ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Diseño de Software

Tarea3-Travel Ease

INTEGRANTES:

Perdomo Ordoñez Paul Isaac

Herrera Nieto Christian Alexander

Muñoz Sanchez Salvador Gabriel

Rosado Alcivar Enrique Gabriel

Grupo:

7

Profesor:

Jurado Mosquera David Alonso

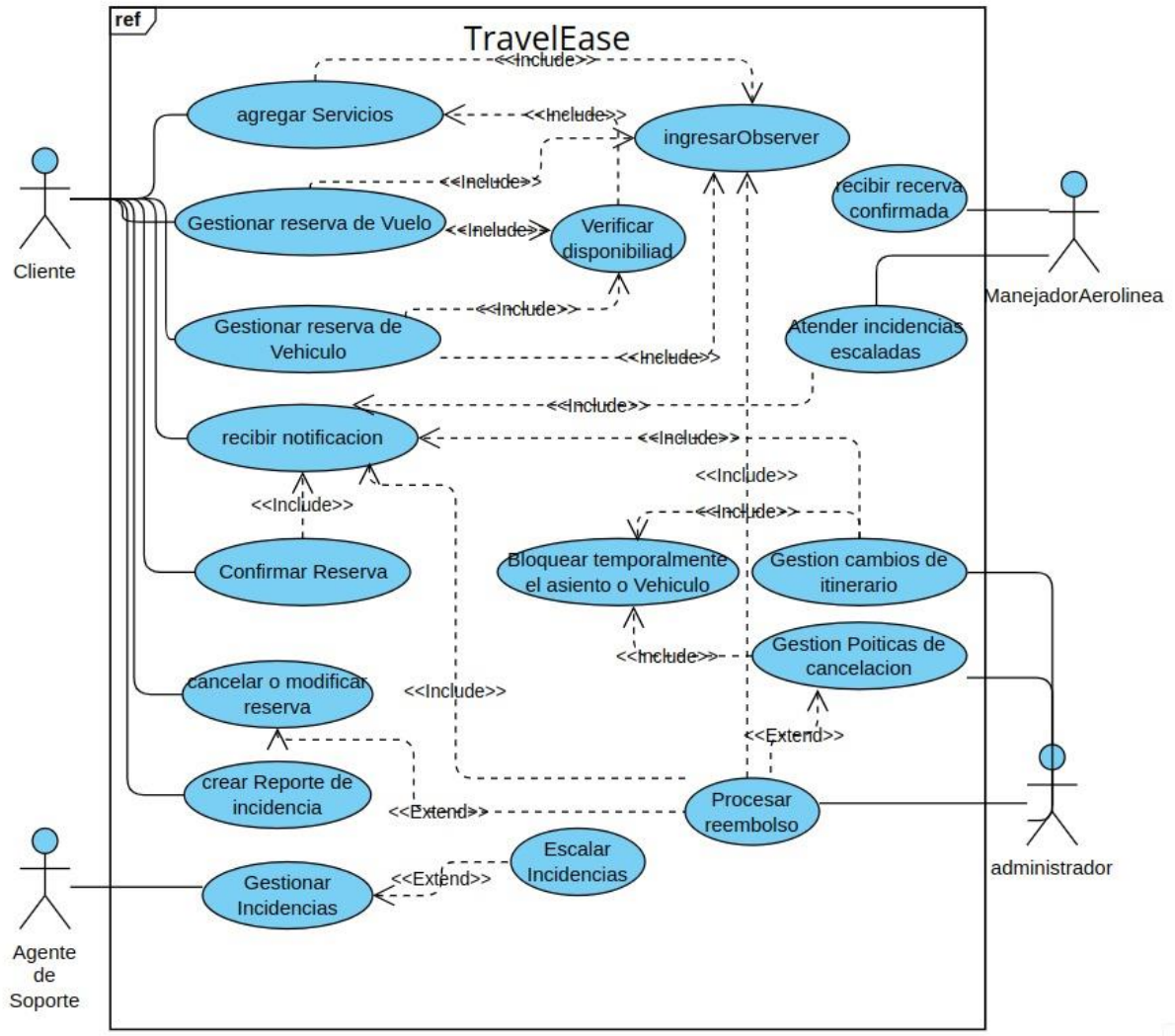
Periodo Académico:

PAO II – 2024

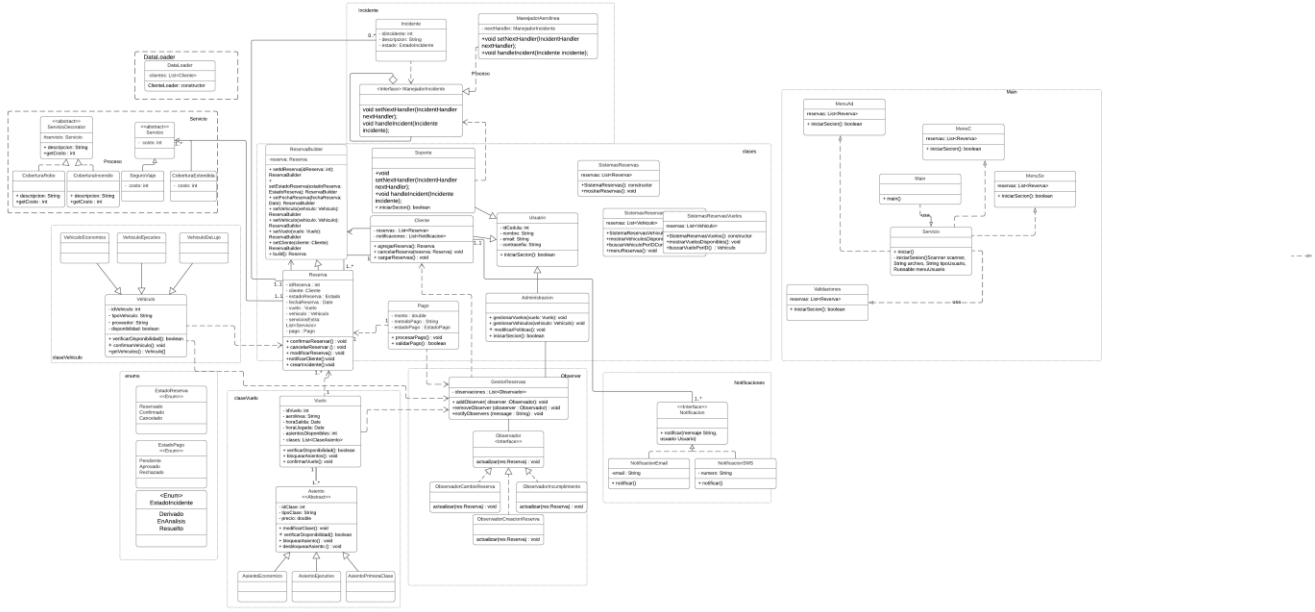
Tabla de contenido

Diagramas

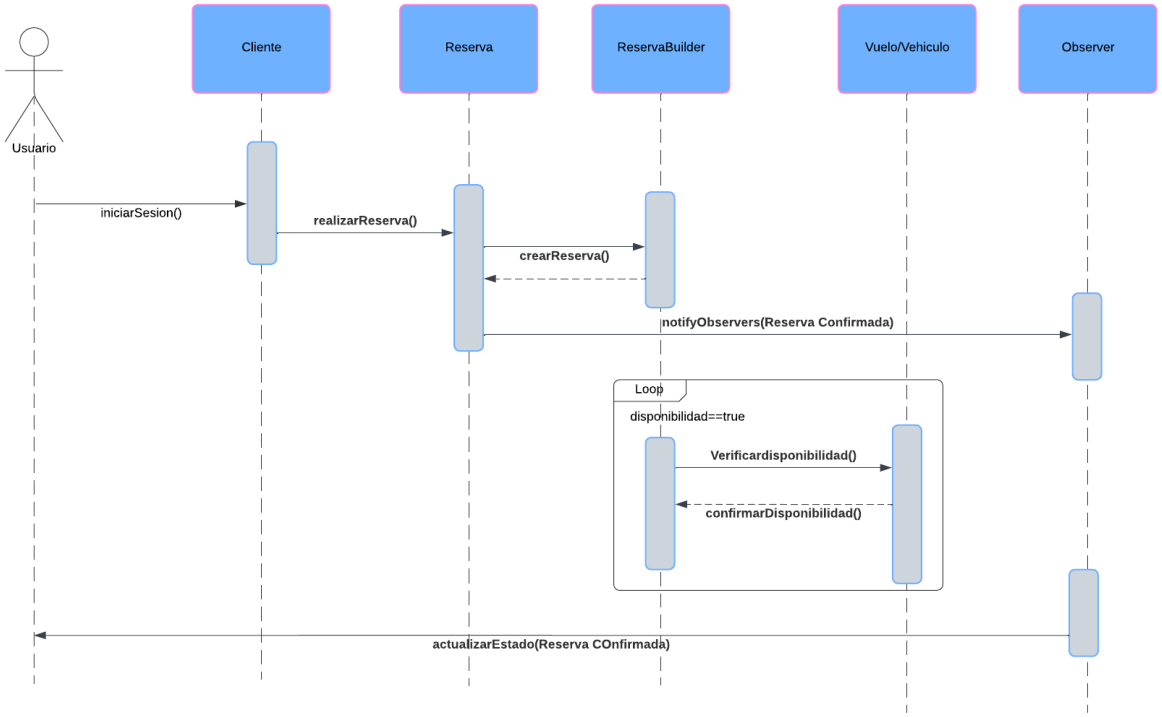
1. Diagrama de Caso de Uso



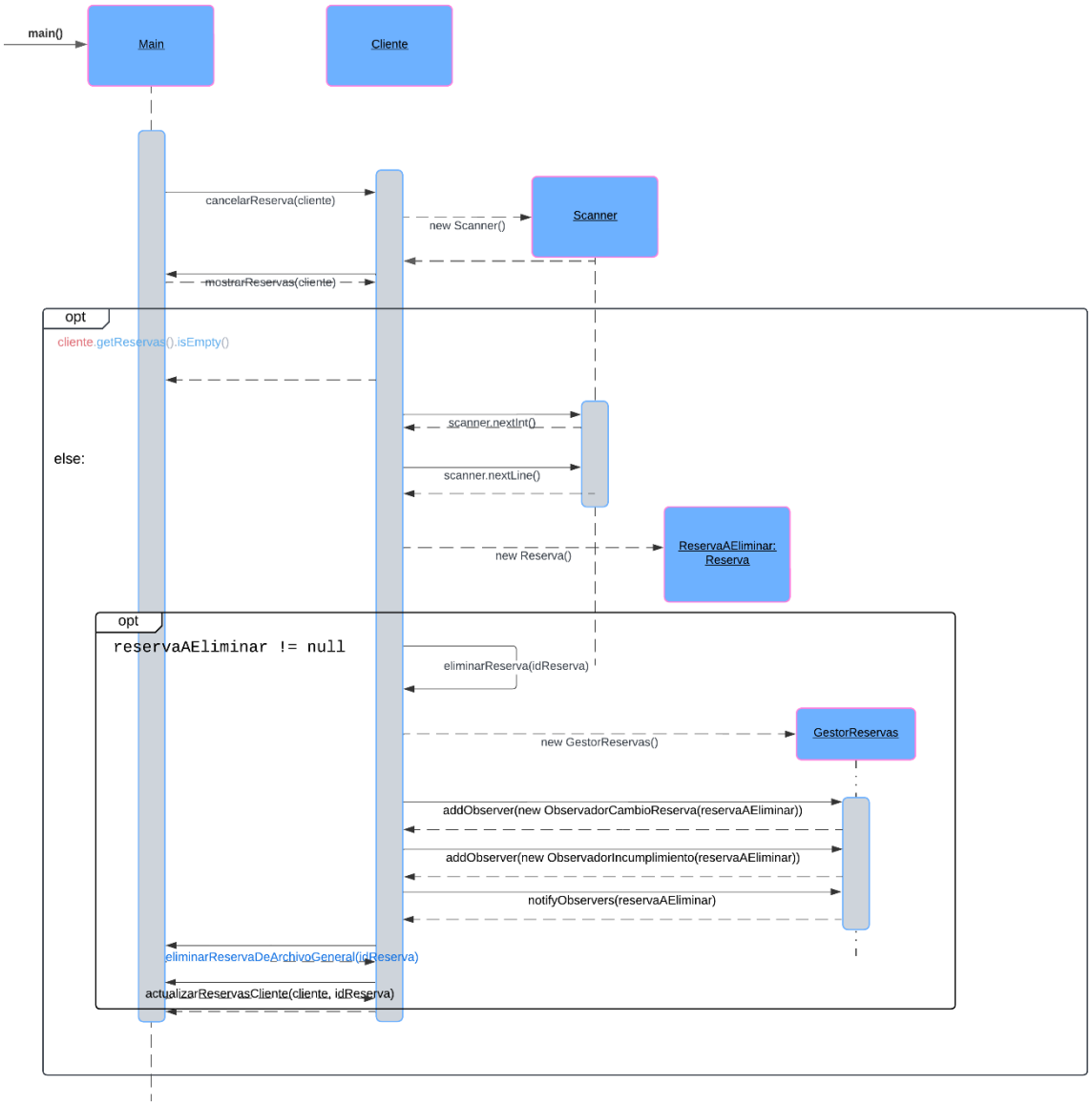
2. Diagrama de Clase



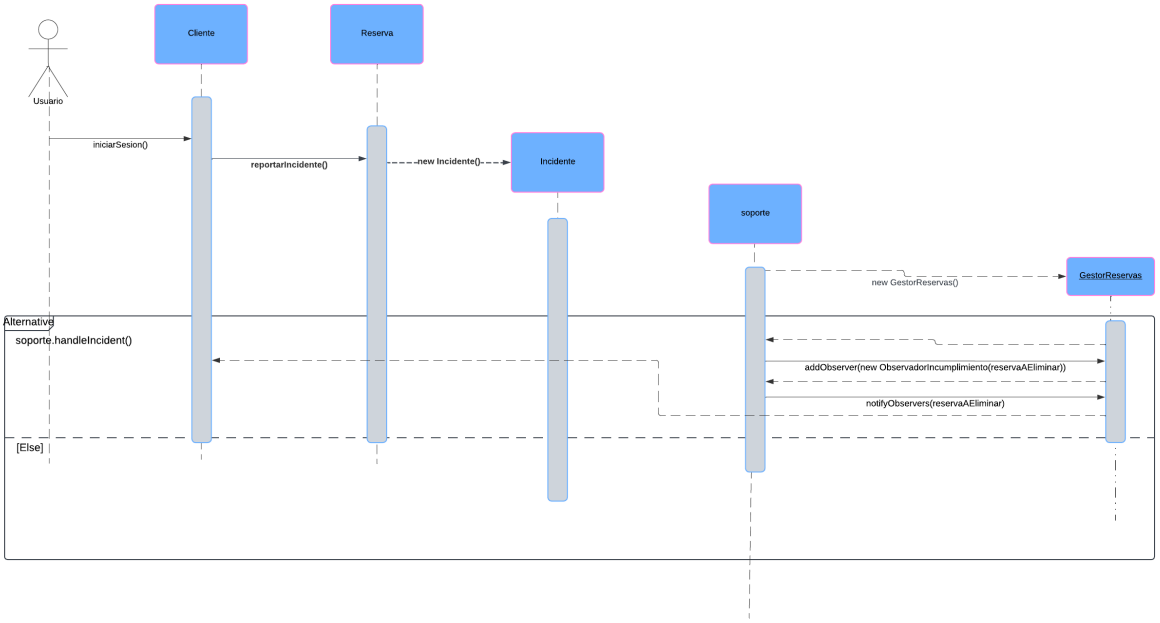
3. Diagrama De Secuencia #1



4. Diagrama De Secuencia #2



5. Diagrama De Secuencia #3



Informe Evaluativo sobre el Diseño Original y Propuestas de Mejora

1. Evaluación de la Flexibilidad del Diseño Original para Incorporar Cambios

El diseño original del sistema presenta una estructura orientada a objetos sólida, pero su capacidad de adaptarse a cambios y nuevas funcionalidades podría mejorarse mediante ciertos ajustes en la arquitectura.

Fortalezas:

- **Uso de Herencia y Polimorfismo**: La implementación de clases abstractas, como `Vehiculo`, y las clases concretas como `VehiculoEconomico` o `VehiculoLujo`, permite una cierta flexibilidad al agregar nuevos tipos de vehículos. Este enfoque de herencia y polimorfismo facilita la adición de nuevos tipos de vehículos sin modificar el código base, aprovechando la reutilización y extensibilidad.
- **Uso del Patrón Builder**: El patrón **Builder** en la clase `Reserva` facilita la construcción de objetos complejos y facilita la adición de nuevas características a las reservas sin afectar el código de otras clases.

Oportunidades de Mejora:

- **Flexibilidad para Nuevas Funcionalidades**: Aunque la estructura es adecuada para los cambios actuales, algunos aspectos del diseño no están tan preparados para cambios de gran escala, especialmente cuando se trata de agregar nuevos métodos de pago, tipos de vehículos, o nuevas características a las reservas. Actualmente, estos tipos están fuertemente acoplados a sus implementaciones específicas.

2. Beneficios y Limitaciones de los Patrones de Diseño Aplicados en el Proyecto

Beneficios:

1. **Patrón Factory Method**:
    a. El **Factory Method** implementado en el código para la creación de `Vehiculos` y `Pagos` permite crear objetos sin exponer la lógica de creación al cliente, lo que facilita la expansión del sistema. Al implementar fábricas para cada tipo específico de vehículo o pago, se

puede agregar nuevas clases de vehículos o métodos de pago sin necesidad de modificar el código cliente.

   b. **Beneficios**: Centraliza la creación de objetos y permite agregar nuevos tipos fácilmente sin afectar al código que utiliza estos objetos.

2. **Patrón Observer**:
   a. El uso del patrón **Observer** permite que el sistema esté preparado para notificar a los usuarios u otros componentes cuando se produzcan cambios importantes, como el estado de una reserva o el pago. Esto es útil para mantener el sistema desacoplado, ya que los observadores pueden reaccionar a los cambios sin que el sujeto tenga que saber de sus implementaciones.
   b. **Beneficios**: Desacoplamiento entre el sujeto y los observadores, lo que permite una alta flexibilidad para agregar nuevos tipos de notificaciones.
3. **Patrón Decorator**:
   a. El **Decorator** aplicado a los servicios permite agregar comportamientos adicionales a los objetos sin alterar sus clases originales. Esto es útil para extender las funcionalidades de los servicios, como la adición de características adicionales a las reservas o notificaciones, sin necesidad de modificar el código base.
   b. **Beneficios**: Facilidad para añadir responsabilidades adicionales a los objetos sin modificar su implementación básica.

Limitaciones:

1. **Acoplamiento de Componentes**:
   a. El código está diseñado de manera que los componentes (como `Vehiculo`, `Pago`, y `Reserva`) están muy acoplados, lo que puede hacer que sea difícil agregar nuevos comportamientos sin modificar las clases existentes.
   b. **Limitación**: La adición de nuevos tipos de vehículos o pagos requiere modificaciones en el código base, lo que puede dificultar la expansión del sistema.
2. **Persistencia en Archivos de Texto**:
   a. El uso de archivos de texto para almacenar información puede ser adecuado para una pequeña escala, pero no es escalable. En sistemas más grandes, la manipulación de archivos de texto se vuelve ineficiente.
   b. **Limitación**: La persistencia de datos basada en archivos no soporta bien consultas complejas o grandes volúmenes de datos, y no es ideal para mantener relaciones entre objetos.

3. Propuestas de Mejora al Diseño Original

Para mejorar la flexibilidad del diseño y soportar nuevos requerimientos de manera más fluida, propongo las siguientes mejoras:

A. Uso de Interfaces y Herencia para Tipos de Vehículos y Pagos

- **Vehículos y Pagos como Interfaces**: Asegurar que `Vehiculo` sea una **interfaz** o **clase abstracta** que define el comportamiento general, mientras que las clases concretas como `VehiculoEconomico` o `VehiculoLujo` implementan esta interfaz. Lo mismo se aplica a los **Pagos**, permitiendo la creación de nuevos tipos de vehículos y pagos sin modificar el código base.

C. Mejora del Patron Observer con Canal de Notificación más Flexible

- Ampliar el uso del patrón **Observer** para permitir una **mejor gestión de notificaciones** mediante diferentes canales (por ejemplo, correo electrónico, SMS, etc.). Implementar una **estrategia de notificación** que permita cambiar dinámicamente el tipo de notificación a enviar, basándose en las preferencias del usuario o tipo de evento.

D. Uso de Inyección de Dependencias

- Implementar **Inyección de Dependencias** para mejorar el desacoplamiento entre las clases. Esto permitirá una mayor flexibilidad y escalabilidad en la gestión de las dependencias entre objetos. Usar un framework como **Spring** puede facilitar la gestión de estas dependencias.

Conclusión

El diseño original del sistema tiene una base sólida, pero podría beneficiarse de algunas mejoras clave para aumentar su flexibilidad y escalabilidad. Las principales recomendaciones son: el uso de interfaces para tipos de vehículos y pagos, la migración de la persistencia de archivos a una base de datos, la ampliación del patrón **Observer** para una gestión de notificaciones más flexible, y la implementación de **Inyección de Dependencias** para desacoplar los componentes del sistema. Con estas mejoras, el sistema podrá adaptarse mejor a nuevos requisitos y cambios en el futuro.

Links

Github: https://github.com/Herrera2005/Tarea2Patrones.git

Diagrama de uso: https://online.visual-paradigm.com/w/cpshlpme/diagrams/#diagram:workspace=cpshlpme&proj=2&id=20&type=UseCaseDiagram&width=11&height=8.5&unit=inch

Diagrama de clase: https://lucid.app/lucidchart/2297f08f-6e79-4053-8c04-b162c390b93c/edit?viewport_loc=-1564%2C-826%2C4402%2C2054%2C0_0&invitationId=inv_8c844a89-1e5f-47b2-9fc5-ba0c1aa6bf89

Diagramas de secuencia #1: https://lucid.app/lucidchart/23bf59fd-7f03-4c64-ad0d-a36deda0d48a/edit?viewport_loc=-712%2C-241%2C2215%2C1033%2C0_0&invitationId=inv_cbb1153f-0f52-4220-9c30-f020e77a0cf8

Diagramas de secuencia #2: https://lucid.app/lucidchart/fc6dba49-937d-41b0-a933-9dbdeb75f294/edit?view_items=K1UyUkpB2mjI&invitationId=inv_6b213f8e-b79a-4afc-8279-4748327b55f0

Diagramas de secuencia #3: https://lucid.app/lucidchart/f2eb6405-ab52-4634-8d67-4473dfac53df/edit?view_items=wYUyyqaKfF9h&invitationId=inv_1608339e-91f7-4f74-b21f-5a15ecdb82cf