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

Repositorios proyecto 1:

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

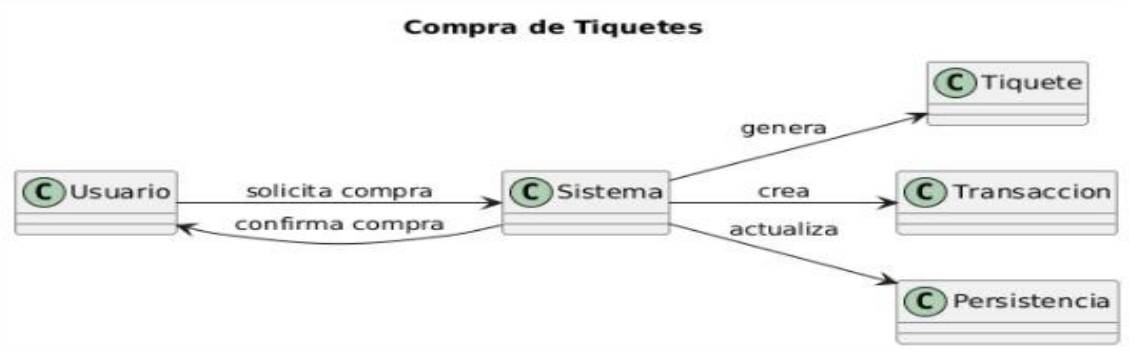
Elementos que forman la solución:

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

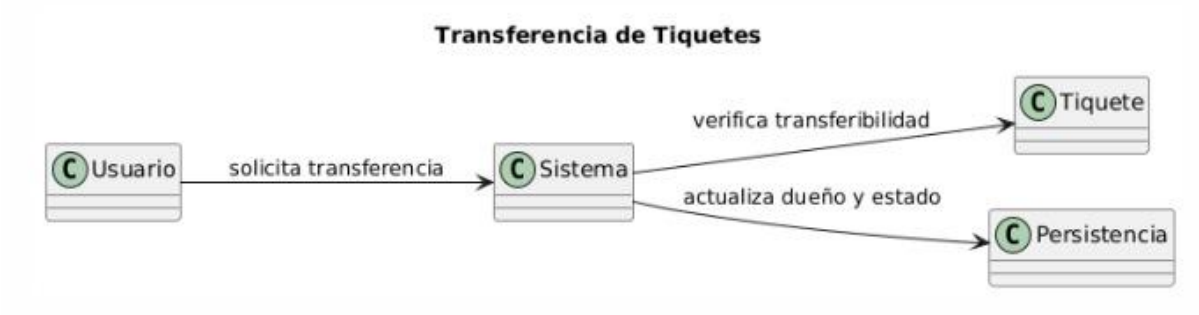
- Clases que representan las entidades del negocio (Evento, Venue, Localidad, Tiquete y sus subtipos, Usuario y subtipos, Transacción, Reembolso, Oferta, etc.).
- Clase *App* que implementa los flujos de uso (comprar, transferir, consultar, etc.) y orquesta llamadas entre entidades y persistencia.
- 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.

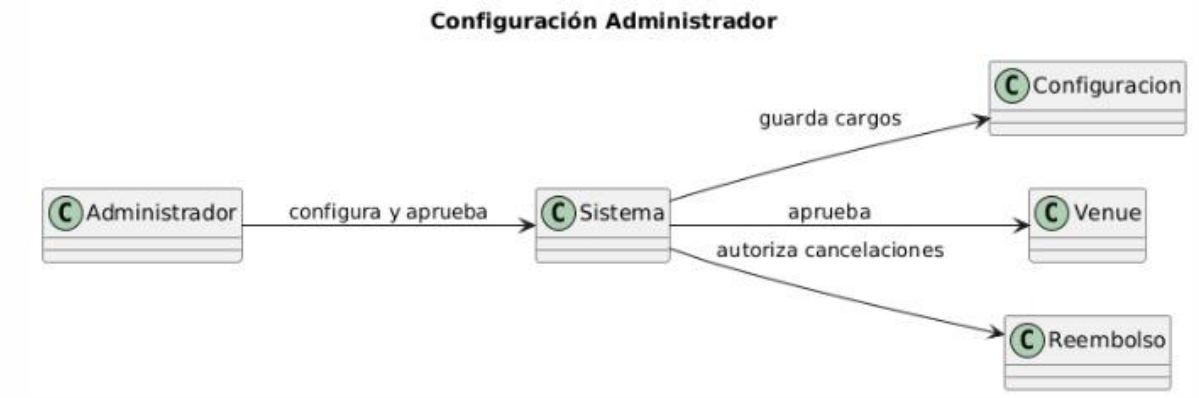


El Usuario solicita transferir → el sistema verifica transferibilidad del tiquete (ej. `TiqueteDeluxe` no transferible), valida credenciales y actualiza dueño y estado.



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.