costoEmisionFijo

Métodos tipo:

calcularPorcentajeServicio(tipo)

calcularCostoTotal(precioBase, tipoEvento)

aprobarVenue, agregarVenuePendiente

aprobarPromotor(Organizador), rechazarPromotor(Organizador)

Relacionado con:

HU1, HU2, HU3, HU4, HU9, HU13

Paquete Eventos

Clases:

Evento:

Tiene:

id, nombre, tipo, fecha, hora

organizador, venue

cancelado : boolean

localidades : List<Localidad>

totalTiquetes

Relaciones:

1 Evento → muchas Localidad

1 Evento → 1 Organizador

1 Evento → 1 Venue

Conecta con:

HU6 (crear evento)

HU7 (definir localidades)

HU9 (cancelar evento)

Localidad:

Tiene:

id, nombre, tipo (TipoLocalidad)

precioPublico, capacidad

tiquetes : List<Tiquete>

Se encarga de generar tiquetes, saber cuántos hay disponibles, etc.

Relación:

1 Localidad → muchos Tiquete

HU7 puro y duro.

Venue:

Tiene:

id, nombre, dirección, capacidadMaxima, aprobado

eventosProgramados : List<Evento>

Método clave: estaDisponible(fecha, hora)

Relación:

1 Venue → muchos Evento

Se usa en:

HU2 (gestionar eventos)

HU6 (crear evento con venue disponible)

Paquete Tiquetes

Clases:

Tiquete (abstracta):

Campos:

id, evento, localidad

estado : TipoTiquetes

transferible

precioBase

propietarioActual, propietarioAsignado, etc.

Métodos clave:

marcarVendido(Cliente)

transferir(Cliente)

puedeTransferirse() (abstracto)

TiqueteNormal:

Es un tiquete estándar.

Suele ser transferible.

Representa HU12, HU6, HU7.

TiqueteDeluxe:

Extiende Tiquete.

Tiene:

beneficios : List<String> (backstage, comida, etc.)

HU7: tipos de tiquete especiales.

TiqueteMultiple:

Es un “paquete” de tiquetes normales.

Contiene:

entradas : List<TiqueteNormal>

campos como transferibleParcial, bloqueado.

Permite asignar cada entrada a usuarios distintos.

Relaciones mas importantes

Tiquete → Evento y Localidad (sabe de dónde es).

Cliente → muchos Tiquete (lo que el cliente ha comprado).

TiqueteMultiple → varios TiqueteNormal.

Paquete Transacciones

Transaccion:

Campos:

id, cliente, fecha, medioPago

items : List<Tiquete>

subtotal, recargoServicio, costoEmision, descuento, total

Métodos:

agregarTiquete(...)

calcularTotales(...)

confirmar()

Relaciones:

1 Transaccion → 1 Cliente

1 Transaccion → muchos Tiquete

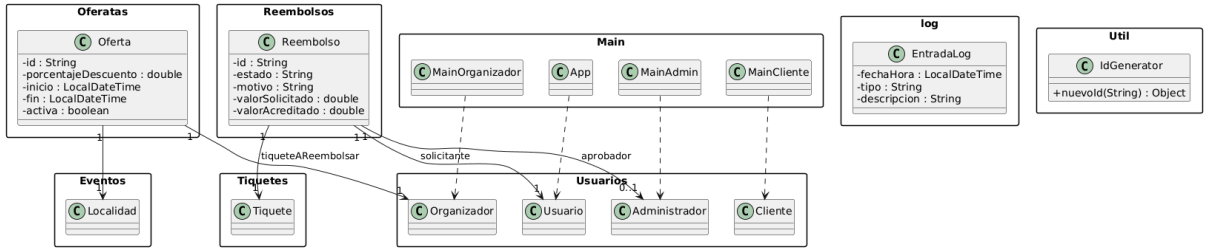
1 Transaccion → 1 MedioPago

Se encuentra en :

HU12 (comprar tiquete)

HU8 y HU14 (ventas e historial)

HU3 (auditar transacciones)



1. Oferatas.Oferta

Campos:

id

creador : Organizador

localidadObjetivo : Localidad

porcentajeDescuento

inicio, fin

activa : boolean

Métodos:

estaVigente() (verifica fechas + activa)

Relación:

1 Oferta → 1 Organizador

1 Oferta → 1 Localidad

Relacionado con:

HU7 (configuración avanzada de precios)

HU6/HU12 cuando se quiere aplicar descuentos.

2. Reembolsos.Reembolso

Campos:

id

aprobador : Administrador (puede ser null al inicio)

solicitante : Usuario

tiqueteAReembolsar : Tiquete

fechaSolicitud

estado (PENDIENTE, APROBADO, RECHAZADO)

motivo

valorSolicitado, valorServicio, valorAcreditado

Relaciones:

1 Reembolso → 1 Usuario (el que lo pide)

1 Reembolso → 0..1 Administrador (el que lo aprueba)

1 Reembolso → 1 Tiquete (el que se reembolsa)

Se conecta con:

HU9 (promotor tramita reembolsos/cancelación)

HU13 (cliente solicita reembolso)

HU3 (admin revisa reembolsos)

log.EntradaLog

Registra lo que pasa en el sistema:

fechaHora

tipo (INFO, ERROR, WARN…)

descripcion

Se relaciona con:

HU4 (monitorear actividad del sistema / log).

Util.IdGenerator

Clase de utilidad:

Método nuevoId(String prefijo) o algo similar.

Paquete Main

Clases de arranque y menú:

App: carga JSON, orquesta, muestra menú general.

MainAdmin: flujo específico para el admin.

MainCliente: flujo para cliente.

MainOrganizador: flujo para organizador.

En el diagrama se tiene:

MainMainAdmin ..> Administrador

MainMainCliente ..> Cliente

MainMainOrganizador ..> Organizador

Test

TransaccionTest

a) RF_compra_agregarTiqueteDebeAgregarItemsALaLista()

Qué hace: crea un Cliente y una Transaccion vacía, y agrega dos TiqueteNormal a la transacción.

Qué válida:

Que al inicio la lista de ítems está vacía y que después de llamar agregar Tiquete dos veces, el tamaño de la lista es 2.

b) RF_compra_calcularTotalesDebeSumarSubtotalRecargoEmisionYDescuento()

Qué hace:

Crea una transacción con dos tiquetes de 100.000 y 200.000.

Llama a calcularTotales(porcentajeServicio, costoEmisionFijo, descuento, etc.

Qué válida:

Qué subtotal sea igual a la suma de precios base.

Que recargoServicio = subtotal * porcentaje.

Que costoEmision = costo fijo por tiquete * número de tiquetes.

Que total = subtotal + recargo + emisión − descuento.

c) RF_compra_confirmarDebeMarcarTiquetesComoVendidosAlCliente()

Qué hace:

Crea cliente, transacción y dos tiquetes.

Los agrega a la transacción y llama tx.confirmar().

Qué válida:

Que los tiquetes quedan en estado Vendido.

Que el propietario Actual de cada tiquete es el cliente.

2. ClienteTest

RF_compra_clienteDebeRegistrarTransaccionYTiquetes()

Qué hace:

Crea un cliente y una transacción con dos tiquetes para luego llamar cliente.compra(tx).

Qué válida:

Que en cliente.get Transacciones() ahora hay 1 transacción, que es la misma que creamos y que en cliente.getTiquetes() hay los 2 tiquetes comprados.

3. AdministradorTest

a) RF_costos_adminUsaPorcentajeEspecificoCuandoExiste()

Qué hace:

Crea un Administrador.

Define un porcentaje general

Define un porcentaje específico para eventos Musical

Pide el porcentaje para Musical y para Cultural.

Qué válida:

Que para Musical se usa el porcentaje específico
Que para otros tipos (sin configuración específica) se usa el general

b) RF_costos_calcularCostoTotalDebeIncluirRecargoYEmisionFija()
Qué hace:
Configura un admin con un porcentaje de servicio y un costo fijo de emisión.
Llama calcularCostoTotal(precioBase, TipoDeEvento.Musical).
Qué válida:
Que el costo total = precio base + recargo por porcentaje + costo fijo de emisión.

4. TiqueteBaseTest (lógica genérica de Tiquete)
a) RF_tiquete_transferirSoloFuncionaSiEstaVendidoYEsTransferible()
Qué hace:
Crea un tiquete de prueba (TiquetePrueba) marcado como transferible.
Primero intenta transferirlo sin haberlo vendido → debería fallar.
Luego llama marcarVendido(c1) y vuelve a intentar transferir a c2.
Qué valida:
Que no se puede transferir un tiquete que sigue en estado Disponible.
Que sí se puede transferir cuando está Vendido y es transferible.
Que el propietarioAsignado queda siendo el nuevo cliente.

b) RF_tiquete_noSeTransfiereSiNoEsTransferible()
Qué hace:
Crea un tiquete marcado como no transferible.
Lo marca como vendido.
Intenta transferirlo.
Qué valida:
Que transferir devuelve false.
Que no se asigna nuevo propietario (propietarioAsignado sigue null).

5. TiqueteTiposTest
a) RF_tiqueteNormal_debeSerTransferiblePorDefecto()
Qué hace: crea un TiqueteNormal.
Qué valida: que isTransferible() sea true.

b) RF_tiqueteDeluxe_noDebeSerTransferibleYPuedeGestionarBeneficios()
Qué hace:
Crea un TiqueteDeluxe.
Revisa que inicialmente no sea transferible y que su lista de beneficios esté vacía.
Agrega dos beneficios y elimina uno.
Qué valida:
Que el deluxe no es transferible.
Que se pueden agregar y quitar beneficios de la lista.

c) RF_tiqueteMultiple_asignarTiquetesADistintosUsuarios()
Qué hace:

Crea un TiqueteMultiple con dos TiqueteNormal dentro.

Crea dos usuarios (cliente 1 y cliente 2).
Llama asignarTiquetesADistintosUsuarios(listaUsuarios).
Qué valida:
Que el primer tiquete del paquete queda asignado al primer usuario.
Que el segundo tiquete queda asignado al segundo usuario.

6. OfertaTest
a) RF_oferta_activaYDentroDelRangoDebeEstarVigente()
Qué hace:
Crea una Oferta cuya ventana de tiempo incluye la hora actual.
activa = true por defecto.
Qué valida:
Que isActiva() es true.
Que estaVigente() devuelve true.

b) RF_oferta_fueraDelRangoNoDebeEstarVigente()
Qué hace:
Crea una oferta cuya ventana de tiempo ya pasó.
Qué valida:
Que estaVigente() es false.

c) RF_oferta_inactivaNoDebeEstarVigenteAunqueFechasSeanValidas()
Qué hace:
Crea una oferta con fechas válidas, pero luego llama setActiva(false).
Qué valida:
Que estaVigente() sea false aunque la fecha actual esté dentro del rango.

7. VenueTest
a) RF_venue_sinEventosDebeEstarDisponibleSiempre()
Qué hace:
Crea un Venue sin eventos.
Pregunta estaDisponible para una fecha/hora.
Qué valida:
Que la respuesta es true.

b) RF_venue_noDisponibleMismaFechaYHoraDeEvento()
Qué hace:
Crea un venue y un evento en ese venue en cierta fecha y hora.
Pide estaDisponible en esa misma fecha y hora.
Qué valida:
Que la respuesta es false.

\
c) RF_venue_disponibleOtraHoraUOtroDia()
Qué hace:
Mismo evento que antes, pero pregunta:
otra hora del mismo día,
mismo horario pero otro día.
Qué valida:
Que en esos casos el venue sí está disponible.