ESCUELA SUPERIOR POLITECNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

TALLER #2 Patrones Java

Materia: Diseño de Software

Profesor: Ing. David Jurado

Grupo: 1

Estudiantes:

- Javier Alejandro Gutiérrez Macías
- Henry Moisés Olvera Juez
- Brayan Steven Briones Oleas
- Jair Matías Palaguachi Jalca

Periodo Académico: PAO 2 2024.

Paralelo: 1

Contenido

Sección A: Identificación de Patrones de Diseño

1. Analizar el problema asignado e identificar al menos 3 patrones de diseño relevantes (e.g., Singleton, Factory, Observer, Strategy).

Los patrones de diseño elegido modelar este problema de mejor manera fueron:

- Chain Of Responsibility (Patrón de Comportamiento)
- Decorator (Patrón Estructural)
- Observer (Patrón de Comportamiento)
- Abstract Factory (Patrón Creacional)

2. Explicar en un breve informe por qué se seleccionaron esos patrones y cómo resuelven problemas específicos del sistema asignado

Chain of Responsibility (Soporte y escalamiento de incidentes):

Problema específico: Cuando el cliente reporta un incidente, el sistema debe determinar quién debe resolverlo sin que la lógica se vuelva compleja.

Solución con el patrón: El Chain of Responsibility se seleccionó ya que permite encadenar distintos niveles de soporte (por ejemplo, nivel básico de atención al cliente, soporte técnico y administración del evento). Si el primer nivel no soluciona el problema, lo pasa al siguiente, y así sucesivamente, sin que el cliente o el desarrollador deba hacer nuevo código con lógicas separadas para cada posible camino de resolución. Así se reduce la complejidad y se facilita la incorporación de nuevos niveles de soporte.

Decorator (Opciones adicionales en los tickets):

Problema específico: Los boletos pueden incluir características adicionales (bebidas, estacionamiento, servicios VIP) sin que esto ocasione la necesidad de crear una multitud de subclases para cada combinación de extras.

Solución con el patrón: El Decorator se seleccionó ya que permite agregar un ticket estándar con una o varias funcionalidades extras. Así, en lugar de tener que crear nuevas

clases por cada variación (por ejemplo, "TicketConBebidas", "TicketConBebidasYEstacionamiento"), se parte de un objeto base (el ticket) al que se le agregan decoradores. Esto simplifica el tener que agregar más detalles de ofertas adicionales sin cambiar la estructura interna del ticket original.

Observer:

Problema específico: Las modificaciones en la programación, los cambios en el elenco, las reprogramaciones o cancelaciones deben llegar a los usuarios al instante, sin que estos tengan que consultar activamente el estado del evento.

Solución con el patrón: El Observer se seleccionó ya que permite que cada evento notifique automáticamente a todos los usuarios suscritos sobre cualquier cambio. Cuando ocurre una modificación, el evento (sujeto) envía una actualización a todos sus observadores (usuarios), manteniendo la información sincronizada y mejorando la experiencia del cliente al proporcionarle actualizaciones inmediatas.

Abstract Factory (Creación de familias de objetos relacionados):

Problema específico: El sistema debe soportar diferentes tipos de espectáculos (teatro, microteatro, stand-up, etc.), secciones de asientos diversas (platea, balcón, VIP) y esquemas de precios flexibles, sin complicar el código existente.

Solución con el patrón: El Abstract Factory se seleccionó ya que permite crear familias de objetos (como conjuntos de asientos, políticas de precios y configuraciones de espectáculos) sin especificar las clases concretas. Al definir fábricas abstractas, resulta sencillo añadir nuevas categorías de asientos, tipos de espectáculos o políticas de precio, ya que cada fábrica se encarga de proveer los objetos apropiados. Estos facilitan la adaptación del sistema a nuevas modalidades de eventos que haya sin afectar la lógica ya existente.

Sección B: Diagrama de Casos de Uso

1. Diseñar un diagrama de Casos de Uso del sistema utilizando los patrones seleccionados, con al menos 2 refinamientos (include, extend).

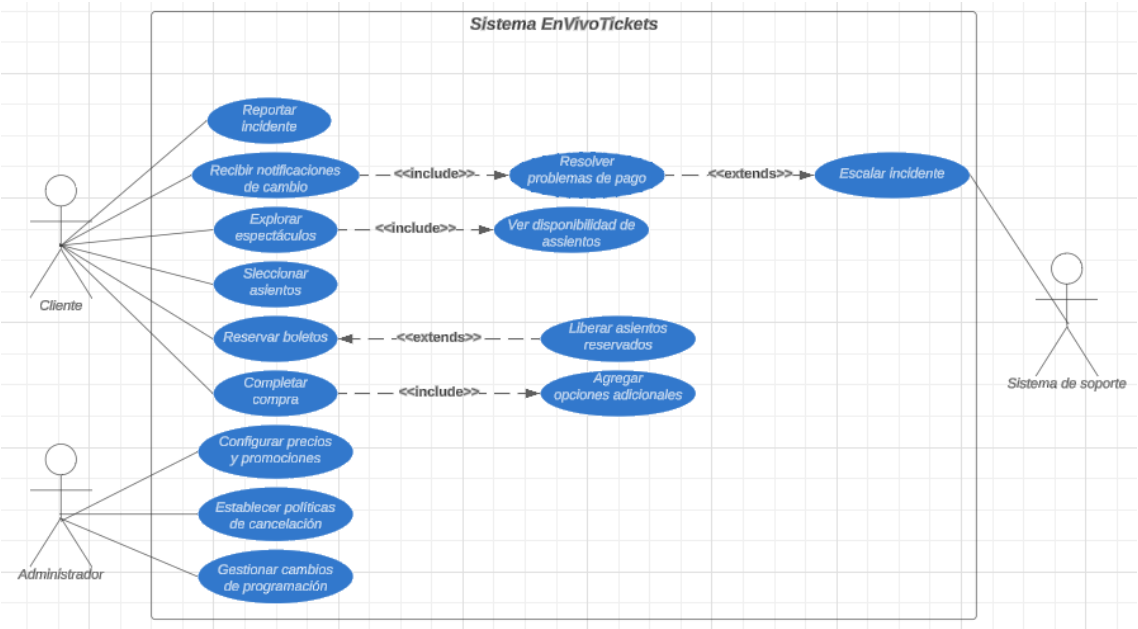


Diagrama UML casos de uso hecho en LucidChart:

https://lucid.app/lucidchart/e41ce6a9-9785-4948-ac1b-046cd7c74a20/edit?viewport_loc=-1549%2C-268%2C2009%2C932%2C0_0&invitationId=inv_5724bd08-09e7-4dbc-839e-2167de46d090

2. Seleccionar 3 casos de uso que involucren a los patrones y documentar:

- **Actores involucrados.**
- **Precondiciones.**
- **Flujo de eventos principal y alternativos.**
- **Postcondiciones.**

Caso de Uso 1: Reportar Incidente

Actores Involucrados:

- Cliente
- Sistema de Soporte

Precondiciones:

- El Cliente debe estar registrado en el sistema.
- El incidente debe estar relacionado con un ticket adquirido o una funcionalidad del sistema.

Flujo de Eventos Principal:

1. El Cliente selecciona la opción de "Reportar Incidente" desde el sistema.
2. El sistema solicita detalles del incidente (tipo, descripción, datos relevantes).
3. El incidente se asigna automáticamente al soporte básico para su resolución.
4. El soporte básico analiza y, si es posible, resuelve el incidente.
5. El Cliente recibe una notificación con la resolución o el estado del incidente.

Flujos Alternativos:

A1. Escalar a Soporte Técnico:

- Si el soporte básico no puede resolver el incidente, este se escala al nivel técnico.
- El Sistema de Soporte técnico analiza el incidente y responde al Cliente.

A2. Escalar a Administración:

- Si el soporte técnico tampoco resuelve el problema, el incidente se escala a la administración del evento.

Postcondiciones:

- El incidente está resuelto o pendiente de resolución en el nivel más adecuado.
- El Cliente es notificado sobre el estado final o el próximo paso.

Caso de Uso 2: Completar Compra

Actores Involucrados:

- Cliente

Precondiciones:

- El Cliente debe haber seleccionado al menos un asiento y haber añadido los tickets al carrito.
- Los tickets seleccionados deben estar en estado reservado.

Flujo de Eventos Principal:

1. El Cliente accede al carrito y revisa los tickets reservados.
2. El sistema solicita los datos de pago y opciones adicionales.
3. El Cliente proporciona los datos de pago y confirma la compra.
4. El sistema procesa el pago y genera los tickets.
5. El Cliente recibe los tickets confirmados en su cuenta y/o correo electrónico.

Flujos Alternativos:

A1. Agregar Opciones Adicionales:

- Antes de confirmar el pago, el Cliente selecciona opciones adicionales como estacionamiento o bebidas.
- El sistema actualiza el costo total de la compra y muestra el desglose.

A2. Error en el Pago:

- Si ocurre un error en el procesamiento del pago, el sistema notifica al Cliente y libera los tickets reservados tras un tiempo límite.

Postcondiciones:

- Los tickets están confirmados y asignados al Cliente.
- Si se seleccionaron opciones adicionales, estas se reflejan en los tickets emitidos.

Caso de Uso 3: Recibir Notificaciones de Cambios

Actores Involucrados:

- Cliente
- Administrador

Precondiciones:

- El Cliente debe estar registrado en el sistema y haber comprado boletos para un evento.
- El Administrador debe haber realizado un cambio en el evento (modificación, reprogramación o cancelación).

Flujo de Eventos Principal:

1. El Administrador realiza un cambio en la programación del evento.
2. El sistema identifica a los Clientes que tienen boletos para el evento afectado.
3. El sistema genera y envía notificaciones automáticas a todos los Clientes afectados.
4. El Cliente recibe la notificación con los detalles del cambio y las opciones disponibles (reembolso, cambio de fecha, etc.).

Flujos Alternativos:

A1. Notificación Fallida:

- Si el Cliente no recibe la notificación (problema en el correo, configuración de usuario), el sistema reintenta enviarla o solicita al Cliente verificar sus datos de contacto.

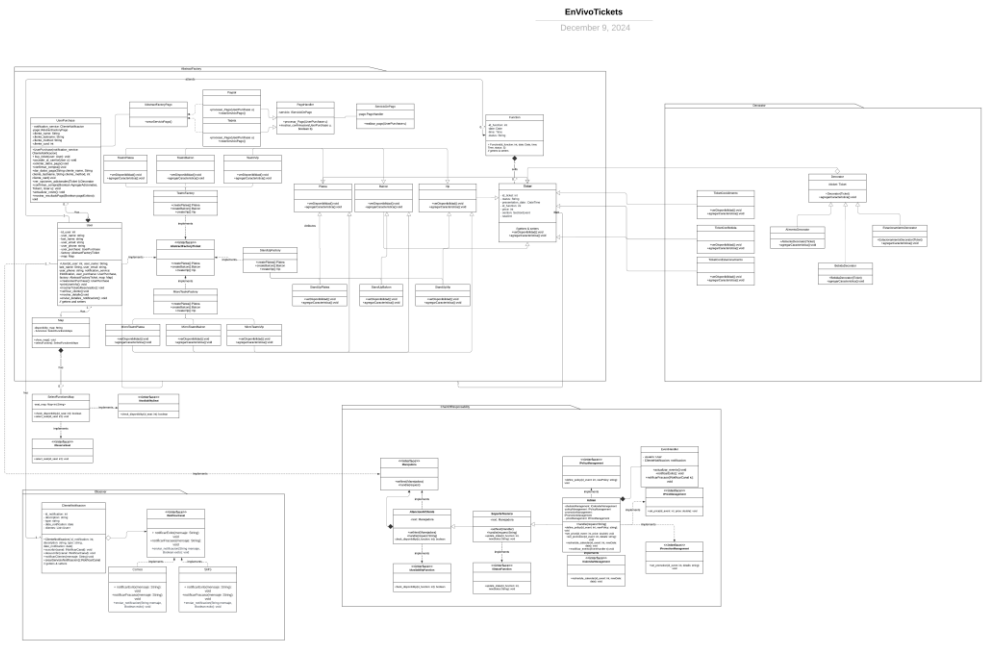
Postcondiciones:

- El Cliente está informado sobre el cambio en el evento.
- El sistema registra la entrega de la notificación y la opción seleccionada.

Sección C: Diagrama de Clases con Patrones de Diseño

1. **Crear un diagrama de clases que refleje las clases, interfaces y relaciones, integrando los patrones de diseño identificados en la Sección A.**
2. **Justificar cómo cada patrón de diseño fue incorporado en las clases y cómo respeta principios SOLID.**
3. **Asegurarse de que el diagrama sea consistente con los Casos de Uso seleccionados.**

Diagrama de Clases:



Enlace del diagrama de clases en Lucid Chart:

https://lucid.app/lucidchart/a0896708-f10a-4682-9b2e-06f3cdce2e13/edit?viewport_loc=708%2C795%2C3306%2C1536%2CHWEp-vi-RSFO&invitationId=inv_d2130265-d61c-458d-977b-e776cc8012c3

Justificación principios SOLID, relaciones y patrones:

El diagrama de clases aplica los principios SOLID de forma detallada para lograr un diseño modular, extensible y mantenible. Siguiendo el Principio de Responsabilidad Única (SRP), se ha dividido las clases e interfaces según sus responsabilidades específicas: User gestiona los datos del usuario, Ticket representa los detalles de los tickets, y UserPurchase se encarga exclusivamente de gestionar las compras, delegando la lógica de notificaciones a ClienteNotificacion. Además, el diseño sigue el Principio de Abierto/Cerrado (OCP), ya que se utilizó como IPriceManagement, IAvailabilitySeat y IPromotionManagement, interfaces permitiendo extender funcionalidades (como la gestión de precios o disponibilidad) sin necesidad de modificar las clases existentes, asegurando así que el sistema pueda adaptarse a cambios futuros sin comprometer su integridad. El Principio de Sustitución de Liskov (LSP) se respeta al permitir que cualquier implementación concreta de interfaces como IAvailabilitySeat o IReserveSeat pueda ser intercambiada sin afectar la lógica del cliente, garantizando que las clases derivadas cumplan con los contratos de sus interfaces. Por otro lado, al separar las funcionalidades en interfaces más específicas, como ICalendarManagement para reprogramar eventos y IPolicyManagement para definir políticas, se sigue el Principio de Segregación de Interfaces (ISP), evitando que las clases deban implementar métodos que no necesitan. Finalmente, el Principio de Inversión de Dependencias (DIP) se aplica al depender de abstracciones en lugar de clases concretas. Por ejemplo, la clase Admin depende de instancias tales como: ICalendarManagement, IPolicyManagement, IPromotionManagement y IPriceManagement, las cuales son interfaces permitiendo flexibilidad y facilitando la inyección de dependencias.De esta manera, se logra mantener el principio SOLID en todas las clases que lo necesiten. En la sección de relaciones, se tiene que la interfaz Manejadora tiene una relación de composición con la clase User. Las relaciones de composición se utilizan cuando una clase contiene a otra y es responsable de su ciclo de vida. En este caso, si no existiera la clase User, no podría existir un servicio que maneje los problemas que presente en usuario. Esta misma lógica la maneja la clase User con UserPurchase, Map con SelectFunctionsMap y Functions con Ticket. Para las relaciones de agregación se tiene la clase ClienteNotificacion con la interfaz INotificarCanal. El motivo de esta relación es porque la clase ClienteNotificacion

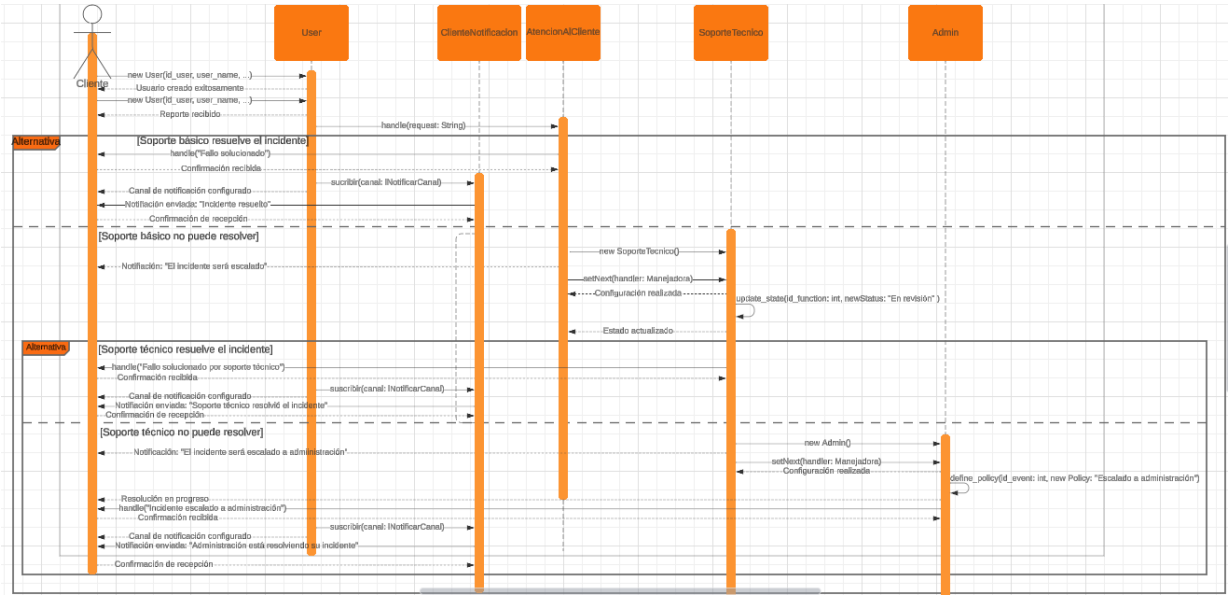
requiere una instancia del tipo INotificarCanal en algunos de sus métodos como lo es “suscribirse”. Sin embargo, si esta clase dejase de existir la interfaz INotificarCanal podría seguir existiendo, esperando ser implementada por otra clase a futuro. Esta misma lógica la maneja la clase Decorator con Ticket. En cuanto a las demás clases, usan relaciones de asociación. Las relaciones de asociación se utilizan cuando una clase necesita interactuar con otra sin poseerla directamente.

Finalmente, se incorporó cuatro patrones de diseño en todo el diagrama de clases. El primero fue el patrón creacional Abstract Factory, el cual se lo incorporó en conjunto con la clase Ticket e interfaces respectivas (fábricas) para poder crear combinaciones de Tickets de acuerdo con el tipo de espectáculo y el tipo de asiento que tenga el ticket. Segundo, se incorporó el patrón estructural Decorator en conjunto con una clase decoradora y la clase Ticket, se empleó este patrón creando clases hijas: “AlimentoDecorator”, “BebidasDecorator” y “EstacionamientoDecorator” para agregarles características respectivas al ticker que manejen. Luego, se empleó el patrón de comportamiento Observer para el manejo de las notificaciones a los usuarios, para ello se creó la clase “ClienteNotificacion” y la interfaz ”INotificarCanal”que la emplean los medios de comunicación por donse se notificarán a los usuarios, en este caso “Correo” y “SMS”. Por último, se empleó el patrón de comportamiento ChainOfResponsability para el manejo de la lógica de solución de problemas notificados por los usuarios, para ello se creó la interfaz “Manejadora” la cual se encarga de delegar parcial o totalmente el problema notificado por el usuario a los demás departamentos, comenzando por la clase “AtencioAlCliente” el cual sino logra manejar el problema en su totalidad lo delegará a la clase “SoporteTecnico” y finalmente si esta no logra manejar el problema lo delegará a la clase “Admin” que manejará el problema haciendo uso de métodos implementados por las interfaces “IPolicyManagement”, “IPriceManagement”, “IPromotionManagement” y “ICalendarManagement”.

Sección D: Diagramas de Secuencia

1. **Diseñar 3 diagramas de secuencia que representen la ejecución de los casos de uso seleccionados, mostrando: i. Mensajes entre objetos. ii. Creación de instancias y llamadas a métodos según los patrones de diseño.**
2. **Garantizar que los diagramas sean coherentes con el diagrama de clases.**

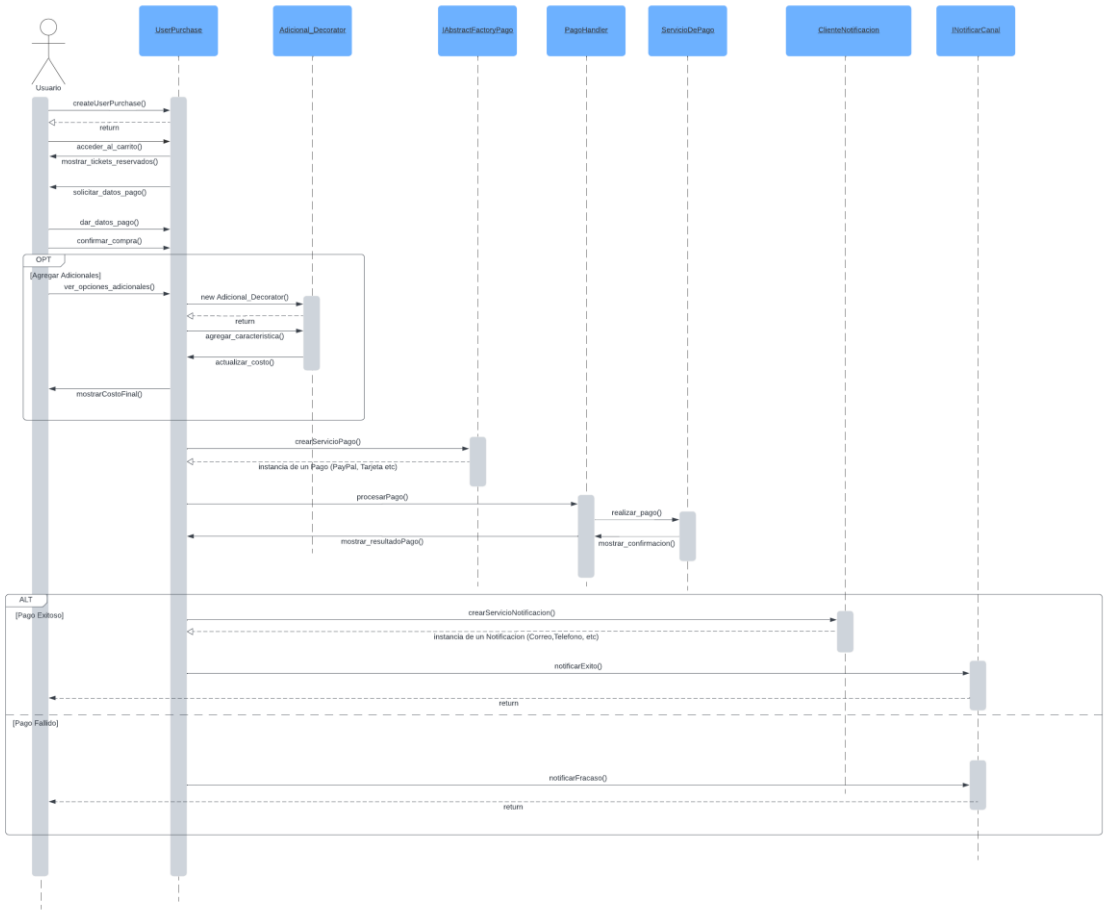
Diagrama de Secuencia del Caso de uso 1



Enlace del diagrama de secuencia en Lucid Chart:

https://lucid.app/lucidchart/e31659f0-e119-4cd1-839d-256e5fcdef65/edit?viewport_loc=-1728%2C-684%2C2688%2C1270%2C0_0&invitationId=inv_f1bd71d4-a977-4034-a6cb-7ab7ce9be289

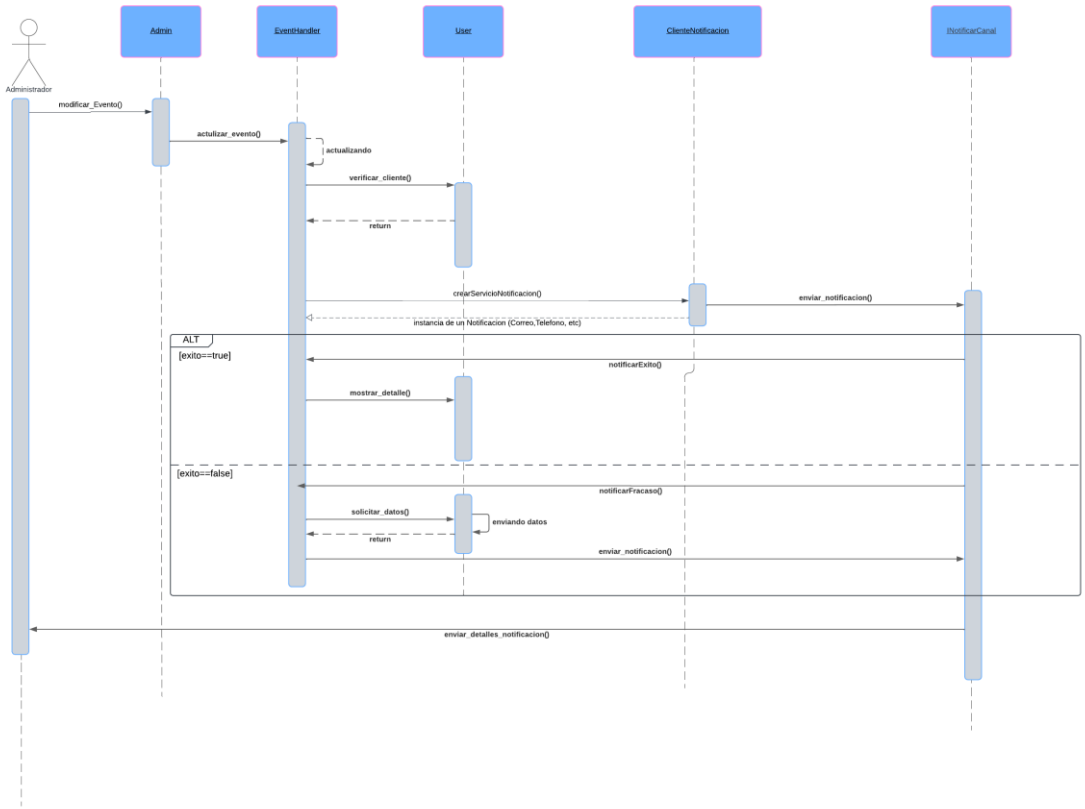
Diagrama de Secuencia del Caso de uso 2



Enlace del diagrama de secuencia en Lucid Chart:

https://lucid.app/lucidchart/60acba8a-e06f-432d-8c39-078c3465baac/edit?viewport_loc=-395%2C-1244%2C4842%2C1707%2C0_0&invitationId=inv_5dd02130-cada-4c6d-95df-330a2d764d89

Diagrama de Secuencia del Caso de uso 3



Enlace del diagrama de secuencia en Lucid Chart:

https://lucid.app/lucidchart/496ac970-689f-44b9-a98b-bda71fb007ff/edit?viewport_loc=-1457%2C-111%2C3397%2C1429%2C0_0&invitationId=inv_6a3e0656-fb7a-4f3b-853a-6e61de89bb3d

Sección E: Generación de Código en Java

Enlace al Repositorio:

https://github.com/BrayanBriones/Tarea_2_G3_EnVivoTickets.git