Enunciado de Práctica
Diseño y Pruebas Unitarias

Temática

El dominio de esta práctica es el mismo que el que habéis estado especificando durante el trabajo de análisis, y que conocéis en detalle: el *Sistema de Micromobilidad Compartida*.

Se pide implementar el código y desarrollar test unitarios de una versión simplificada del caso de uso *Realizar desplazamiento*. Al tratarse de una iteración inicial de la fase de Elaboración, se focaliza en la funcionalidad central, es decir, no incorpora la gestión de las paradas, la función de navegación, ni tampoco el asistente virtual. No incluye tampoco la supervisión de la conducción. Todo ello se irá anexando a la funcionalidad núcleo en sucesivas iteraciones.

Por otro lado, la primera parte de la práctica se centra en el núcleo del caso de uso, es decir, todo el proceso excepto la gestión del pago. La segunda parte, que es opcional, completa la anterior añadiendo la realización del pago, concretamente la modalidad *pago por monedero*.

Se recomienda escribir el código en orden creciente de complejidad, tal y como se propone en este documento, e ir probándolo progresivamente a lo largo del desarrollo.

Comenzaremos formalizando algunas clases consideradas básicas (igual que lo son String, BigDecimal, etc.), dado que su única responsabilidad es la de guardar ciertos valores. Todas ellas se definirán en un paquete denominado data.

El paquete data

El paquete data contendrá algunas clases, la única responsabilidad de las cuales es la de guardar un valor de tipo primitivo o clase de java. Pensad los diversos motivos que hacen que sea conveniente hacerlo así (¡y los *code smell* que evitaréis!)

Se trata de las clases GeographicPoint (un punto geográfico), StationID, VehicleID y UserAccount (identificador de estación de vehículos, identificador de vehículo e identificador de usuario en el sistema, respectivamente).

A continuación se presenta la clase GeographicPoint.

La clase GeographicPoint

Representa una localización geográfica expresada como grados decimales (un número entero decimal), en lugar de hacerlo en grados, minutos y segundos (GMS). Necesitamos un número para la latitud y otro para la longitud.

Esta es su implementación:

```
package data;

/**
* Essential data classes
*/

final public class GeographicPoint {

// The geographical coordinates expressed as decimal degrees

    private final float latitude;
    private final float longitude;

    public GeographicPoint (float lat, float lon) {
        this.latitude = lat;
        this.longitude = lon;
    }

    public float getLatitude () { return latitude; }

    public float getLongitude () { return longitude; }

    @Override
    public boolean equals (Object o) {
        boolean eq;
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        GeographicPoint gP = (GeographicPoint) o;
        eq = ((latitude == gP.latitude) && (longitude == gP.longitude));
        return eq;
    }

    @Override
    public int hashCode () {
        final int prime = 31;
        int result = 1;
        result = prime * result + Float.floatToIntBits(latitude);
        result = prime * result + Float.floatToIntBits(longitude);
        return result;
    }

    @Override
    public String toString () {
        return "Geographic point {" + "latitude='" + latitude + '\'' +
                                    "longitude='" + longitude + '}';
    }
}
```

Estas clases serán **inmutables** (por eso el **final** y la no existencia de setters), y tienen definido un **equals**, que comprueba si dos instancias tienen el mismo valor. Estas clases se denominan **clases valor**, ya que de sus instancias nos interesa tan sólo el valor.

Definid vosotros las otras clases StationID, VehicleID y UserAccount. Podéis escoger vosotros mismos el formato o patrón admitido para estos identificadores (longitud y caracteres admitidos).

Es conveniente añadir las excepciones que consideréis oportunas. Por ejemplo, para el caso de la clase StationID, podemos definir las dos situaciones siguientes: que al constructor le llegue null (objeto sin instanciar), y también un identificador mal formado. ¿Qué excepciones convendrá contemplar para la clase GeographicPoint?

Implementar y realizar test para estas clases (es suficiente con comprobar las excepciones consideradas).

Parte I

El caso de uso *Realizar desplazamiento (8,5 puntos)*

En lo que queda de documento se presenta una versión simplificada del caso de uso a desarrollar: **Realizar desplazamiento**[1]. Se trata de la funcionalidad núcleo del sistema de micromovilidad compartida. Como sabéis, requiere tener activada la conexión Bluetooth para la comunicación de las estaciones de vehículos y los propios vehículos con la app móvil, así como la conexión a internet (wifi o datos) para conectarse con el servidor.

Como ya se ha mencionado, en el desarrollo de esta práctica prescindiremos de las funciones avanzadas del sistema, en concordancia con el escenario trabajado en el DSS (*AnRq-2*). La detección de posibles situaciones de riesgo y el resto de funcionalidades asociadas a los puntos de extensión se dejan para iteraciones más avanzadas de la fase de Elaboración.
Por otro lado, en esta primera parte tan sólo se aborda el núcleo central del caso de uso, dejando la gestión del pago para la segunda parte (*opcional*).

Todo ello se desarrollará en concordancia con los diagramas utilizados como referencia durante el análisis (DCU y *Modelo del Dominio*) y, en particular, con el DSS que se incluye como anexo al final de este documento. Este DSS es una versión más avanzada que el de la solución de referencia, donde ya se presentan algunas componentes y decisiones de diseño y, además, está adaptado al escenario de pago por monedero. Además, disponéis de los contratos de las operaciones para los eventos de entrada al sistema, los cuales deberán cumplirse.
Los casos de uso implicados en esta primera parte de la práctica son, por tanto, los siguientes: **Realizar desplazamiento**, **Emparejar vehículo** y **Desemparejar vehículo**.

Adicionalmente, definiremos la clase JourneyRealizeHandler[2]. Será la clase responsable de manejar los eventos del caso de uso, tanto los que proceden de la interfaz de usuario como del resto de canales de entrada al sistema: canal Bluetooth por defecto y los procedentes del software de control de los microcontroladores de los vehículos. Se explica más adelante.

Además, se implementarán los servicios involucrados. Se presentan a continuación.

---

[1] Como estamos realizando pruebas unitarias, no se presenta ninguna interfaz de usuario, sino que tenemos métodos que representan los eventos de entrada. Tal y como se presenta en el tema de *Patrones GRASP*, implementaremos y probaremos un *controlador de caso de uso*, (un manejador artificial que hará de intermediario entre el sistema y sus canales de entrada).

[2] Una clase artificial que se utiliza específicamente como *controladora del caso de uso* que nos ocupa.

Servicios involucrados

Agruparemos los servicios involucrados en este caso de uso en un paquete denominado services.

Por un lado, tenemos el servicio externo Server. Adicionalmente, otras componentes que actúan como utilidades (hardware y/o software) serán tratadas también como servicios del sistema. Son los siguientes: 1) UnbondedBTSignal; representa la conexión Bluetooth del Smartphone, en particular la conexión BT por defecto, válida para descubrir dispositivos (las estaciones de vehículos en nuestro caso). 2) QRDecoder; representa el software específico para descifrar el código QR de los vehículos. En síntesis, una combinación de visión por computador y algoritmos de decodificación específicos. Y, por último, 3) ArduinoMicroController; representa el software de control de los microcontroladores incrustados en los vehículos.

A continuación, se presentan con más detalle, así como su interacción con el sistema.

Server. Este servicio interviene en tres ocasiones en los pasos de emparejamiento y desemparejamiento, y se vale de los siguientes métodos:

- checkPMVAvail(VehicleID vhID): a partir del identificador del vehículo verifica si éste se encuentra disponible, como paso indispensable para el emparejamiento.
  Excepciones: Además de ConnectException[3], PMVNotAvailException, para indicar que el vehículo se encuentra vinculado con otro usuario.
- registerPairing(UserAccount user, VehicleID veh, StationID st, GeographicPoint loc, LocalDateTime[4] date): se proporciona la información inicial indispensable para vincular una cuenta de usuario con un vehículo (dicho vehículo deja de estar disponible, evitando así que otros usuarios puedan hacer uso de él).
  Excepciones: Además de ConnectException, InvalidPairingArgsException, para indicar que alguno de los argumentos es incorrecto (casos como: el vehículo no se encuentra en dicha estación, la ubicación facilitada no corresponde a ninguna estación, o no es la de estación facilitada, etc.).
- stopPairing(UserAccount user, VehicleID veh, StationID st, GeographicPoint loc, LocalDateTime date, float avSp, float dist, int dur, BigDecimal imp): se proporciona la información completa del servicio, una vez finalizado y realizados los cálculos pertinentes (de tipo float excepto la duración –ésta se proporciona en minutos y es un int- y el importe, un BigDecimal). Dicha información se registra persistentemente en el servidor, así como también la ubicación actual del vehículo, el cual pasa a estado disponible. Todas estas acciones se llevan a cabo en servidor durante el desemparejamiento.

[3] Excepción proporcionada por la API. Señala un error producido al intentar conectar un socket a una dirección y puerto remotos. Por lo general, es debido a que la conexión fue rechazada remotamente (e.g. ningún proceso fue escuchado en la dirección/puerto remoto). Se encuentra en el paquete java.net. La utilizaremos también para las excepciones relacionadas con el uso de Bluetooth.

[4] La clase LocalDateTime, de la API java.time es una alternativa más moderna (java 8) que la clase Date, y soluciona varios de los problemas que esta última presenta. Combina fecha y hora.

Excepciones: Además de ConnectException, InvalidPairingArgsException, para indicar situaciones similares a las mencionadas anteriormente para el método registerPairing().

Operaciones internas:

- setPairing (UserAccount user, VehicleID veh, StationID st, GeographicPoint loc, LocalDateTime date): se guardan persistentemente los datos iniciales del emparejamiento y se actualiza en el servidor el estado del vehículo.
- unPairRegisterService (JourneyService s): se actualiza el registro previo en servidor de la vinculación entre el vehículo y la cuenta de usuario (paso de emparejamiento). Los valores relativos al servicio se encapsulan en s.
  Excepciones: PairingNotFoundException para indicar que no se dispone de la información asociada a dicho emparejamiento en servidor.
- registerLocation (VehicleID veh, StationID st): se actualizan en el servidor el estado y la nueva ubicación del vehículo, una vez finalizado el servicio.

La definición del servicio Server queda tal y como se muestra a continuación:

```
package services;

/**
 * External services involved in the shared micromobility system
*/

public interface Server { // External service for the persistent storage

    // To be invoked by the use case controller

    void checkPMVAvail(VehicleID vhID)
                        throws PMVNotAvailException, ConnectException;

    void registerPairing(UserAccount user, VehicleID veh, StationID st,
                        GeographicPoint loc, LocalDateTime date)
                        throws InvalidPairingArgsException, ConnectException;

    void stopPairing(UserAccount user, VehicleID veh, StationID st,
              GeographicPoint loc, LocalDateTime date, float avSp, float dist,
              int dur, BigDecimal imp)
                        throws InvalidPairingArgsException, ConnectException;

          // Internal operations

    void setPairing(UserAccount user, VehicleID veh, StationID st,
                                      GeographicPoint loc, LocalDateTime date)

    void unPairRegisterService(JourneyService s)
                        throws PairingNotFoundException;

    void registerLocation(VehicleID veh, StationID st);

}
```

Adicionalmente, definiremos interfaces para representar ciertas prestaciones, algunas de las cuales otorgan autonomía al sistema. Aunque son diversas, las que están involucradas en esta iteración son las siguientes: UnbondedBTSignal, QRDecoder y ArduinoMicroController.

Ambas se incorporarán en un subpaquete de services llamado smartfeatures. Veámoslos con detalle:

UnbondedBTSignal. Representa el canal de comunicación Bluetooth que actúa por defecto, sin necesidad de configuración previa, permitiendo la conexión a cualquier dispositivo Bluetooth cercano. Es el utilizado para descubrir las estaciones de vehículos por parte de la app. El canal BT en este caso, a diferencia del utilizado para la comunicación con los vehículos, no está restringido.

La acción de emitir la información sobre el ID de la estación se representa mediante la llamada al método broadcastStationID(), perteneciente a la clase controladora del caso de uso, JourneyRealizeHandler. Será tratado como un evento de entrada al sistema, puesto que al igual que en el caso de la interfaz de usuario, procede del exterior.

El papel de UnbondedBTSignal consistirá exclusivamente en un bucle infinito que incorpora llamadas sucesivas (cada cierto intervalo de tiempo) al método broadcastStationID(). Encapsularemos dicho comportamiento en el siguiente método:

- BTbroadcast (): método encargado de transmitir por BT el ID de la estación de manera continuada e indefinida cada cierto intervalo de tiempo.
  Excepciones: ConnectException.

La definición de UnbondedBTSignal queda tal y como se muestra a continuación:

```
package services.smartfeatures;

/**
 * External services involved in the functioning of some features
*/
public interface UnbondedBTSignal {  // Broadcasts the station ID by BT
       void BTbroadcast () throws ConnectException[5];
}
```

Como situación excepcional a tratar tenemos únicamente la de fallos asociados a la conexión Bluetooth, que trataremos también mediante ConnectException.

QRDecoder. Software específico para descifrar el código QR identificativo de los vehículos (combina visión por computador y algoritmos de decodificación), con el siguiente método:

- getVehicleID (BufferedImage QRImg): recibe la imagen del QR (un BufferedImage) captada por la cámara de fotos del smartphone, lista para ser procesada.
  Excepciones: CorruptedImgException, indica que la imagen captada no es válida.

[5] En el caso de canales de BT, podéis optar, si os interesa, por una excepción más personalizada para la detección de errores de conexión específicos de Bluetooth (utilizando una API de terceros, o bien definiendo una excepción propia).

La definición de la interface QRDecoder queda de este modo:

```
public interface QRDecoder {  // Decodes QR codes from an image

    VehicleID getVehicleID(BufferedImage QRImg) throws CorruptedImgException;
}
```

ArduinoMicroController. Representa el software tipo Arduino [6] de control de los microcontroladores incrustados en los vehículos. Esta tecnología, entre otras cosas, permite a los vehículos de micromobilidad comunicarse con el sistema autónomamente utilizando la conexión BT configurada específicamente con el smartphone. Así, cada vez que el conductor aplica acciones clave sobre el vehículo como, por ejemplo, el inicio y la detención del trayecto, éstas le serán comunicadas al sistema.
Para representar esas acciones clave sobre el vehículo utilizaremos los siguientes métodos:

- startDriving (): representa que el software Arduino detecta que el conductor se ha montado en el vehículo y está siendo manipulado para iniciar un desplazamiento.
  Excepciones: PMVPhisicalException, que indica un problema técnico en el vehículo que impide su puesta en marcha, ConnectException y ProceduralException.
- stopDriving (): representa que el software Arduino detecta que el conductor está frenando de manera continuada hasta la detención del vehículo.
  Excepciones: PMVPhisicalException, ídem, en este caso relacionado con los frenos, ConnectException y ProceduralException.

Adicionamente, dos métodos más para gestionar el canal BT con el Smartphone:

- setBTconnection (): representa la acción de establecer el canal BT entre vehículo y smartphone en el paso de emparejamiento.
  Excepciones: ConnectException.
- undoBTconnection (): acción de deshacer la conexión BT con el smartphone.

La definición de la interface ArduinoMicroController queda de este modo:

```
public interface ArduinoMicroController {  // Software for microcontrollers

  public void setBTconnection () throws ConnectException;

  public void startDriving () throws PMVPhisicalException, ConnectException,
                                     ProceduralException;
  public void stopDriving () throws PMVPhisicalException, ConnectException,
                                    ProceduralException;

  public void undoBTconnection ();
}
```

Obviaremos el paso de envío de un correo electrónico con los detalles de la factura. Es por ello que no se incluye el servicio cliente de correo-e.

Los servicios se inyectarán a la/s clase/s pertinente/s, por ejemplo, mediante un *setter*.

[6] Diseñado para tareas de automatización simples y específicas (controlar sensores, motores y actuadores), ideal para proyectos de bajo consumo energético y control en tiempo real, que interactúan con hardware. Es económico y preciso.

El paquete micromobility

Este paquete contiene las clases directamente involucradas con el uso del sistema de micromobilidad compartida, en colaboración directa con los servicios descritos. Se trata de las clases PMVehicle y JourneyService, que se corresponden con las entidades conceptuales del mismo nombre en el *Modelo del Dominio* proporcionado (*DC-MicromovilidadComp-CoreFunctionality-SolRef.jpg* -funcionalidad núcleo del sistema). Adicionalmente, se define la clase JourneyRealizeHandler, para actuar como controladora del caso de uso.

La clase PMVehicle

A continuación, se presenta la clase PMVehicle. Además de los métodos getter, cuenta con los métodos setter siguientes para la modificación de su estado y su ubicación.

- setNotAvailb (): representa la acción de modificar el estado del vehículo a *NotAvailable*.
- setUnderWay (): representa la acción de modificar el estado del vehículo a *UnderWay*.
- setAvailb (): representa la acción de modificar el estado del vehículo a *Available*.
- setLocation (GeographicPoint gP): para modificar la ubicación del vehículo a la proporcionada como argumento.

El resto de transiciones entre estados (*estacionamiento temporal*) serán obviadas, puesto que no se abordan esas situaciones. Obviaremos también las acciones de (des)bloquear el vehículo y encender o apagar la luz verde.

La estructura, aunque incompleta, de la clase PMVehicle es la siguiente:

```
package micromobility;
/**
* Internal classes involved in in the use of the service
*/
public class PMVehicle {

	???	// The class members
	PMVState	state;
	???	// The constructor/s
	// All the getter methods

	// The setter methods to be used are only the following ones

    public void setNotAvailb () { . . . }

    public void setUnderWay () { . . . }

    public void setAvailb () { . . . }

    public void setLocation (GeographicPoint gP) { . . . }

}
```

Siendo la clase PMVState el enumerado siguiente:

```
public enum PMVState { Availbale, NotAvailable, UnderWay, TemporaryParking;}
```

La clase JourneyService

La estructura, aunque incompleta, de la clase JourneyService es la siguiente:

```
public class JourneyService {

       ???   // The class members, according to the Domain Model
       ???   // The constructor/s
       // All the getter methods
       // Among the setter methods must appear these ones:
    public void setServiceInit () { . . . }
    public void setServiceFinish () { . . . }
}
```

Deberéis seguir las pautas del *Modelo del Dominio* (*DC-MicromovilidadComp-CoreFunctionality-SolRef.jpg*). ***Tened en cuenta*** que las asociaciones entre clases deberán ser implementadas también mediante atributos.
Caso de no implementar la **Parte II** (*opcional*), podéis prescindir del atributo ServiceID (otra clase valor) que representa el identificador del servicio.

La clase controladora del caso de uso JourneyRealizeHandler

Se trata de la clase *controladora del caso de uso* **Realizar desplazamiento**. Se trata de una manejadora artificial de los eventos del caso de uso, y que por tanto se encarga de hacer de intermediario entre la interfaz de usuario y el resto de canales de entrada (canal Bluetooth y el microcontrolador Arduino) y el sistema.

La estructura, aunque incompleta, de la clase JourneyRealizeHandler es la siguiente:

```
public class JourneyRealizeHandler {

      ???   // The class members
      ???   // The constructor/s
   // Different input events that intervene
      // User interface input events
  public void scanQR ()
            throws ConnectException, InvalidPairingArgsException,
             CorruptedImgException, PMVNotAvailException, ProceduralException
                 { . . . }
```

```
  public void unPairVehicle ()
              throws ConnectException, InvalidPairingArgsException,
              PairingNotFoundException, ProceduralException
                   { . . . }

      // Input events from the unbonded Bluetooth channel

  public void broadcastStationID (StationID stID)
              throws ConnectException
                   { . . . }

      // Input events from the Arduino microcontroller channel

  public void startDriving ()
              throws ConnectException, ProceduralException
                   { . . . }

  public void stopDriving ()
              throws ConnectException, ProceduralException
                   { . . . }

       // Internal operations

  private void calculateValues (GeographicPoint gP, LocalDateTime date)
                   { . . . }

  private void calculateImport (float dis, int dur, float avSp,
                                      LocalDateTime date)
                   { . . . }

   (. . .) // Setter methods for injecting dependences
}
```

A continuación, se presentan a grandes rasgos dichos métodos. Los contratos de referencia de todos los eventos de entrada se pueden consultar en el documento *ModeloCasosUso-ParteContratos.pdf*[7].

Eventos de entrada:

- scanQR (): Es un método crucial para el sistema, puesto que es el que da acceso a su funcionalidad principal: *realizar desplazamiento*. Básicamente decodifica el código QR para conocer si el vehículo en cuestión se encuentra disponible y, si todo es correcto, realiza las actualizaciones y preparativos pertinentes, tal y como se detalla en el DSS del anexo. Cabe mencionar que requiere del servicio de QRDecoder y del propio Server.
  Excepciones: ConnectException, InvalidPairingArgsException, CorruptedImgException, PMVNotAvailException y ProceduralException.

[7] Este documento incluye los contratos de los eventos de entrada al sistema, considerando que el sistema engloba la app móvil, los VMP y las estaciones de VMP. Es por ello que, además de los típicos eventos que emulan la interacción del usuario con la interfaz, se incluyen los que emulan la interacción del vehículo con el sistema a través de BT, así como también la recepción de información proveniente de las estaciones de VMP a través del canal BT por defecto (el ID de las mismas).

- unPairVehicle (): en este caso desencadena todas las acciones y actualizaciones necesarias para concluir el servicio y registrar los datos asociados al mismo persistentemente. Requiere también conectarse con el Server.
  Excepciones: ConnectException, InvalidPairingArgsException, PairingNotFoundException y ProceduralException.
- broadcastStationID (StationID stID): evento que emula la acción de recibir por el canal BT del smartphone el ID de la estación.
  Excepciones: ConnectException.
- startDriving (): evento que indica que el usuario inicia su desplazamiento. Se trata de uno de los eventos que le llegan al sistema a través del canal BT preconfigurado entre el vehículo y el smartphone.
  Excepciones: ConnectException y ProceduralException.
- stopDriving (): igual que el anterior, representa un evento procedente del canal BT preconfigurado con el vehículo. En este caso indica que el usuario está deteniendo el vehículo en una estación.
  Excepciones: ConnectException y ProceduralException.

Operaciones internas:

- calculateValues (GeographicPoint gP, LocalDateTime date): calcula los valores relativos al trayecto: duración, distancia y velocidad promedio.
- calculateImport (float dis, int dur, float avSp, LocalDateTime date): calcula el importe correspondiente al servicio a partir de los valores calculados y demás parámetros.

Consideraciones:

- Como veis, **deben tratarse también las situaciones descritas en las precondiciones** (documento *ModeloCasosUso-ParteContratos.pdf*), para detectar si se han completado con éxito los pasos que preceden a cada evento del caso de uso. Manejaréis una única excepción: ProceduralException, para representar todas las situaciones relacionadas con este aspecto, las cuales están detalladas específicamente para cada evento de entrada en el documento mencionado.
- Por lo que respecta a las excepciones correspondientes a las clases del paquete data **quedan para vosotros** (deberán ser incorporadas también en las cabeceras de los métodos).
- Definiréis las excepciones como clases propias (subclases de Exception –excepciones *checables*) y, adicionalmente, la excepción proporcionada por la API: ConnectException, tal y como aparece en las cabeceras de los métodos.

Implementar y realizar test para el caso de uso descrito aquí, utilizando dobles para los servicios colaboradores.

Parte II (*OPCional*)

El Caso de Uso *Realizar pago monedero* (1,5 *puntos*)

Se trata de implementar las clases inmutables, clases adicionales, excepciones, operaciones y eventos de entrada específicos para el caso de uso **Realizar pago monedero**, a fin de resolver esta modalidad de pago. Esta segunda parte es *OPCional*.

En primer lugar, definiréis una nueva clase valor, a incluir también en el paquete data: ServiceID. Es el identificador asignado automáticamente por parte del sistema al servicio, antes de proceder a su almacenamiento persistente en servidor. Igual que para los otros identificadores, podéis escoger vosotros mismos el formato o patrón admitido para estos identificadores (longitud y caracteres admitidos).

Veamos las operaciones que introduce este caso de uso en los servicios y en las clases.

Server. Este servicio introduce una operación más. Es la siguiente:

- registerPayment(ServiceID servID, UserAccount user, BigDecimal imp, char payMeth): registra en servidor los datos asociados al pago, junto con la información acerca del método de pago (un carácter para distinguir entre los cuatro métodos posibles).
  Excepciones: ConnectException.

```
void registerPayment(ServiceID servID, UserAccount user, BigDecimal imp,
                     char payMeth) throws ConnectException;
```

JourneyRealizeHandler. Las incorporaciones a añadir a JourneyRealizeHandler para este caso de uso consisten en un nuevo evento de entrada y una nueva operación interna.

Evento de entrada:

- selectPaymentMethod (char opt): evento para indicar el método de pago escogido por el usuario (un char para distinguir entre los cuatro métodos de pago).
  Excepciones: ProceduralException, ConnectException y NotEnoughWalletException.

Operación interna:

- realizePayment (BigDecimal imp): lleva a cabo el pago deduciendo el valor del importe a la cantidad disponible en el monedero de la app.
  Excepciones: NotEnoughWalletException, para indicar que no se dispone de suficiente dinero en el monedero de la app.

```
    // Specific input event

public void selectPaymentMethod (char opt) throws ProceduralException,
                            NotEnoughWalletException, ConnectException
               { . . . }

    // Specific internal operation

private void realizePayment (BigDecimal imp) throws NotEnoughWalletException
               { . . . }
```

Finalmente, se presentan las clases adicionales relacionadas con el pago, que deberán ser incorporadas.

Clases relacionadas con el **Pago por monedero**

Con el propósito de implementar el caso de uso **Realizar pago monedero**, definiréis:

- La clase Payment, clase base de la jerarquía de pagos admitidos por el sistema. Tened en cuenta que esta clase, tal y como se detalla en el contrato de selectPaymentMethod() (*ModeloCasosUso-ParteContratos.pdf*), mantiene una asociación con el servicio (JourneyService) y el usuario involucrado (UserAcc).
- La clase WalletPayment (subclase de Payment), que representa el pago asociado al servicio mediante el método del monedero, tal y como aparece en el *Modelo del Dominio*. Como veis, esta clase mantiene una asociación con la clase Wallet.
- La clase Wallet, que representa el depósito de dinero almacenado en el monedero de la app. Define un método para aplicar la deducción del importe del servicio. Es el siguiente:

```
private void deduct (BigDecimal imp) throws NotEnoughWalletException
                { . . . }
```

Como excepciones a manejar, a parte de NotEnoughWalletException, las que consideréis oportunas para la clase ServiceID.

Se deja para vosotros la definición de la estructura y métodos de estas clases.

Todas estas clases se definirán en el paquete micromobility.payment.

Implementar y realizar test para las clases Payment. Implementar este caso de uso y realizar test.

Consideraciones generales

- Deberíais resolver esta práctica **en grupos de entre dos y tres personas**.
- Esta práctica tiene un valor de un **25%** sobre la nota final y **no es recuperable**.
- Utilizaréis el sistema de **control de versiones** git y un **repositorio remoto**, a fin de coordinaros con vuestros compañeros (e. g. GitHub o GitLab –podrán ser repositorios privados). Algunas recomendaciones:
  - Cada vez que hagáis un test y el sistema lo pase, haced un commit. Nunca incorporar a la rama remota código que no ha pasado un test.
  - Cada vez que apliquéis un paso de refactoring en el código, haced un commit indicando el motivo que os ha llevado a hacerlo (¿quizás algún *code smell* o principio de diseño?), así como del refactoring aplicado.
  - Con el fin de facilitar los test, podéis definiros diversos constructores para las clases, simplificando con ello su inicialización.
  - Es recomendable ir trabajando cada miembro del grupo en ramas distintas para así lograr una mejor colaboración y sincronización de vuestro trabajo (e.g. desarrollo de distintos requisitos/funcionalidades/ramas por separado).
- Como entregaréis un ZIP con el directorio del proyecto, entregaréis también el repositorio git (subdirectorio .git), por lo que podré comprobar los commits (*no os lo dejéis para el último día, y colaborad todos los miembros del equipo !*).
- Por lo que respecta al SUT (System Under Test), los eventos de entrada **deben satisfacer los contratos** de referencia facilitados (documento *ModeloCasosUso-ParteContratos.pdf* –carpeta *Práctica-Proyecto/Soluciones/Dominio* del CV), los cuales cumplen con el planteamiento expuesto aquí.

- **Nivel de exhaustividad en los test**:
  - Para las clases del paquete data es suficiente con probar las excepciones.
  - **Donde las pruebas deben ser exhaustivas** es en el paquete micromobility. **Deberán tratarse todas las situaciones posibles** de cada escenario, así como todas sus situaciones excepcionales. Para ello **es imprescindible planear una cierta estructura para el código de test**, tanto para las clases de test, como para los test dobles (e.g., podrían definirse por separado los distintos casos de test: los de éxito y los de fracaso).
  - Además, podéis recurrir a la definición de **interfaces de test**, así como la definición de **métodos default**, tal y como se ha sugerido en clase para la resolución de los problemas de la colección.
- Os indico, además, **cómo testear la clase controladora del caso de uso**:
  - Como ya se ha mencionado anteriormente, los eventos de entrada del caso de uso no se testean individualmente. Cada evento es testeado precediéndolo de los eventos que van por delante en el caso de uso. De esta forma se va probando el progreso del caso de uso. En el caso de no seguir el orden establecido, se lanza la excepción ProceduralException, asociada a las **precondiciones** de dichos eventos.
- No hagáis test dobles complicados para poder aprovecharlos más de una vez. Se trata de definir **test dobles lo más simples posible**, con objeto de probar los distintos escenarios. En general, un buen enfoque consiste en definir dobles por separado para diferenciar entre diferentes tipos de situaciones (e.g. de éxito y de fracaso).
  Los test dobles no necesitan testearse. Debéis enfocaros exclusivamente en el SUT. Estos serán testados indirectamente al testear las clases de producción.
- Los test dobles pueden definirse internamente en las clases de test, o bien como clases separadas en paquetes separados.

Entrega

¿Qué debéis entregar?

Un **ZIP** que contenga:

- El **proyecto desarrollado** (podéis utilizar IntelliJ IDEA o cualquier otro entorno).
- Un **informe** en el que expliquéis con vuestras palabras el/los criterios empleados para tomar vuestras decisiones de diseño (principios SOLID, patrones GRASP, etc.), y justificación de las mismas, si es el caso.
  Enumerad también los **métodos de refactoring** aplicados para resolver los posibles **code smell** detectados, ya sea en vuestro código o en el diseño que se propone en este enunciado.
  En cuanto a las situaciones que habéis probado en cada uno de los test realizados, **no es necesario explicarlas**. Tan sólo si hay algún aspecto o situación relevante, que valga la pena mencionar, o bien cualquier otro detalle que pueda ayudar a valorar mejor vuestro trabajo.

Como siempre, haced la entrega a través del CV **tan sólo uno de los miembros del equipo**, indicando el nombre de vuestros compañeros.

Anexo. DSS adaptado al diseño aquí descrito