Isabella Bracho 202423192
Martín De Angulo 202421628

Documento de Diseño

Contexto y Análisis del problema

La implementación es una aplicación de boletería digital orientada a la creación, publicación y gestión de eventos, con tres roles principales: Administrador, Organizador y Cliente. El sistema permite registrar venues, aprobarlos, definir localidades y emitir tiquetes; además, los usuarios pueden comprar, transferir y solicitar reembolsos, mientras que la administración controla los cargos de servicio, cancelaciones y reportes de ganancias.

El dominio implementado incluye las clases Evento, Venue, Localidad, Tiquete (con subclases como Simple, Numerado, Múltiple y Paquete Deluxe), así como las entidades de usuario Administrador, Organizador y Cliente. Cada componente tiene tareas específicas. El organizador crea eventos y define precios, el cliente realiza compras y transferencias, y el administrador supervisa la operación fijando cargos, aprobando venues y gestionando cancelaciones.

Las reglas del negocio se reflejan directamente en el código:

- Restricciones de transferencia
- Estados del evento (Activo, Cancelado)
- Aprobación obligatoria de venues antes de su uso
- Reembolsos condicionados al tipo de cancelación (por administrador u organizador).
- Cálculo de precios finales con cargos fijos definidos por el administrador.

El sistema utiliza una persistencia en formato JSON para almacenar usuarios, eventos y venues, garantizando la conservación del estado entre ejecuciones. Toda la interacción se realiza desde una consola de demostración, que ejecuta casos de prueba (creación, compra, transferencia y cancelación de eventos).

Primero (en la primera entrega) se construyó un modelo UML para identificar las entidades y relaciones generales. Sin embargo, al llevarlo a código el modelo se cambió un poco para lograr la implementación

Diagrama de alto nivel

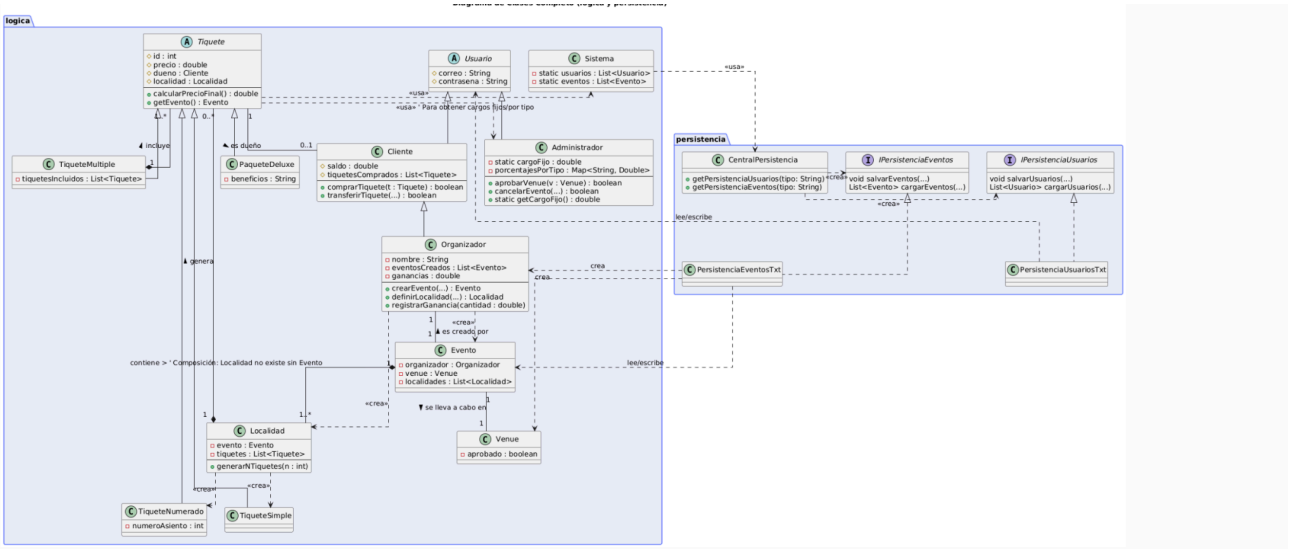
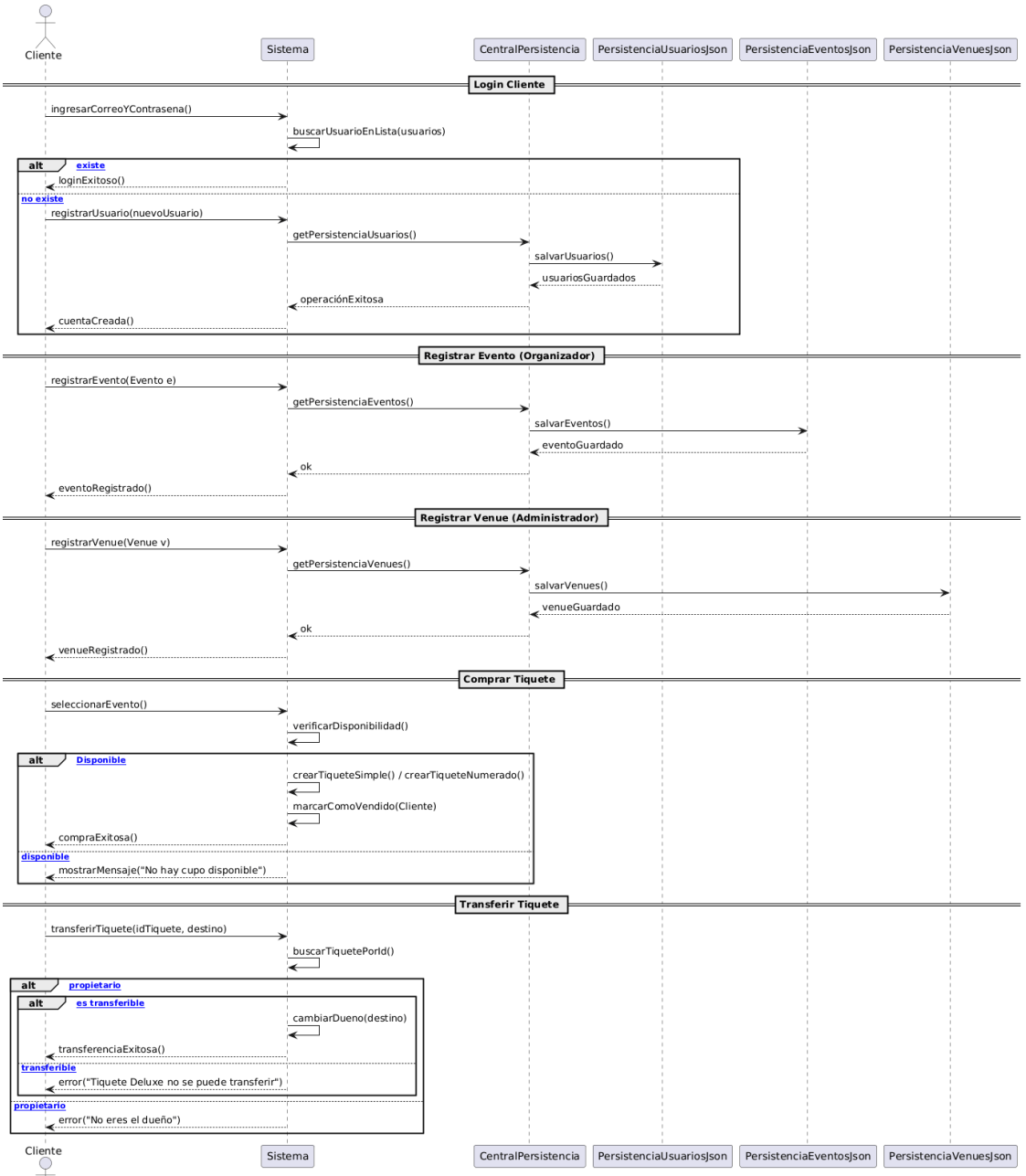


**Imagen en el repositorio

Diagramas de secuencia



Los diagramas de secuencia representan cómo interactúan los distintos componentes del sistema durante la ejecución de los casos de uso principales. En este caso, modelamos las interacciones entre usuarios, el Sistema principal, y los módulos de persistencia responsables del manejo de los datos en formato JSON.

El diagrama corresponde a algunos casos: Login de cliente, Registro de evento, Registro de venue, Compra de tiquete y Transferencia de tiquete.

Cada uno de estos procesos muestra las interacciones entre las capas lógica (Sistema) y persistencia (CentralPersistencia y las implementaciones de la persistencia), mostrando el flujo de información desde la entrada del usuario hasta la actualización de los datos que se guardan en la persistencia.

Implementación

1) **Clase Usuario:**

```
package logica;
public abstract class Usuario {
  protected String correo;
  protected String contrasena;
  public Usuario(String correo, String contrasena) {
     this.correo = correo;
     this.contrasena = contrasena;
  }
  public String getCorreo() {
     return correo;
  }
  public String getContrasena() {
     return contrasena;
  }
}
```

Funcionalidad:

La clase Usuario define la estructura base para los tipos de usuario dentro del sistema (administrador, cliente, organizador). Su función es centralizar los atributos y comportamientos esenciales relacionados con la identidad y autenticación de los usuarios, asegurando la coherencia y reutilización de código en las subclases que heredan de ella.

Al ser una clase abstracta, no puede instanciarse directamente, sino que sirve como plantilla para la creación de tipos de usuario específicos que implementan las funcionalidades particulares de cada rol dentro del sistema.

2) **Clase Administrador**

```java
package logica;
import java.util.HashMap;
import java.util.Map;
public class Administrador extends Usuario {
  private static double cargoFijo = 1.0;
  private Map<String, Double> porcentajesPorTipo;
  public Administrador(String correo, String contrasena) {
     super(correo, contrasena);
     this.porcentajesPorTipo = new HashMap<>();
  }
  public boolean aprobarVenue(Venue v) {
     try {
        if (v == null) {
         throw new Exception("Venue inválido");
        }
        v.setAprobado(true);
        return true;
     }
     catch (Exception e) {
        System.out.println("Error al aprobar venue");
        e.printStackTrace();
        return false;
     }
  }
  public boolean cancelarEvento(Evento e, boolean porAdministrador) {
     try {
        if (e == null) {
         throw new Exception("Evento inválido");
        }
        e.setEstado("Cancelado");
        if (porAdministrador) {
           for (Tiquete t : e.getAllTiquetesVendidos()) {
              Cliente dueño = t.getDueno();
              if (dueño != null) {
                 double reembolso = t.getPrecio();
                 dueño.setSaldo(dueño.getSaldo() + reembolso - cargoFijo);
              }
           }
        }
        return true;
     }
     catch (Exception ex) {
        System.out.println("Error al cancelar evento");
        ex.printStackTrace();
```

```
        return false;
    }
  }
  public void fijarCargoPorTipo(String tipo, double porcentaje) {
    porcentajesPorTipo.put(tipo, porcentaje);
  }
  public Double getCargoPorTipo(String tipo) {
    return porcentajesPorTipo.getOrDefault(tipo, 0.0);
  }
  public void setCargoFijo(double cargo) {
        cargoFijo = cargo;
  }
  public static double getCargoFijo() {
    return cargoFijo;
  }
}
```

Funcionalidad:

La clase Administrador representa al usuario con más permisos dentro del sistema. Su función principal es aprobar venues, cancelar eventos y definir los cargos o porcentajes según el tipo de evento.

También se encarga de manejar un cargo fijo que se usa, por ejemplo, cuando se hacen reembolsos, y una lista de porcentajes por tipo de evento para calcular los cobros adicionales.

3) Clase Organizador

```
package logica;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class Organizador extends Cliente {
  private String nombre;
  private List<Evento> eventosCreados;
  private double ganancias;
  public Organizador(String correo, String contrasena, String nombre) {
    super(correo, contrasena);
    this.nombre = nombre;
    this.eventosCreados = new ArrayList<>();
    this.ganancias = 0.0;
  }
  public String getNombre() {
    return nombre;
```

```
  }
  public List<Evento> getEventosCreados() {
     return eventosCreados;
  }
   public Evento crearEvento(String nombreEvento, LocalDateTime fechaHora, String
tipo, Venue venue) {
     try {
        if (venue == null) {
         throw new Exception("Venue inválido");
        }
        if (!venue.estaAprobado()) {
         throw new Exception("Venue no aprobado");
        }
        if (!venue.estaDisponible(fechaHora)) {
         throw new Exception("Venue ocupado en esa fecha");
        }
        Evento e = new Evento(nombreEvento, fechaHora, tipo, this, venue);
        eventosCreados.add(e);
        return e;
     }
     catch (Exception e) {
        System.out.println("Error al crear evento");
        e.printStackTrace();
        return null;
     }
  }
     public Localidad definirLocalidad(String nombreLocalidad, boolean numerada,
double precioBase, int capacidad, Evento evento) {
     try {
        if (evento == null) {
         throw new Exception("Evento no válido");
        }
                Localidad l = new Localidad(nombreLocalidad, numerada, precioBase,
capacidad, evento);
        evento.agregarLocalidad(l);
        return l;
     }
     catch (Exception e) {
        System.out.println("Error al definir localidad");
        e.printStackTrace();
        return null;
     }
  }
  public void establecerPrecio(Localidad l, double nuevoPrecio) {
```

```
        try {
            if (l == null) {
             throw new Exception("Localidad no válida");
            }
            l.setPrecioBase(nuevoPrecio);
        }
        catch (Exception e) {
            System.out.println("Error al establecer precio");
            e.printStackTrace();
        }
    }
    public double consultarGanancias() {
        return ganancias;
    }
    public void registrarGanancia(double cantidad) {
        ganancias += cantidad;
    }
}
```

La clase Organizador representa a los usuarios encargados de crear y administrar eventos dentro del sistema. Puede registrar nuevos eventos, definir localidades (como secciones o zonas del evento) y establecer precios para cada una.Además, el organizador lleva un registro de las ganancias obtenidas por sus eventos.

4) **Clase Cliente**

```
package logica;
import java.util.ArrayList;
import java.util.List;
public class Cliente extends Usuario {
    protected double saldo;
    protected List<Tiquete> tiquetesComprados;
    public Cliente(String correo, String contrasena) {
        super(correo, contrasena);
        this.saldo = 0.0;
        this.tiquetesComprados = new ArrayList<>();
    }
    public double getSaldo() {
        return saldo;
    }
```

```
  public void setSaldo(double saldo) {
     this.saldo = saldo;
  }
  public List<Tiquete> getTiquetesComprados() {
     return tiquetesComprados;
  }
  public boolean comprarTiquete(Tiquete t) {
     try {
        if (t == null) {
              throw new Exception("Tiquete no válido");
        }
        if (!t.isDisponible()) {
              throw new Exception("Tiquete no disponible");
        }

        double total = t.calcularPrecioFinal();

        if (saldo < total) {
              throw new Exception("Saldo insuficiente");
        }
        saldo -= total;
        boolean ok = t.marcarComoVendido(this);
        if (!ok) {
              throw new Exception("Error al marcar el tiquete");
        }
        tiquetesComprados.add(t);
                        Organizador org = t.getEvento() != null ?
t.getEvento().getOrganizador() : null;
        if (org != null) {
              org.registrarGanancia(t.getPrecio());
        }
        return true;
     }
     catch (Exception e) {
        System.out.println("Error en compra");
        return false;
     }
```

```
  }
      public boolean transferirTiquete(Tiquete t, Cliente destino, String
contrasenaTransferidor) {
     try {
        if (t == null || destino == null) {
                throw new Exception("Datos inválidos");
        }
        if (!this.contrasena.equals(contrasenaTransferidor)) {
                throw new Exception("Contraseña incorrecta");
        }
        if (!tiquetesComprados.contains(t)) {
                throw new Exception("Tiquete no pertenece al cliente");
        }
        if (!t.isTransferible()) {
                throw new Exception("Tiquete no transferible");
        }
        if (t instanceof PaqueteDeluxe) {
                throw new Exception("Paquete Deluxe no se puede
transferir");
        }
        tiquetesComprados.remove(t);
        destino.tiquetesComprados.add(t);
        t.setDueno(destino);
        return true;
     }
     catch (Exception e) {
        System.out.println("Error en transferencia");
        e.printStackTrace();
        return false;
     }
  }
  public boolean solicitarReembolso(Tiquete t) {
     try {
        if (t == null) {
                throw new Exception("Tiquete inválido");
        }
        if (!tiquetesComprados.contains(t)) {
```

```
                throw new Exception("Tiquete no encontrado");
            }
            if (t.isVencido()) {
                throw new Exception("Tiquete vencido");
            }
            double reembolso = t.getPrecio();
            tiquetesComprados.remove(t);
            saldo += reembolso;
            t.setDisponible(true);
            t.setDueno(null);
            return true;
        }
        catch (Exception e) {
            System.out.println("Error en reembolso");
            e.printStackTrace();
            return false;
        }
    }
}
```

Funcionalidad:

La clase Cliente representa a los usuarios que compran, transfieren o solicitan reembolsos de tiquetes en el sistema. Cada cliente tiene un saldo virtual con el que realiza las compras y una lista de tiquetes adquiridos. Entre sus principales acciones están: Comprar tiquetes, verificando disponibilidad y saldo; transferir tiquetes a otros clientes, siempre que sean transferibles; solicitar reembolsos, si el evento no ha ocurrido y el tiquete no está vencido. En pocas palabras, el cliente es el usuario final del sistema, que interactúa directamente con los eventos y tiquetes.

5) **Clase Evento**

```
package logica;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class Evento {
    private String nombre;
    private LocalDateTime fechaHora;
    private String tipo;
    private String estado;
```

```
  private Organizador organizador;
  private Venue venue;
  private List<Localidad> localidades;
    public Evento(String nombre, LocalDateTime fechaHora, String tipo, Organizador
organizador, Venue venue) {
      this.nombre = nombre;
      this.fechaHora = fechaHora;
      this.tipo = tipo;
      this.estado = "Activo";
      this.organizador = organizador;
      this.venue = venue;
      this.localidades = new ArrayList<>();
  }
  public String getNombre() {
      return nombre;
  }
  public LocalDateTime getFechaHora() {
      return fechaHora;
  }
  public String getTipo() {
      return tipo;
  }
  public String getEstado() {
      return estado;
  }
  public void setEstado(String estado) {
      this.estado = estado;
  }
  public Organizador getOrganizador() {
      return organizador;
  }
  public Venue getVenue() {
      return venue;
  }
  public List<Localidad> getLocalidades() {
      return localidades;
  }
  public void agregarLocalidad(Localidad l) {
      if (l != null && !localidades.contains(l)) {
          localidades.add(l);
      }
  }
  public List<Tiquete> getAllTiquetesVendidos() {
      List<Tiquete> vendidos = new ArrayList<>();
```

```
    for (Localidad l : localidades) {
        for (Tiquete t : l.getTiquetes()) {
            if (!t.isDisponible()) {
                vendidos.add(t);
            }
        }
    }
    return vendidos;
  }
}
```

Funcionalidad:

La clase Evento representa cada evento creado por un organizador dentro del sistema. Contiene la información principal del evento, como su nombre, fecha y hora, tipo, estado, venue y las localidades disponibles. Permite agregar localidades y consultar todos los tiquetes vendidos de esas localidades.

6) **Clase Localidad**

```
package logica;
import java.util.ArrayList;
import java.util.List;
public class Localidad {
  private String nombre;
  private boolean numerada;
  private double precioBase;
  private int capacidad;
  private Evento evento;
  private List<Tiquete> tiquetes;
      public Localidad(String nombre, boolean numerada, double precioBase, int capacidad, Evento evento) {
    this.nombre = nombre;
    this.numerada = numerada;
    this.precioBase = precioBase;
    this.capacidad = capacidad;
    this.evento = evento;
    this.tiquetes = new ArrayList<>();
  }
  public String getNombre() {
    return nombre;
  }
  public boolean isNumerada() {
```

```
    return numerada;
  }
  public double getPrecioBase() {
    return precioBase;
  }
  public void setPrecioBase(double precioBase) {
    this.precioBase = precioBase;
  }
  public int getCapacidad() {
    return capacidad;
  }
  public Evento getEvento() {
    return evento;
  }
  public List<Tiquete> getTiquetes() {
    return tiquetes;
  }
  public void generarNTiquetes(int n) {
    try {
      if (n <= 0) {
       throw new Exception("Cantidad inválida de tiquetes");
      }
      if (tiquetes.size() + n > capacidad) {
       throw new Exception("Capacidad excedida");
      }
      for (int i = 0; i < n; i++) {
        Tiquete t;
        if (numerada) {
          t = new TiqueteNumerado(this, precioBase, i + 1);
        }
        else {
          t = new TiqueteSimple(this, precioBase);
        }
        tiquetes.add(t);
      }
    }
    catch (Exception e) {
      System.out.println("Error al generar tiquetes: " + e.getMessage());
    }
  }
  public int capacidadRestante() {
    return capacidad - tiquetes.size();
  }
}
```

Funcionalidad:

La clase Localidad representa una sección o zona dentro de un evento. Cada localidad tiene un nombre, una capacidad máxima, un precio base y puede ser numerada o no numerada (según si los asientos están asignados). Además, se encarga de generar los tiquetes asociados a esa sección y de controlar la capacidad disponible.

7) **Clase Venue**

```
package logica;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class Venue {
  private String nombre;
  private String ubicacion;
  private int capacidadMaxima;
  private boolean aprobado;
  private List<Evento> eventosProgramados = new ArrayList<>();
  public Venue(String nombre, String ubicacion, int capacidadMaxima) {
     this.nombre = nombre;
     this.ubicacion = ubicacion;
     this.capacidadMaxima = capacidadMaxima;
     this.aprobado = false;
  }
  public String getNombre() {
     return nombre;
  }
  public String getUbicacion() {
     return ubicacion;
  }
  public int getCapacidadMaxima() {
     return capacidadMaxima;
  }
  public boolean estaAprobado() {
     return aprobado;
  }
  public void setAprobado(boolean aprobado) {
     this.aprobado = aprobado;
  }

  public void agregarEvento(Evento evento) {
     if (evento != null && !eventosProgramados.contains(evento)) {
        eventosProgramados.add(evento);
```

```
    }
  }
  public boolean estaDisponible(LocalDateTime fecha) {
    for (Evento e : eventosProgramados) {
      if (e.getFechaHora().toLocalDate().equals(fecha.toLocalDate())) {
        return false;
      }
    }
    return true;
  }
}
```

Funcionalidad:

La clase Venue representa el lugar físico donde se realizan los eventos. Guarda información básica como el nombre, la ubicación, la capacidad máxima y si el lugar ha sido aprobado por un administrador para ser usado. También incluye un método para verificar la disponibilidad del lugar en una fecha determinada

8) Clase Tiquete

```
package logica;
public abstract class Tiquete {
  private static int contador = 0;
  protected int id;
  protected double precio;
  protected boolean disponible;
  protected boolean transferible;
  protected Cliente dueno;
  protected Localidad localidad;
  public Tiquete(Localidad localidad, double precio) {
    this.id = ++contador;
    this.localidad = localidad;
    this.precio = precio;
    this.disponible = true;
    this.transferible = true;
    this.dueno = null;
  }
  public int getId() {
    return id;
  }
  public double getPrecio() {
    return precio;
  }
  public boolean isDisponible() {
```

```
    return disponible;
  }
  public void setDisponible(boolean disponible) {
    this.disponible = disponible;
  }
  public boolean isTransferible() {
    return transferible;
  }
  public void setTransferible(boolean transferible) {
    this.transferible = transferible;
  }
  public Cliente getDueno() {
    return dueno;
  }
  public void setDueno(Cliente dueno) {
    this.dueno = dueno;
  }
  public Localidad getLocalidad() {
    return localidad;
  }
  public Evento getEvento() {
    return localidad != null ? localidad.getEvento() : null;
  }



  public double calcularPrecioFinal() {
    double precioFinal = precio + Administrador.getCargoFijo();
    Evento evento = getEvento();
    if (evento != null) {
                                                    Double      cargoTipo      =
Sistema.getAdministrador().getCargoPorTipo(evento.getTipo());
        precioFinal += (cargoTipo / 100.0) * precio;
    }
    return precioFinal;
  }
  public boolean marcarComoVendido(Cliente c) {
    if (disponible) {
      disponible = false;
      dueno = c;
      return true;
    }
    return false;
  }
```

```
  public boolean isVencido() {
    Evento e = getEvento();
    if (e == null) return false;
    return e.getFechaHora().isBefore(java.time.LocalDateTime.now());
  }
}
```

Funcionalidad:

Tiquete representa una entrada individual a un evento. Cada tiquete pertenece a una localidad, tiene un precio, puede estar disponible o vendido, puede o no ser transferible, y tiene un dueño (cliente) cuando es vendido. En base a esta se implementan Tiquete simple, numerado, deluxe y múltiple

9) **Clase Sistema**

```
package logica;
import java.util.ArrayList;
import java.util.List;
public class Sistema {
  private static List<Usuario> usuarios = new ArrayList<>();
  private static List<Evento> eventos = new ArrayList<>();
  private static List<Venue> venues = new ArrayList<>();
  private static Administrador administrador;
  public static void setAdministrador(Administrador admin) {
    administrador = admin;
  }
  public static Administrador getAdministrador() {
    return administrador;
  }
  public static void registrarUsuario(Usuario u) {
    if (u != null && !usuarios.contains(u)) {
      usuarios.add(u);
    }
  }
  public static void registrarEvento(Evento e) {
    if (e != null && !eventos.contains(e)) {
      eventos.add(e);
    }
  }
  public static void registrarVenue(Venue v) {
    if (v != null && !venues.contains(v)) {
      venues.add(v);
```

```
        }
    }
    public static List<Usuario> getUsuarios() {
        return usuarios;
    }
    public static List<Evento> getEventos() {
        return eventos;
    }
    public static List<Venue> getVenues() {
        return venues;
    }
}
```

Funcionalidad:

La clase Sistema funciona como el núcleo del programa: es la encargada de almacenar y gestionar todos los datos importantes de la aplicación, como los usuarios, los eventos, y los venues.

10) Clase AdminLog

```
package log;
import java.io.IOException;
import java.util.List;
import logica.Administrador;
import logica.Usuario;
public class AdminLog {
    private final IPersistenciaLog repo;
    public AdminLog(IPersistenciaLog repo) {
        this.repo = repo;
    }
    public void log(String actorId, LogEntrada.Tipo tipo, String detalle) {
        try {
            repo.append(new LogEntrada(actorId, tipo, detalle));
        } catch (IOException e) {
            throw new RuntimeException("Error escribiendo el log", e);
        }
    }
    public List<LogEntrada> listar(Usuario solicitante) {
        if (!(solicitante instanceof Administrador)) {
            throw new SecurityException("Solo el administrador puede consultar el log.");
        }
        try {
            return repo.leerTodo();
        } catch (IOException e) {
            throw new RuntimeException("Error leyendo el log", e);
        }
    }
}
```

Funcionalidad:

La clase AdminLog gestiona el riesgo de acciones del sistema, permitiendo guardar y consultar eventos importantes. Solo el administrador puede acceder al historial completo garantizando seguridad y control sobre las operaciones registradas

11) Clase LogEntrada

```
package log;
import java.time.Instant;
import java.util.UUID;
public final class LogEntrada {
		public enum Tipo {
	OFERTA_PUBLICADA,
	OFERTA_CANCELADA,
	OFERTA_REMOVIDA_ADMIN,
	CONTRAOFERTA_PROPUESTA,
	CONTRAOFERTA_ACEPTADA,
	CONTRAOFERTA_RECHAZADA,
	COMPRA_DIRECTA,
	VENTA_COMPLETADA
  }
		private final String id;
  private final String actorId;
  private final Tipo tipo;
  private final String detalle;
  private final String tiempo;

  public LogEntrada(String actorId, Tipo tipo, String detalle) {
    this.id = UUID.randomUUID().toString();
    this.actorId = actorId;
    this.tipo = tipo;
    this.detalle = detalle;
    this.tiempo = Instant.now().toString();
  }
  public LogEntrada(String id, String actorId, Tipo tipo, String detalle, String tiempo) {
    this.id = id;
    this.actorId = actorId;
    this.tipo = tipo;
    this.detalle = detalle;
    this.tiempo = tiempo;
  }

		public String getId() {
			return id;
		}
		public String getActorId() {
			return actorId;
		}
		public Tipo getTipo() {
			return tipo;
		}
```

```
        public String getDetalle() {
                return detalle;
        }
        public String getTiempo() {
                return tiempo;
        }

  @Override
  public String toString() {
    return tiempo + " | " + tipo + " | actor=" + actorId + " | " + detalle;
  }
}
```

Funcionalidad:

La clase LogEntrada representa un registro individual dentro del historial de actividades del sistema, cada entrada guarda informacion sobre quien realizó una acción (actorId), el tipo de evento (tipo), una descripción (detalle) y la hora en que ocurrio (tiempo); esto con el objetivo de asegurar trazabilidad y auditoría de las operaciones del marketplace, como las publicaciones, compras o cancelaciones

12) Clase Contraoferta

```
package marketplace;
import java.time.LocalDateTime;
import java.util.UUID;
public class Contraoferta {
  public enum Estado { PENDIENTE, ACEPTADA, RECHAZADA }
  private String id;
  private String idOfertaOriginal;
  private String idComprador;
  private double precioPropuesto;
  private Estado estado = Estado.PENDIENTE;
  private LocalDateTime fechaPropuesta = LocalDateTime.now();
  private LocalDateTime fechaResolucion;
  public Contraoferta(String idOferta, String comprador, double precio) {
    this.id = UUID.randomUUID().toString();
    this.idOfertaOriginal = idOferta;
    this.idComprador = comprador;
    this.precioPropuesto = precio;
  }
  public void aceptar() { this.estado = Estado.ACEPTADA; this.fechaResolucion = LocalDateTime.now(); }
        public void rechazar() { this.estado = Estado.RECHAZADA; this.fechaResolucion =
LocalDateTime.now(); }
  public String getId() { return id; }
  public String getIdOfertaOriginal() { return idOfertaOriginal; }
  public String getIdComprador() { return idComprador; }
  public double getPrecioPropuesto() { return precioPropuesto; }
  public Estado getEstado() { return estado; }
  public LocalDateTime getFechaPropuesta() { return fechaPropuesta; }
  public LocalDateTime getFechaResolucion() { return fechaResolucion; }
```

```
}
```

Funcionalidad:

La clase Contraoferta representa una propuesta alternativa de compra hecha por un cliente sobre una oferta existente en el marketplace, esta contiene la información del comprador, el precio propuesto y el estado de la negociación (pendiente, aceptada o rechazada), además registra las fechas de creación y resolución, permitiendo seguir de cerca los ciclos de vida de las contraofertas dentro del marketplace

13) Clase ControlMarketplace

```
package marketplace;
import java.util.List;
import java.util.stream.Collectors;
import log.AdminLog;
import log.LogEntrada;
import logica.Administrador;
import logica.Cliente;
import logica.Sistema;
import logica.Tiquete;
public class ControlMarketplace {
  private final IPersistenciaMarketplace repo;
  private final AdminLog log;
  public ControlMarketplace(IPersistenciaMarketplace repo, AdminLog log) {
    this.repo = repo;
    this.log = log;
  }
  public Oferta publicarOferta(Cliente vendedor, List<Tiquete> tiqs, double precio) {
    if (precio <= 0) throw new IllegalArgumentException("Precio inválido");
    if (tiqs == null || tiqs.isEmpty()) throw new IllegalArgumentException("No hay tiquetes");
    for (Tiquete t : tiqs) {
     }
    List<Integer> ids = tiqs.stream().map(Tiquete::getId).collect(Collectors.toList());
    Oferta o = Oferta.nueva(vendedor.getCorreo(), ids, precio);
    repo.guardarOferta(o);
    log.log(vendedor.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
        "oferta=" + o.getId() + " tiquetes=" + ids + " precio=" + precio);
    return o;
  }
  public void cancelarOferta(Cliente vendedor, String idOferta) {
    Oferta o = repo.buscarOferta(idOferta);
    if (o == null) throw new IllegalArgumentException("Oferta no existe");
    if (!o.getIdVendedor().equals(vendedor.getCorreo()))
      throw new SecurityException("No puedes cancelar una oferta de otro usuario");
    o.cancelar();
    repo.guardarOferta(o);
    log.log(vendedor.getCorreo(), LogEntrada.Tipo.OFERTA_CANCELADA, "oferta=" + idOferta);
  }
  public void removerOfertaComoAdmin(Administrador admin, String idOferta) {
```

```
    Oferta o = repo.buscarOferta(idOferta);
    if (o == null) throw new IllegalArgumentException("Oferta no existe");
    o.removerPorAdmin();
    repo.guardarOferta(o);
    log.log(admin.getCorreo(), LogEntrada.Tipo.OFERTA_REMOVIDA_ADMIN, "oferta=" + idOferta);
}
public List<Oferta> listarPublicadas() {
    return repo.listarPublicadas();
}
public void comprarOferta(Cliente comprador, String idOferta) {
    Oferta o = repo.buscarOferta(idOferta);
    if (o == null) throw new IllegalArgumentException("La oferta no existe");
    if (o.getEstado() != Oferta.Estado.PUBLICADA)
        throw new IllegalStateException("La oferta no está disponible para compra");
    o.marcarVendidaA(comprador.getCorreo());
    repo.guardarOferta(o);
    log.log(comprador.getCorreo(), LogEntrada.Tipo.COMPRA_DIRECTA,
            "compra directa oferta=" + o.getId() + " vendedor=" + o.getIdVendedor() +
            " precio=" + o.getPrecioPublicado());
    log.log(o.getIdVendedor(), LogEntrada.Tipo.VENTA_COMPLETADA,
            "venta completada oferta=" + o.getId() + " comprador=" + comprador.getCorreo());
}
public Contraoferta proponerContraoferta(Cliente comprador, String idOferta, double nuevoPrecio) {
    Oferta o = repo.buscarOferta(idOferta);
    if (o == null) throw new IllegalArgumentException("La oferta no existe");
    if (o.getEstado() != Oferta.Estado.PUBLICADA)
        throw new IllegalStateException("No puedes hacer contraoferta: la oferta no está activa");
    if (nuevoPrecio <= 0 || nuevoPrecio >= o.getPrecioPublicado())
        throw new IllegalArgumentException("El precio propuesto debe ser menor al publicado");
    Contraoferta cf = new Contraoferta(idOferta, comprador.getCorreo(), nuevoPrecio);
    log.log(comprador.getCorreo(), LogEntrada.Tipo.CONTRAOFERTA_PROPUESTA,
            "contraoferta=" + cf.getId() + " oferta=" + idOferta + " nuevoPrecio=" + nuevoPrecio);
    return cf;
}
public void aceptarContraoferta(Cliente vendedor, Contraoferta cf) {
    Oferta o = repo.buscarOferta(cf.getIdOfertaOriginal());
    if (o == null) throw new IllegalArgumentException("Oferta original no encontrada");
    if (!o.getIdVendedor().equals(vendedor.getCorreo()))
        throw new SecurityException("Solo el vendedor puede aceptar");
    if (cf.getEstado() != Contraoferta.Estado.PENDIENTE)
        throw new IllegalStateException("La contraoferta ya fue procesada");
    cf.aceptar();
    o.marcarVendidaA(cf.getIdComprador());
    repo.guardarOferta(o);
    log.log(vendedor.getCorreo(), LogEntrada.Tipo.CONTRAOFERTA_ACEPTADA,
            "oferta=" + o.getId() + " contraoferta=" + cf.getId() + " precio=" + cf.getPrecioPropuesto());
    log.log(vendedor.getCorreo(), LogEntrada.Tipo.VENTA_COMPLETADA,
            "venta completada por contraoferta a " + cf.getIdComprador());
}
public void rechazarContraoferta(Cliente vendedor, Contraoferta cf) {
    Oferta o = repo.buscarOferta(cf.getIdOfertaOriginal());
    if (o == null) throw new IllegalArgumentException("Oferta original no encontrada");
```

```
    if (!o.getIdVendedor().equals(vendedor.getCorreo()))
        throw new SecurityException("Solo el vendedor puede rechazar");
    if (cf.getEstado() != Contraoferta.Estado.PENDIENTE)
        throw new IllegalStateException("La contraoferta ya fue procesada");
    cf.rechazar();
    log.log(vendedor.getCorreo(), LogEntrada.Tipo.CONTRAOFERTA_RECHAZADA,
            "oferta=" + o.getId() + " contraoferta=" + cf.getId() + " comprador=" + cf.getIdComprador());
  }
  public Oferta buscarOferta(String idOferta) {
    var o = repo.buscarOferta(idOferta);
    if (o == null) throw new IllegalArgumentException("La oferta no existe");
    return o;
  }
}
```

Funcionalidad:

La clase ControlMarketplace coordina todas las operaciones del sistema de compraventa de tiquetes, permite publicar, comprar y cancelar ofertas, además de gestionar contraofertas entre usuarios. Se apoya de la capa IPersistenciaMarketplace para almacenar datos y con AdminLog para registrar cada acción, con el objetivo de mantener la lógica de negocio central y garantiza la trazabilidad de las transacciones

14) Clase Oferta

```
package marketplace;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
public class Oferta {
  public enum Estado { PUBLICADA, CANCELADA, REMOVIDA_ADMIN, VENDIDA }
  private String id;
  private String idVendedor;
  private List<Integer> idsTiquetes = new ArrayList<>();
  private double precioPublicado;
  private Estado estado = Estado.PUBLICADA;
  private LocalDateTime fechaCreacion = LocalDateTime.now();
  private LocalDateTime fechaCierre;
  private String idCompradorFinal;
  public static Oferta nueva(String idVendedor, List<Integer> idsTiquetes, double precio) {
    Oferta o = new Oferta();
    o.id = UUID.randomUUID().toString();
    o.idVendedor = idVendedor;
    o.idsTiquetes.addAll(idsTiquetes);
    o.precioPublicado = precio;
    return o;
  }
  public void cancelar() { this.estado = Estado.CANCELADA; this.fechaCierre = LocalDateTime.now(); }
    public void removerPorAdmin() { this.estado = Estado.REMOVIDA_ADMIN; this.fechaCierre =
LocalDateTime.now(); }
```

```
  public void marcarVendidaA(String idComprador) {
     this.estado = Estado.VENDIDA;
     this.idCompradorFinal = idComprador;
     this.fechaCierre = LocalDateTime.now();
  }
  public String getId() {
          return id;
          }
  public String getIdVendedor() {
          return idVendedor;
          }
  public List<Integer> getIdsTiquetes() {
          return idsTiquetes;
          }
  public double getPrecioPublicado() {
          return precioPublicado;
          }
  public Estado getEstado() {
          return estado;
          }
  public LocalDateTime getFechaCreacion() {
          return fechaCreacion;
          }
  public LocalDateTime getFechaCierre() {
          return fechaCierre;
          }
  public String getIdCompradorFinal() {
          return idCompradorFinal;
          }
  public void setPrecioPublicado(double p) { this.precioPublicado = p; }
}
```

Funcionalidad:

La clase Oferta representa una publicacion de venta de tiquetes dentro del marketplace, contiene los datos del vendedor, los tiquetes ofrecidos, el precio y el estado de la oferta (publicada, vendida, removida, cancelada), también registra las fechas y permite actualizar el estado según las acciones

15) Clase AdminLogTest

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.List;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import log.*;
import logica.Administrador;
import logica.Cliente;
import java.nio.file.*;
```

```java
public class AdminLogTest {
  private AdminLog log;
  private Path filePath;
  @BeforeEach
  void setUp() throws Exception {
     filePath = Paths.get("data/test_audit.jsonl");
     Files.deleteIfExists(filePath);
     log = new AdminLog(new PersistenciaLogJson(filePath.toString()));
  }
  @AfterEach
  void tearDown() throws Exception {
     Files.deleteIfExists(filePath);
  }
  @Test
  void testLogYLecturaPorAdmin() {
     log.log("cliente@test.com", LogEntrada.Tipo.OFERTA_PUBLICADA, "detalle=prueba");
     log.log("cliente@test.com", LogEntrada.Tipo.OFERTA_CANCELADA, "detalle=cancelado");
     Administrador admin = new Administrador("admin@test.com", "1234");
     List<LogEntrada> entradas = log.listar(admin);
     assertEquals(2, entradas.size(),
        "La cantidad de entradas en el log no es la esperada.");
     assertEquals(LogEntrada.Tipo.OFERTA_PUBLICADA, entradas.get(0).getTipo(),
        "El tipo de la primera entrada no coincide con OFERTA_PUBLICADA.");
  }
  @Test
  void testSoloAdminPuedeConsultarLog() {
     log.log("cliente@test.com", LogEntrada.Tipo.OFERTA_PUBLICADA, "detalle=prueba");
     Cliente c = new Cliente("cliente@test.com", "1234");
     assertThrows(SecurityException.class, () -> log.listar(c),
        "Un cliente no debería poder consultar el log");
  }
}
```

Funcionalidad:

El test AdminLogTest valida el correcto funcionamiento del registro de auditoría del sistema, comprueba que las entradas del log se guardan y se lean de manera adecuada mediante la clase AdminLog, asegurando que solo los administradores tengan permiso para consultarlas. El primer test verifica que se registren correctamente dos eventos y se lean en orden. El segundo test confirma que un cliente común no pueda acceder al log, lanzando una excepción de seguridad

16) Clase ControlMarketplaceTest

```java
package test;
import static org.junit.jupiter.api.Assertions.*;
import java.nio.file.*;
import java.util.List;
import org.junit.jupiter.api.*;
import log.*;
```

```
import logica.*;
import marketplace.*;
public class ControlMarketplaceTest {
  private ControlMarketplace mkt;
  private AdminLog log;
  private Path ofertasFile;
  private Path logFile;
  private Cliente vendedor;
  private Cliente comprador;
  private Localidad locPrueba;
  @BeforeEach
  void setUp() throws Exception {
    ofertasFile = Paths.get("data/test_ofertas.json");
    logFile = Paths.get("data/test_audit.jsonl");
    Files.deleteIfExists(ofertasFile);
    Files.deleteIfExists(logFile);
    log = new AdminLog(new PersistenciaLogJson(logFile.toString()));
    mkt = new ControlMarketplace(new PersistenciaMarketplaceJson(ofertasFile.toString()), log);
    vendedor = new Cliente("vendedor@test.com", "1234");
    comprador = new Cliente("comprador@test.com", "abcd");
    locPrueba = new Localidad("General", false, 100.0, 50, null);
  }
  @AfterEach
  void tearDown() throws Exception {
    Files.deleteIfExists(ofertasFile);
    Files.deleteIfExists(logFile);
  }
  @Test
  void testPublicarYCancelarOferta() {
    TiqueteNumerado t1 = new TiqueteNumerado(locPrueba, 100.0, 1);
    Oferta o = mkt.publicarOferta(vendedor, List.of(t1), 100.0);
    assertEquals(Oferta.Estado.PUBLICADA, o.getEstado(),
        "La oferta debería iniciar en estado PUBLICADA.");
    mkt.cancelarOferta(vendedor, o.getId());
    Oferta cancelada = mkt.buscarOferta(o.getId());
    assertEquals(Oferta.Estado.CANCELADA, cancelada.getEstado(),
        "La oferta no cambió a estado CANCELADA.");
  }
  @Test
  void testCompraDirectaCambiaEstadoYGeneraLog() {
    TiqueteNumerado t2 = new TiqueteNumerado(locPrueba, 120.0, 2);
    Oferta o = mkt.publicarOferta(vendedor, List.of(t2), 120.0);
    mkt.comprarOferta(comprador, o.getId());
    Oferta vendida = mkt.buscarOferta(o.getId());
    assertEquals(Oferta.Estado.VENDIDA, vendida.getEstado(),
        "La oferta no cambió a estado VENDIDA tras la compra.");
    Administrador admin = new Administrador("admin@test.com", "0000");
    var entradas = log.listar(admin);
    assertTrue(entradas.stream().anyMatch(e -> e.getTipo() == LogEntrada.Tipo.COMPRA_DIRECTA),
        "No se registró la compra directa en el log.");
                                    assertTrue(entradas.stream().anyMatch(e -> e.getTipo() ==
LogEntrada.Tipo.VENTA_COMPLETADA),
```

```
            "No se registró la venta completada en el log.");
    }
    @Test
    void testContraofertaPropuestaYAceptada() {
        TiqueteNumerado t3 = new TiqueteNumerado(locPrueba, 150.0, 3);
        Oferta o = mkt.publicarOferta(vendedor, List.of(t3), 150.0);
        Contraoferta cf = mkt.proponerContraoferta(comprador, o.getId(), 100.0);
        mkt.aceptarContraoferta(vendedor, cf);
        assertEquals(Contraoferta.Estado.ACEPTADA, cf.getEstado(),
                "La contraoferta no cambió a estado ACEPTADA.");
        assertEquals(Oferta.Estado.VENDIDA, mkt.buscarOferta(o.getId()).getEstado(),
                "La oferta original no cambió a VENDIDA tras aceptar la contraoferta.");
    }
}
```

Funcionalidad:

El test ControlMarketplaceTest valida el funcionamiento completo del marketplace, comprueba que las ofertas se publiquen, se vendan y se cancelen de forma correcta y que las acciones queden registradas en el log. Los test verifican que las ofertas se creen en estado PUBLICADA y pase a CANCELADA, que una compra cambia la oferta a VENDIDA y genera los registros en el log, y el último valida que una contraoferta pueda proponerse y ser aceptada, cambiando de estado

17) Clase OfertaTest

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.List;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import marketplace.Oferta;
public class OfertaTest {
    private Oferta oferta;
    @BeforeEach
    void setUp() throws Exception {
        oferta = Oferta.nueva("vendedor@test.com", List.of(1, 2, 3), 120.0);
    }
    @AfterEach
    void tearDown() throws Exception {
        oferta = null;
    }
    @Test
    void testCreacionOferta() {
        assertEquals("vendedor@test.com", oferta.getIdVendedor(),
            "El id del vendedor no es el esperado.");
        assertEquals(3, oferta.getIdsTiquetes().size(),
            "La cantidad de tiquetes no es la esperada.");
        assertEquals(120.0, oferta.getPrecioPublicado(),
            "El precio publicado no es el esperado.");
```

```
        assertEquals(Oferta.Estado.PUBLICADA, oferta.getEstado(),
            "El estado inicial de la oferta debe ser PUBLICADA.");
    }
    @Test
    void testCancelarOferta() {
        oferta.cancelar();
        assertEquals(Oferta.Estado.CANCELADA, oferta.getEstado(),
            "La oferta no cambió a estado CANCELADA.");
        assertNotNull(oferta.getFechaCierre(),
            "La fecha de cierre no fue registrada al cancelar la oferta.");
    }
    @Test
    void testMarcarVendidaA() {
        oferta.marcarVendidaA("comprador@test.com");
        assertEquals(Oferta.Estado.VENDIDA, oferta.getEstado(),
            "La oferta no cambió a estado VENDIDA.");
        assertEquals("comprador@test.com", oferta.getIdCompradorFinal(),
            "El comprador final no fue registrado correctamente.");
    }
}
```

Funcionalidad:

El test OfertaTest verifica el correcto comportamiento de la clase oferta. Los test verifican que una oferta se cree con los datos correctos (vendedor, precio, tiquetes, y su estado), verifican que la oferta se pueda cancelar y su estado cambie, y el ultimo valida que al completarse una venta, su estado cambie y el comprador final quede registrado

18) Clase Principal

```
package presentacion;
import java.time.LocalDateTime;
import logica.Administrador;
import logica.Cliente;
import logica.Evento;
import logica.Localidad;
import logica.Organizador;
import logica.Sistema;
import logica.Tiquete;
import logica.Venue;
import log.LogEntrada;
public class Principal {
    private Sistema sistema;
    public Principal() {
        sistema = new Sistema();
        caso1();
        caso2();
    }
    private void caso1() {
        System.out.println("\n=== CASO 1 ===");
```

```
    Administrador admin = new Administrador("martin@dpoo.com", "1234");
    Sistema.setAdministrador(admin);
    sistema.getAudit().log(admin.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Administrador creado y asignado: " + admin.getCorreo());
    Venue v = new Venue("Auditorio ML", "Carrera 1e", 500);
    Sistema.registrarVenue(v);
    admin.aprobarVenue(v);
    sistema.getAudit().log(admin.getCorreo(), LogEntrada.Tipo.VENTA_COMPLETADA,
            "Administrador aprobó venue: " + v.getNombre());
    Organizador org = new Organizador("isabella@dpoo.com", "abcd", "Organizador 1");
    Sistema.registrarUsuario(org);
    LocalDateTime fecha = LocalDateTime.now().plusDays(5);
    Evento e = org.crearEvento("Musical Shakespeare", fecha, "musical", v);
    Sistema.registrarEvento(e);
    sistema.getAudit().log(org.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Organizador creó evento: " + e.getNombre());
    Localidad loc = org.definirLocalidad("General", false, 100.0, 50, e);
    loc.generarNTiquetes(5);
    sistema.getAudit().log(org.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Localidad definida: " + loc.getNombre() + " con " + loc.getTiquetes().size() + " tiquetes");
    Cliente c1 = new Cliente("juanca@dpoo.com", "5678");
    c1.setSaldo(500.0);
    Sistema.registrarUsuario(c1);
    Cliente c2 = new Cliente("hector@dpoo.com", "efgh");
    c2.setSaldo(300.0);
    Sistema.registrarUsuario(c2);
    Tiquete t1 = loc.getTiquetes().get(0);
    c1.comprarTiquete(t1);
    sistema.getAudit().log(c1.getCorreo(), LogEntrada.Tipo.COMPRA_DIRECTA,
            "Cliente " + c1.getCorreo() + " compró tiquete " + t1.getId());
    c1.transferirTiquete(t1, c2, "pass1");
    sistema.getAudit().log(c1.getCorreo(), LogEntrada.Tipo.VENTA_COMPLETADA,
            "Cliente " + c1.getCorreo() + " transfirió tiquete " + t1.getId() + " a " + c2.getCorreo());
    admin.setCargoFijo(2.0);
    admin.fijarCargoPorTipo("musical", 10.0);
    sistema.getAudit().log(admin.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Administrador fijó cargos para tipo musical.");
    admin.cancelarEvento(e, true);
    sistema.getAudit().log(admin.getCorreo(), LogEntrada.Tipo.OFERTA_CANCELADA,
            "Administrador canceló evento: " + e.getNombre());
    System.out.println("Evento cancelado: " + e.getNombre());
    System.out.println("Estado del evento: " + e.getEstado());
    System.out.println("Saldo cliente 1: " + c1.getSaldo());
    System.out.println("Saldo cliente 2: " + c2.getSaldo());
}
private void caso2() {
    System.out.println("\n=== CASO 2 ===");
    Administrador admin = new Administrador("pepito@dpoo.com", "ijkl");
    Sistema.setAdministrador(admin);
    sistema.getAudit().log(admin.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Administrador creado: " + admin.getCorreo());
    admin.aprobarVenue(null);
```

```
    Venue v = new Venue("Auditorio Lleras", "Calle 19A", 1000);
    Sistema.registrarVenue(v);
    Organizador org = new Organizador("perensejo@dpoo.com", "0910", "Organizador 2");
    Sistema.registrarUsuario(org);
    Evento e = org.crearEvento("Concierto fallido", LocalDateTime.now().plusDays(1), "musical", v);
    sistema.getAudit().log(org.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Intento de crear evento fallido en venue no aprobado.");
    admin.aprobarVenue(v);
    e = org.crearEvento("Concierto correcto", LocalDateTime.now().plusDays(1), "musical", v);
    Sistema.registrarEvento(e);
    sistema.getAudit().log(org.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Evento aprobado: " + e.getNombre());
    Localidad l = org.definirLocalidad("VIP", true, 200.0, 5, e);
    l.generarNTiquetes(3);
    Cliente c = new Cliente("batman@dpoo.com", "mnop");
    Sistema.registrarUsuario(c);
    c.setSaldo(50);
    sistema.getAudit().log(c.getCorreo(), LogEntrada.Tipo.OFERTA_PUBLICADA,
            "Cliente creado con saldo inicial 50");
    Tiquete t = l.getTiquetes().get(0);
    c.comprarTiquete(t);
    sistema.getAudit().log(c.getCorreo(), LogEntrada.Tipo.COMPRA_DIRECTA,
            "Cliente " + c.getCorreo() + " intentó comprar con saldo insuficiente.");
    c.setSaldo(500);
    c.comprarTiquete(t);
    sistema.getAudit().log(c.getCorreo(), LogEntrada.Tipo.COMPRA_DIRECTA,
            "Cliente " + c.getCorreo() + " compró tiquete " + t.getId());
    Cliente c2 = new Cliente("robin@dpoo.com", "1112");
    Sistema.registrarUsuario(c2);
    c.transferirTiquete(t, c2, "incorrecta");
    sistema.getAudit().log(c.getCorreo(), LogEntrada.Tipo.CONTRAOFERTA_RECHAZADA,
            "Cliente " + c.getCorreo() + " intentó transferir con clave incorrecta.");
    c.transferirTiquete(t, c2, "1112");
    sistema.getAudit().log(c.getCorreo(), LogEntrada.Tipo.VENTA_COMPLETADA,
            "Cliente " + c.getCorreo() + " transfirió tiquete a " + c2.getCorreo());
  }

  public static void main(String[] args) {
    new Principal();
  }
}
```

Funcionalidad:

La clase Principal actua como punto de entrada del sistema ejecutando casos de demostracion para validar el funcionamiento general del negocio. En el caso 1 se simula un evento exitoso, donde un administrador aprueba el venue, un organizador crea un evento y define localidades, un cliente compra y transfiere tiquetes y el administrador cancela el evento con reembolsos. En el caso 2 se prueban situaciones

con errores: eventos sin aprobación, compras con saldo insuficiente, transferencias con contraseña incorrecta

Historias de usuario

- **Administrador:**
    a. Aprobar promotores registrados

    Como administrador, quiero aprobar a los nuevos promotores registrados para que puedan crear y gestionar su evento.

    Entrada: lista de promotores pendientes, selección de promotores a aprobar

    Salida: promotores obtienen acceso completo al módulo de organización

    Resultado: los promotores obtienen acceso completo al módulo de organización.

    b. Consultar log de auditoría del sistema

    Como administrador, quiero consultar el log de auditoría para conocer todas las acciones realizadas por los usuarios, tales como publicaciones, compras y cancelaciones.

    Entrada: autenticación del administrador

    Salida: listado de registros con fecha, hora, tipo de acción, actor de la acción y detalles de la acción

    Resultado: el administrador puede revisar y auditar la actividad del sistema

    c. Eliminar oferta publicada

    Como administrador, quiero eliminar una oferta del marketplace para modelar publicaciones inapropiadas o incorrectas.

    Entrada: ID de la oferta

    Salida: mensaje “Oferta removida por el administrador”

    Resultado: cambio de estado de la oferta a REMOVIDA_ADMIN y se registra en el log

d. Fijar cargos por tipo de evento

   Como administrador, quiero definir cargos o comisiones según el tipo de evento para controlar la rentabilidad del negocio

   Entrada: tipo de evento, valor o porcentaje del cargo

   Salida: mensaje de confirmación

   Resultado: el cargo queda guardado y se aplica a nuevos eventos de ese tipo

e. Consultar listado de usuarios y estadísticas

   Como administrador, quiero ver la lista de todos los usuarios registrados y sus roles para tener control sobre el sistema

   Entrada: selección “Listar usuarios” desde el menú

   Salida: tabla con nombre, correo, rol y estado (aprobado/pendiente)

   Resultado: el administrador puede verificar la actividad general del sistema

- **Promotor/Organizador:**
  a. Registro como promotor

     Como promotor, quiero registrarme en el sistema para poder crear y vender eventos

     Entrada: nombre, correo, contraseña, nombre de la empresa

     Salida: mensaje de confirmación “Registro pendiente de aprobación”

     Resultado: el administrador debe aprobar al promotor antes de que pueda crear eventos

  b. Crear un nuevo evento

     Como organizador, quiero crear un evento

     Entrada: nombre del evento, tipo del evento, fecha y venue

Salida: mensaje “Evento creado exitosamente”

Resultado: el evento queda guardado y disponible para definir localidades

c. Definir localidades y generar tiquetes

Como organizador, quiero definir localidades con su precio y la cantidad de tiquetes por cada una

Entrada: nombre de la localidad, cantidad de tiquetes, precio de tiquetes, tipo de tiquete (numerada/no numerada)

Salida: mensaje de confirmación “Tiquetes generados correctamente”

Resultado: los tiquetes quedan disponibles para la venta

d. Cancelar un evento

Como organizador, quiero cancelar un evento

Entrada: ID del evento a cancelar

Salida: mensaje “Evento cancelado con éxito”

Resultado: cambio de estado del evento a CANCELADO y se registra en el log

e. Consultar eventos creados

Como organizador, quiero ver la lista de mis eventos y su estado para gestionar mis ventas

Entrada: menú “Ver mis eventos”

Salida: tabla con nombre, fecha, estado, total de tiquetes

Resultado: el promotor puede consultar y modificar sus eventos activos

- **Cliente:**
  a. Registro como cliente

Como cliente, quiero registrarme para poder comprar y revender tiquetes

Entradas: correo, contraseña, nombre completo

Salida: mensaje “Cuenta creada exitosamente”

Resultado: el cliente queda registrado y puede autenticarse en el sistema

b. Comprar tiquetes de eventos

Como cliente, quiero comprar tiquetes de un evento publicado

Entrada: ID del evento, localidad, cantidad de tiquetes, saldo

Salida: mensaje “Compra realizada correctamente”

Resultado: el sistema descuenta el saldo, asigna los tiquetes y registra la compra en el log

c. Publicar oferta en el marketplace

Como cliente, quiero publicar una oferta de reventa de mis tiquetes

Entrada: ID del tiquete, precio de reventa

Salida: mensaje “Oferta publicada exitosamente”

Resultado: la oferta se guarda con estado PUBLICADA y se agrega al archivo de ofertas

d. Cancelar una oferta

Como cliente, quiero cancelar una oferta publicada en el marketplace

Entrada: ID de la oferta

Salida: mensaje “Oferta cancelada”

Resultado: cambio de estado de la oferta a CANCELADA y se registra en el log

e. Comprar oferta de reventa

   Como cliente, quiero comprar una oferta del marketplace

   Entrada: ID de la oferta, saldo

   Salida: mensaje “Compra completada exitosamente”

   Resultado: el comprador obtiene los tiquetes, el vendedor recibe la confirmación, ambos quedan registrados en el log

f. Hacer una contraoferta

   Como cliente, quiero hacer una contraoferta sobre una oferta del marketplace

   Entrada: ID de oferta, nuevo precio propuesto

   Salida: mensaje “Contraoferta enviada al vendedor”

   Resultado: la contraoferta queda registrada y el vendedor puede aceptarla o no

g. Aceptar o rechazar contraoferta

   Como cliente y vendedor de un tiquete en el marketplace, quiero aceptar o rechazar una contraoferta

   Entrada: ID de la contraoferta, acción (aceptar/rechazar)

   Salida: mensaje “Contraoferta aceptada/rechazada correctamente”

h. Transferir tiquetes

   Como cliente, quiero transferir un tiquete a otro usuario para ceder mi entrada

   Entrada: ID del tiquete, correo del destinatario, contraseña

   Salida: mensaje “Tiquete transferido con éxito”

   Resultado: el tiquete cambia de propietario y se registra el cambio en el log

Consola

Se desarrollaron 3 interfaces de consola en el proyecto, una por cada tipo de usuario (administrador, organizador, cliente), cada una cuenta con un método main independiente que permite iniciar sesión y acceder a su respectivo menú con opciones correspondientes a su rol. En la consola del administrador se implementaron las opciones para ver el log del sistema y eliminar ofertas del marketplace. Mientras que en la consola del organizador se implementaron funciones para crear eventos, definir localidades y cancelar eventos. Por último en la consola del cliente se implementaron las opciones de comprar tiquetes, publicar y cancelar ofertas, comprar tiquetes en el marketplace y hacer, aceptar o rechazar contraofertas. Las 3 consolas cargan datos de demostración al iniciar para seguir el funcionamiento de la lógica del negocio y la persistencia del mismo.