Jon Stegherr Llop, Myroslav Tatchyn y Míriam Rodríguez Carranza

Informe de la Práctica: Diseño y Pruebas Unitarias.

Jon Stegherr Llop, Myroslav Tatchyn y Míriam Rodríguez Carranza

Índice

Jon Stegherr Llop, Myroslav Tatchyn y Míriam Rodríguez Carranza

Resumen del Trabajo Realizado

El trabajo desarrollado aborda la implementación del caso de uso "Realizar desplazamiento" dentro del sistema de micromovilidad compartida, siguiendo las especificaciones descritas en el enunciado.

La práctica se estructura en dos partes:

1. **Parte I (obligatoria):** Desarrollo de la funcionalidad núcleo para realizar desplazamientos, incluyendo las siguientes actividades:
    - Implementación de las clases valor (GeographicPoint, StationID, VehicleID y UserAccount).
    - Desarrollo de la lógica para emparejar y desemparejar vehículos.
    - Creación de servicios auxiliares y controladores, como `JourneyRealizeHandler` y `JourneyService`.
    - Manejo de excepciones específicas para validar estados y garantizar la robustez del sistema.
    - Pruebas unitarias exhaustivas para todos los componentes clave y todos los casos requeridos en el enunciado.
2. **Parte II (opcional):** Extensión del sistema para incluir el caso de uso "Realizar pago monedero". En esta parte se desarrollaron las clases y métodos relacionados con el pago mediante monedero electrónico, además de las pruebas correspondientes.

Criterios de Diseño Empleados

Principios SOLID

1. **Responsabilidad Única:** Las clases valor como `GeographicPoint`, `StationID`, etc., tienen la única responsabilidad de encapsular datos primitivos y asegurar su validez.
2. **Abierto/Cerrado:** Las interfaces de servicios (`Server`, `QRDecoder`, etc.) están diseñadas para permitir la extensión mediante nuevas implementaciones, sin modificar su estructura original.
3. **Sustitución de Liskov:** Las subclases como `WalletPayment` (herencia de `Payment`) respetan las expectativas de comportamiento definidas en sus superclases.
4. **Segregación de Interfaces:** Se definieron interfaces específicas para los servicios (`UnbondedBTSignal`, `ArduinoMicroController`), minimizando dependencias innecesarias.
5. **Inversión de Dependencias:** Las dependencias de los servicios externos se inyectan mediante métodos setter, permitiendo el uso de dobles de prueba en lugar de implementaciones reales.

Patrones GRASP

1. **Controlador:** `JourneyRealizeHandler` centraliza la gestión de los eventos del caso de uso.
2. **Experto en Información:** Clases como `JourneyService` manejan los cálculos relacionados con los desplazamientos (duración, distancia, velocidad promedio) debido a su conocimiento de los datos necesarios.
3. **Alta Cohesión:** Se evitó agregar responsabilidades innecesarias a las clases, promoviendo una estructura modular y mantenible.
4. **Bajo Acoplamiento:** Las dependencias entre clases se gestionaron mediante interfaces y métodos de inyección, facilitando cambios futuros y pruebas unitarias.

Justificación de las Decisiones Tomadas

- **Clases Valor:** La elección de clases inmutables para `GeographicPoint`, `StationID`, `VehicleID` y `UserAccount` asegura consistencia y evita efectos colaterales en el manejo de datos.
- **Excepciones Personalizadas:** Se definieron excepciones como `InvalidPairingArgsException` y `PMVNotAvailException` para identificar errores específicos y mejorar la legibilidad del código.
- **Uso de Interfaces:** La definición de interfaces para servicios externos (`Server`, `QRDecoder`) permite la simulación de escenarios complejos mediante dobles de prueba.
- **Refactorización:** Durante el desarrollo, se identificaron y corrigieron "code smells" relacionados con la duplicación de código y la complejidad ciclomatica, aplicando técnicas como extracción de métodos y encapsulación.

Cobertura de las Pruebas

1. **Clases Valor:**
    - Pruebas unitarias para validar excepciones al crear instancias con datos inválidos.
2. **Controlador del Caso de Uso:**
    - La clase `JourneyRealizeHandler` fue probada en profundidad con los siguientes escenarios:
        - Emparejar vehículo (`scanQR`): Validación del flujo de éxito y manejo de excepciones como `PMVNotAvailException` y `ProceduralException`.

    - Desemparejar vehículo (`unPairVehicle`): Pruebas de éxito y errores como intento de desemparejar sin emparejamiento previo.
    - Inicio y detención del desplazamiento (`startDriving` y `stopDriving`): Validación de precondiciones y manejo de flujos anómalos.

3. **Servicios Involucrados:**
    - `ServerStub`: Probado para manejar correctamente los métodos de disponibilidad, emparejamiento y desemparejamiento.
    - `QRDecoderStub`: Validado con escenarios exitosos y fallidos (e.g., imágenes corruptas).
    - `UnbondedBTSignal` y `ArduinoMicroController`: Implementados y probados con mocks para garantizar el manejo de sus métodos principales.
4. **Clases del Dominio:**
    - `PMVehicle`: Pruebas completas para cambios de estado (`setAvailb`, `setNotAvailb`, `setUnderWay`) y actualización de ubicación (`setLocation`), incluyendo manejo de excepciones en transiciones no válidas.
    - `JourneyService`: Pruebas para la inicialización (`setServiceInit`) y finalización del servicio (`setServiceFinish`).
5. **Pruebas de Escenarios Completos:**
    - Flujo normal de "Realizar desplazamiento": Emparejamiento, inicio, detención y desemparejamiento.
    - Manejo de errores: Validación de todos los casos de excepción indicados en el enunciado.
6. **Parte Opcional (Pago):**
    - `Wallet`: Pruebas unitarias para la deducción de fondos y manejo de excepciones por saldo insuficiente.
    - `JourneyRealizeHandler`: Validación del flujo de pago mediante monedero, asegurando la correcta integración de los métodos `selectPaymentMethod` y `realizePayment`.

Observaciones Finales

- **Desafíos Encontrados:**
    - Integración de múltiples dependencias externas como servicios Bluetooth y QR decoding.
    - Cobertura de escenarios excepcionales, especialmente en el manejo de estados de vehículos y conexiones interrumpidas.
- **Aspectos Destacados:**
    - Modularidad del sistema: La separación de responsabilidades facilita la extensión y mantenimiento del código.
    - Calidad de pruebas: Los tests diseñados garantizan un alto nivel de confianza en la funcionalidad implementada.

En conclusión, el desarrollo de la práctica ha permitido consolidar los conceptos de diseño y pruebas unitarias en un contexto complejo, asegurando un sistema robusto y extensible para futuras iteraciones.