✨ Descripción del Contexto

Uber es una plataforma de transporte que conecta conductores con pasajeros para ofrecer servicios de movilidad 🚗. Los conductores reciben pagos por cada viaje realizado, dependiendo de la distancia, tiempo y tarifa base establecida por la plataforma 💵.

En este contexto, los conductores necesitan llevar un registro detallado de sus viajes ✉️ para analizar su desempeño, calcular ingresos y optimizar sus horarios de trabajo ⏳.
Actualmente, Uber proporciona información en su aplicación, pero no permite personalizar el almacenamiento ni realizar cálculos según las necesidades del conductor 🔍.
La falta de un sistema propio de gestión de viajes puede dificultar la toma de decisiones 🧐, ya que el conductor no tiene acceso inmediato a métricas clave como el promedio de ingresos por hora o la distribución geográfica de los viajes 🌐.

📈 Definición de Necesidad

💡 a. Explicación del problema o necesidad

Como conductor de Uber, es fundamental llevar un registro detallado de los viajes realizados para evaluar el rendimiento y optimizar los horarios de trabajo ⏰. Sin embargo, actualmente no se cuenta con una herramienta personalizada que permite almacenar, consultar y analizar esta información de manera eficiente 📝.

⚠️ b. Consecuencias de la falta de una solución

❌ Dificultad para calcular el promedio de ingresos por hora de trabajo.
❌ Falta de un registro histórico de viajes para análisis posterior.
❌ Imposibilidad de evaluar cuáles son los horarios y localidades más rentables.
❌ Dependencia de los reportes de Uber, que pueden ser limitados o no mostrar la información de manera personalizada.

📚 Justificación de la Solución

💻 a. Beneficios concretos de la implementación

La creación de una aplicación en Java para registrar y consultar los viajes permitirá al conductor:

✅ Tener un control detallado de sus ingresos diarios.
✅ Calcular automáticamente el promedio de ingresos por hora de trabajo.
✅ Identificar patrones en los viajes (horarios y zonas más rentables).
✅ Facilitar la toma de decisiones sobre cuándo y dónde trabajar para maximizar las ganancias.
✅ Tener un respaldo de los datos sin depender de la aplicación de Uber.

💸 b. Cuantificación de los beneficios

Si el conductor puede identificar las horas y zonas más rentables, podría optimizar su tiempo de trabajo y aumentar sus ingresos 📊. Por ejemplo:

Si actualmente trabaja 10 horas y gana en promedio $18k por hora, pero identifica que en ciertos horarios puede ganar $25k por hora, podría incrementar sus ingresos diarios en un 39% aproximadamente ⬆️.

Al reducir el tiempo en zonas poco rentables, podría hacer más viajes en menos tiempo, aumentando su eficiencia ⏳.

Contar con un registro claro también facilita el control de ingresos 🧾.

🚀 Propuesta de Solución

🛠️ a. Desarrollo de una aplicación en Java

Para abordar la necesidad identificada, se propone el desarrollo de una aplicación en Java que permita a los conductores de Uber registrar y consultar información sobre sus viajes 📝. Esta aplicación brindará herramientas para gestionar datos de manera eficiente y visualizar métricas clave en tiempo real 📊.

📌 b. Funcionamiento de la aplicación

La aplicación permitirá a los conductores:

Registrar manualmente cada viaje ingresando localidad, valor, hora y duración.
Almacenar la información dentro de la consola para futuras consultas 📂.
Calcular métricas clave automáticamente, como:
Número total de viajes en un día 📈.
Ingreso total diario 💰.
Promedio de ingresos por hora trabajada ⏳.
visualizar datos dentro de la consola para análisis.
Interfaz simple y amigable, con menús claros y opciones intuitivas.

🎯 c. Funcionalidades principales

Ingreso de datos de viajes: Permite registrar localidad, hora, valor del viaje, etc.
Cálculo automático de ingresos: Calcula totales diarios y promedio por hora.

Historial de viajes dentro de la consola: Guarda y permite consultar viajes previos.
Interfaz en consola: Fácil de usar y sin necesidad de conexión a internet.

🔧 Mejoras en la arquitectura del sistema

El proyecto fue diseñado bajo una arquitectura orientada a objetos siguiendo los principios SOLID, los cuales son fundamentales para crear sistemas que sean fáciles de mantener, extender y refactorizar en el futuro. La implementación de estos principios ha permitido que el código sea más robusto y flexible. Aquí se detallan los principios aplicados:

1. **Responsabilidad Única (Single Responsibility Principle - SRP)**:
   Cada clase y método en el sistema cumple con una sola responsabilidad. Por ejemplo, la clase GeneradorRecibosCSV se encarga exclusivamente de generar los recibos y almacenarlos en archivos CSV, mientras que las clases como RegistroViajes se encargan de gestionar la lógica de los viajes y sus registros. Esta separación de responsabilidades asegura que el código sea más fácil de entender, mantener y extender sin que un cambio en una parte del sistema afecte otras áreas de forma no deseada.
2. **Abierto/Cerrado (Open/Closed Principle - OCP)**:
   El diseño del sistema permite que nuevas funcionalidades se agreguen sin necesidad de modificar el código existente. Por ejemplo, si se desean agregar nuevas métricas de cálculo o nuevas formas de persistir los datos, esto se puede hacer añadiendo nuevas clases o interfaces sin alterar el código central. Las interfaces IPersistencia y la implementación del GeneradorRecibosCSV permiten agregar fácilmente otros métodos de persistencia si es necesario, como base de datos o almacenamiento en la nube.
3. **Sustitución de Liskov (Liskov Substitution Principle - LSP)**:
   Las clases que implementan interfaces o extienden clases abstractas pueden ser sustituidas por otras sin romper el comportamiento esperado del sistema. Un ejemplo claro es la relación entre RegistroViajes y la interface IPersistencia, que permite cambiar la implementación de la persistencia de datos sin que el resto del sistema se vea afectado.
4. **Segregación de la Interfaz (Interface Segregation Principle - ISP)**:
   En lugar de crear interfaces grandes y complejas, se crearon interfaces específicas como IPersistencia, que define los métodos necesarios para la persistencia de datos, permitiendo que las clases implementen solo los métodos que realmente necesitan. Esto mejora la modularidad y evita que las clases tengan que implementar métodos innecesarios.
5. **Inversión de Dependencias (Dependency Inversion Principle - DIP)**:
   El sistema depende de abstracciones (interfaces), no de implementaciones concretas. Por ejemplo, la clase RegistroViajes depende de la interfaz IPersistencia en lugar de depender directamente de una clase concreta como GeneradorRecibosCSV. Esto permite cambiar la implementación de la persistencia sin afectar a la clase que realiza el registro de los viajes, promoviendo una mayor flexibilidad.

Además, se añadieron excepciones personalizadas, como la clase DatoInvalidoException, para garantizar que el sistema gestione los errores de manera controlada, evitando que el programa termine inesperadamente ante entradas incorrectas o problemas en el procesamiento de los datos.

🗃️ Persistencia de datos mediante archivos CSV

La persistencia de los datos es un aspecto crucial para la aplicación, ya que garantiza que la información de los viajes, los ingresos y otros datos importantes se conserven entre diferentes sesiones. En este proyecto, se ha implementado un sistema de persistencia utilizando archivos CSV, lo que facilita la gestión de grandes volúmenes de datos y la posibilidad de abrir y modificar los archivos con otras aplicaciones (como hojas de cálculo).

El componente encargado de esta funcionalidad es la clase GeneradorRecibosCSV. Esta clase permite almacenar los datos de los viajes realizados, generando un recibo en formato

CSV con la información relevante como el valor del viaje, la duración y la hora en que se realizó. Los beneficios de esta solución incluyen:

- **Facilidad de acceso**: Los recibos generados se guardan como archivos CSV, que pueden ser fácilmente abiertos y consultados tanto dentro de la aplicación como mediante otras herramientas como Excel o Google Sheets.
- **Independencia de la plataforma**: La aplicación no depende de servicios externos para almacenar los datos, lo que le da mayor control sobre la información y mayor autonomía.
- **Persistencia confiable**: Los datos se guardan de forma estructurada y segura, lo que asegura su integridad durante el uso prolongado de la aplicación.

⚠ **Manejo de Excepciones**

El manejo adecuado de excepciones es esencial para que una aplicación sea robusta y no falle ante situaciones inesperadas. En este sistema, se implementó un manejo de excepciones detallado que cubre casos como:

1. **Datos inválidos**: Si el usuario ingresa información incorrecta, como valores no numéricos cuando se espera un número o un formato incorrecto de fecha, se lanza una excepción personalizada DatoInvalidoException. Esto evita que el programa se cierre abruptamente y le proporciona al usuario una retroalimentación clara sobre el error cometido.
2. **Problemas al leer o escribir archivos**: Al trabajar con archivos, es común encontrar excepciones como FileNotFoundException o IOException. Se crearon excepciones personalizadas para estos casos, como ArchivoException, que ayudan a manejar los errores de lectura o escritura de forma controlada, asegurando que el sistema proporcione mensajes útiles al usuario y no se cierre de forma inesperada.
3. **Prevención de fallos en tiempo de ejecución**: Mediante la implementación de bloques try-catch, se garantiza que cualquier excepción generada durante la ejecución sea atrapada y manejada correctamente, evitando que el sistema falle de manera inesperada y manteniendo la estabilidad del sistema.