Proyecto 2 – Entrega : Ajustes con nuevas implementaciones de marketplace y persistencia.

Facultad de Ingeniería, Universidad de los Andes

ISIS 1226: Diseño y programación orientada a objetos

Hector Florez

3 de Noviembre 2025

Sistema: Boletamaster

Integrantes:

- Tomás Fernando Albarracín Malaver - 202421942
- Juan Esteban García Bonillas - 202420481
- Juan Esteban Angel Perdigon - 202420145

Introducción

En esta segunda entrega del proyecto Boletamaster, se desarrolló la capa de interacción por consola para los distintos tipos de usuarios (Administrador, Organizador y Cliente), se implementó la persistencia de datos mediante archivos JSON, y se consolidó la estructura modular del sistema con principios de bajo acoplamiento y alta cohesión.

El sistema permite crear usuarios, eventos, venues y gestionar tiquetes simples dentro de un flujo que simula un marketplace. Se incluyó un esquema de persistencia automático que guarda el estado de la base de datos y del marketplace en archivos dentro del directorio data/, garantizando que toda la información pueda cargarse nuevamente al reiniciar la aplicación.

Además, se incorporó un conjunto de pruebas unitarias con JUnit 5 para validar las operaciones principales del sistema, como transferencias, recargas, reembolsos y persistencia.

---

1. Estado de la Aplicación

1.1 Nuevo paquete marketplace

Se adiciona el submódulo Marketplace, encargado de gestionar la compraventa de tiquetes entre usuarios utilizando el SaldoVirtual como único medio de transacción.

Clases nuevas:

- Marketplace
  +ofertas: List<Oferta>
  +log: List<LogRegistro>
  +publicarOferta(vendedor: Usuario, t: Tiquete, precio: double): Oferta
  +ofertar(ofertaId: String, c: Contraoferta): boolean
  +aceptarContraoferta(ofertaId: String, c: Contraoferta, vendedor: Usuario): boolean
  +eliminarOferta(ofertaId: String, vendedor: Usuario): boolean
  +registrarAccion(usuario: String, accion: String, detalle: String): void
  +getOfertas(): List<Oferta>
  +getLog(): List<LogRegistro>

- Oferta
  +id: String
  +vendedor: Usuario
  +tiquete: Tiquete
  +precioBase: double
  +activa: boolean
  +contraofertas: List<Contraoferta>
  +agregarContraoferta(c: Contraoferta): void
  +aceptarContraoferta(c: Contraoferta): void
  +isActiva(): boolean

- Contraoferta
  +comprador: Usuario
  +precioPropuesto: double
  +aceptada: boolean
  +aceptar(): void
- LogRegistro
  +fechaHora: LocalDateTime
  +usuario: String
  +accion: String
  +detalle: String
  +toString(): String

1.2 Ajustes menores al dominio existente

Las clases Usuario, Tiquete, Evento, Localidad y sus derivadas mantienen sus atributos y operaciones, pero ahora pueden participar en flujos del Marketplace. Se conservan las reglas de dominio ya implementadas: uso obligatorio del SaldoVirtual, cashout por número de cuenta verificada, precio base uniforme por localidad, unicidad venue-fecha y aprobación de venues por el administrador.

---

2. Objetivos

Objetivo general: Implementar la capa de persistencia y las consolas interactivas del sistema Boletamaster, asegurando la correcta serialización de datos, el funcionamiento modular y la ejecución de pruebas automatizadas.

Objetivos específicos:

- Diseñar una estructura modular que separe la lógica de negocio, la persistencia y las interfaces de usuario.
- Implementar persistencia JSON mediante las clases JsonStore y MarketplaceStore.
- Desarrollar tres consolas independientes (AdminConsola, OrganizadorConsola, ClienteConsola) con menús funcionales y flujo de interacción.
- Permitir la creación, consulta y manipulación de eventos, usuarios y tiquetes desde la consola.
- Implementar pruebas unitarias con JUnit 5 para garantizar la estabilidad del sistema.
- Mantener bajo acoplamiento y alta cohesión siguiendo los principios de diseño orientado a objetos.

---

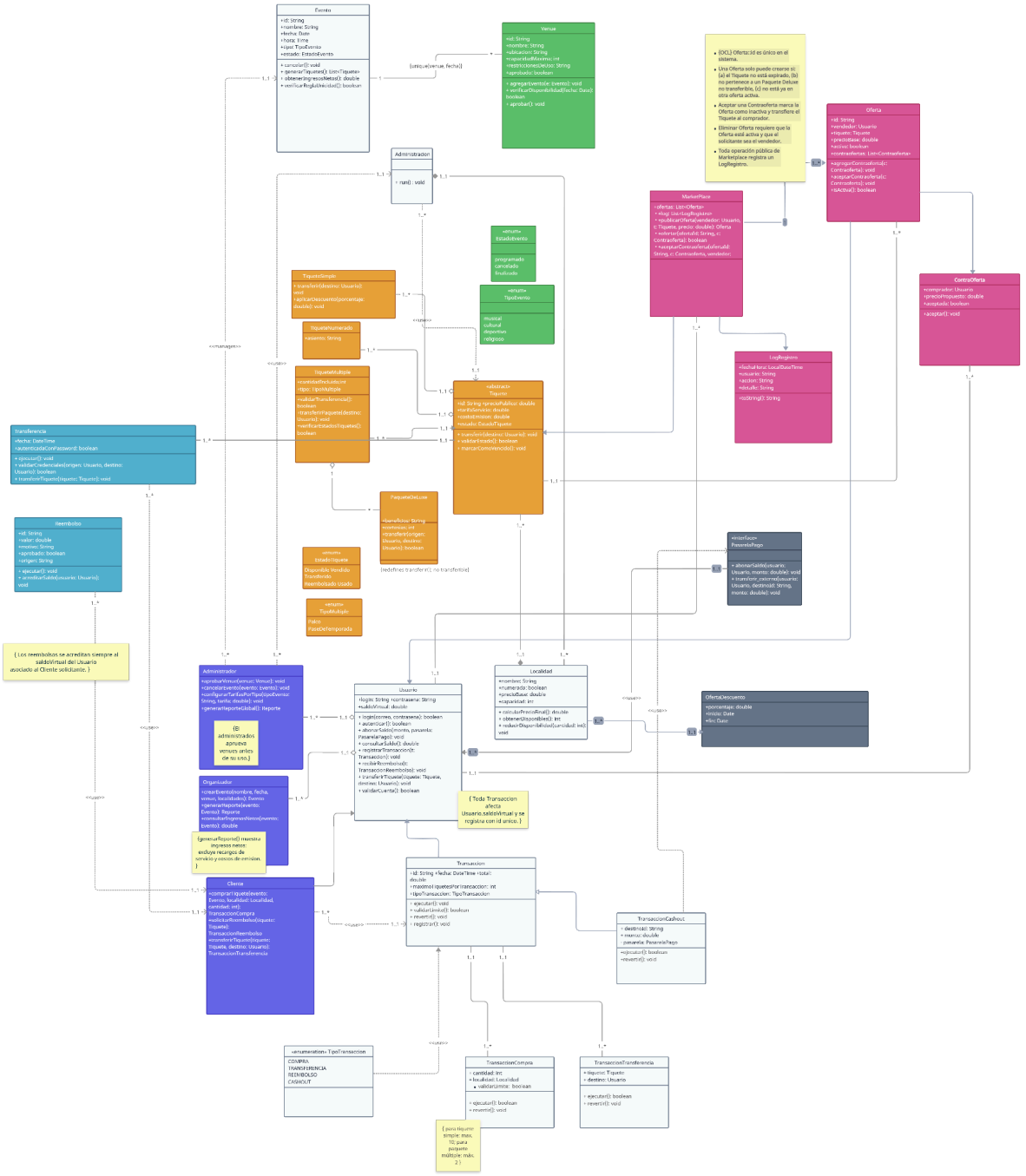
3. Reglas de Dominio (actualizadas)

1. Todo usuario debe autenticarse con login y contraseña válidos antes de acceder al sistema.
2. Cada usuario posee un saldo virtual que se usa como medio exclusivo de pago dentro de la plataforma.
3. Los administradores pueden crear y listar usuarios, además de realizar guardados manuales de la base de datos.
4. Los organizadores pueden crear, listar y aprobar venues, así como registrar eventos asociados a venues aprobados.
5. Los clientes pueden listar eventos disponibles, seleccionar una localidad y comprar tiquetes simples siempre que tengan saldo suficiente.
6. Cada evento debe estar asociado a un venue existente y aprobado antes de su publicación.
7. El sistema guarda automáticamente los datos del marketplace y la base de datos cada vez que se ejecuta una acción de guardado o cierre de consola.
8. La persistencia se realiza a través de archivos JSON que almacenan usuarios, venues, eventos y marketplace.
9. Ningún usuario puede realizar transacciones con saldo negativo.
10. Las clases mantienen relaciones unidireccionales para evitar dependencias circulares.

11. Cada consola usa un flujo independiente que hereda de la clase abstracta MainConsola, la cual controla el ciclo principal de ejecución (boot → login → menú → shutdown).
12. La creación y manipulación de entidades está restringida al rol correspondiente (por ejemplo, solo el organizador puede crear eventos).
13. El sistema permite la reutilización de objetos Usuario, Evento y Venue entre sesiones mediante persistencia.

---

4. UML actualizado



5. Implementación

El sistema se estructuró en capas:

- Capa de negocio: maneja la lógica de usuarios, eventos, tiquetes y marketplace.
- Capa de persistencia: utiliza JsonStore para serializar los objetos del sistema y guardarlos en data/.
- Capa de presentación: implementada mediante las consolas interactivas que usan entrada estándar (Scanner) para leer comandos.
- Capa de pruebas: desarrollada en JUnit 5 para verificar las operaciones principales.

Las consolas (AdminConsola, ClienteConsola, OrganizadorConsola) heredan de MainConsola e implementan menús independientes con métodos como renderMenu(), handleOption(), afterBoot() y shutdown(). El sistema se ejecuta desde el método main() de cada consola. Los archivos JSON generados permiten restaurar todo el estado de la base de datos y del marketplace tras cada ejecución.

---

6. Requerimientos (funcionales y no funcionales)

Requerimientos funcionales:

- Permitir publicar, eliminar y consultar ofertas en el marketplace.
- Registrar contraofertas y permitir su aceptación o rechazo por el vendedor.
- Guardar automáticamente los datos del marketplace en un archivo JSON tras cada acción.
- Cargar el estado del marketplace desde el archivo JSON al iniciar el programa.
- Registrar cada acción en un log con usuario, detalle y fecha/hora.
- Permitir al administrador consultar el historial completo de acciones.
- Validar las reglas de negocio previas: unicidad de eventos, saldo virtual, precio base uniforme y aprobación de venues.

Requerimientos no funcionales:

- Bajo acoplamiento entre módulos de dominio, marketplace y persistencia.
- Alta cohesion en las clases y métodos.
- Persistencia automática sin intervención del usuario.
- Validación de entradas en consola para evitar errores.
- Modularidad y legibilidad del codigo fuente.
- Tiempos de respuesta cortos al cargar y guardar datos.
- Pruebas unitarias y de integración con JUnit 5.

---

7. Pruebas unitarias

Se desarrollaron pruebas con JUnit 5 para validar las siguientes funcionalidades:

- Creación y autenticación de usuarios.
- Registro y aprobación de venues.
- Creación de eventos con fecha, hora y tipo.
- Compra de tiquetes simples y verificación de saldo.
- Persistencia de datos mediante JsonStore y MarketplaceStore.

Todas las pruebas fueron ejecutadas desde el paquete test, obteniendo resultados exitosos (sin fallos ni errores).

---

8. Historias de usuario

8.1 Administrador aprueba un venue antes de su uso

Actor: Administrador

Descripción: El administrador debe aprobar los venues creados por los promotores antes de que puedan usarse en un evento.

Entradas: Identificador del venue pendiente de aprobación.

Flujo principal:

- El sistema muestra la lista de venues en estado "no aprobados".
- El administrador selecciona uno y elige la opción "Aprobar venue".
- El sistema cambia el atributo aprobado a verdadero.

Salidas o resultados esperados: Mensaje "Venue aprobado exitosamente".

Reglas aplicadas: Un venue no puede alojar eventos si no esta aprobado (R-Administracion).

Criterio de aceptación: El venue aparece como aprobado y puede ser usado por el promotor al crear un evento.

8.2 Promotor crea un evento en un venue aprobado

Actor: Promotor

Descripción: El promotor registra un nuevo evento en un venue ya aprobado por el administrador.

Entradas: Nombre del evento, fecha, hora, venue, localidades y precios base.

Flujo principal:

- El promotor inicia sesión.
- Selecciona "Crear nuevo evento".
- El sistema valida que el venue elegido este aprobado.
- El evento se registra y queda asociado al venue.

Salidas o resultados esperados: Confirmación "Evento creado correctamente".

Reglas aplicadas: Un venue no puede tener dos eventos el mismo dia. Las localidades del evento deben tener precios base uniformes.

Criterio de aceptación: El evento aparece en la lista del promotor y es visible para los clientes.

8.3 Cliente compra tiquetes de un evento

Actor: Cliente

Descripción: Un cliente autenticado compra tiquetes usando su saldo virtual.

Entradas: Evento seleccionado, localidad y cantidad de tiquetes.

Flujo principal:

- El cliente inicia sesión.
- Selecciona un evento y la localidad deseada.
- El sistema calcula el total y valida que el cliente tenga saldo suficiente.
- Se crea una transacción de compra asociada al usuario.
- Se descuentan los fondos del saldo virtual.

Salidas o resultados esperados: Mensaje "Compra realizada con exito".

Reglas aplicadas: Todas las transacciones usan el saldo virtual. Se registra automáticamente una transacción.
Criterio de aceptación: Los tiquetes aparecen en la cuenta del cliente y el saldo se actualiza correctamente.

8.4 Cliente revende un tiquete en el marketplace
Actor: Cliente
Descripción: Un cliente publica en el marketplace un tiquete que desea revender.
Entradas: Identificador del tiquete y precio de reventa.
Flujo principal:

- El cliente elige el tiquete y selecciona la opción "Revender en Marketplace".
- El sistema verifica que el tiquete esté en estado válido, no vencido ni reembolsado.
- Se crea una oferta en el marketplace asociada al usuario.

Salidas o resultados esperados: Mensaje "Oferta publicada exitosamente".
Reglas aplicadas: Solo se pueden revender tiquetes simples. El tiquete debe estar válido. Se registra un log de acción.
Criterio de aceptación: La oferta aparece activa en el marketplace y en el historial de logs.

8.5 Otro cliente realiza una contraoferta y el vendedor la acepta
Actor: Cliente comprador y cliente vendedor
Descripción: Un cliente ofrece un precio menor por una oferta activa y el vendedor decide aceptarla.
Entradas: Identificador de la oferta y valor propuesto.
Flujo principal:

- El comprador selecciona una oferta activa.
- Introduce su contraoferta y la envia.
- El sistema la agrega a la lista de contraofertas de la oferta original.
- El vendedor revisa y acepta la contraoferta.
- El sistema marca la contraoferta como aceptada y la oferta como inactiva.

Salidas o resultados esperados: Mensaje "Contraoferta aceptada" y registro en el log.
Reglas aplicadas: Solo ofertas activas aceptan contraofertas. Aceptar una contraoferta cierra la oferta. Todas las acciones quedan registradas en el log.
Criterio de aceptación: El estado de la oferta cambia a inactiva y el log muestra la acción correspondiente.

8.6 Sistema guarda automáticamente los datos del marketplace
Actor: Sistema (operacion automatica)
Descripción: Cada vez que se realiza una acción en el marketplace, como crear oferta, aceptar contraoferta o eliminar oferta, el sistema guarda el estado actualizado en un archivo JSON.
Entradas: Ninguna.
Flujo principal:

- El sistema detecta una acción relevante.
- Llama a MarketplaceStore.save() para serializar el objeto marketplace.
- Se actualiza el archivo data/marketplace.json.

Salidas o resultados esperados: Archivo actualizado en disco.
Reglas aplicadas: Persistencia automática y archivo JSON único.
Criterio de aceptación: Al reiniciar el sistema, las ofertas y logs se cargan correctamente desde el archivo JSON.

8.7 Administrador consulta el historial de acciones
Actor: Administrador
Descripción: El administrador consulta el log de todas las acciones del marketplace.
Entradas: Ninguna.
Flujo principal:

- El administrador abre la consola de administración.
- Selecciona "Ver historial de acciones".
- El sistema muestra los registros de log.

Salidas o resultados esperados: Lista con usuario, acción, detalle y fecha y hora.
Reglas aplicadas: Todas las acciones generan logs.
Criterio de aceptación: El log incluye todas las acciones recientes del marketplace y coincide con el archivo JSON.

---

9. Persistencia

- La persistencia del sistema se implementó mediante archivos JSON, permitiendo almacenar y recuperar toda la información del sistema (usuarios, eventos, venues y marketplace) de forma automática.
- Se desarrollaron las clases JsonStore y MarketplaceStore, encargadas de la serialización y deserialización de los objetos, garantizando que los datos se mantengan entre ejecuciones.
- Los archivos generados se guardan en la carpeta data/, y son cargados nuevamente al iniciar cada consola.
- Este mecanismo asegura la continuidad de la información y evita la pérdida de datos sin necesidad de usar bases de datos externas.

---

10. Conclusiones

En esta segunda etapa del proyecto se fortaleció el diseño modular de Boletamaster al incorporar la persistencia de datos, las consolas independientes para cada tipo de usuario y las pruebas unitarias que validan su funcionamiento. El sistema ahora puede guardar y recuperar la información de manera confiable, ofreciendo una interacción clara y sencilla para administradores, organizadores y clientes. Su estructura modular facilita futuras extensiones, como la integración con interfaces gráficas o servicios web. Además, el uso de principios de diseño orientado a objetos permitió mantener un código ordenado, fácil de mantener y escalar, cumpliendo así con los objetivos planteados para esta entrega.