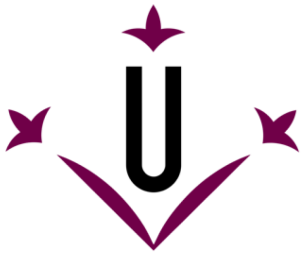

Universitat de Lleida

Diseño y Pruebas Unitarias

Sistema de Micromobilidad Compartida

Boulhani Zanzan, Hamza

Sànchez i Hidalgo, Carles

Serrano Ortega, Aniol

Fecha: 29/12/2024

Practica 3

Enginyeria del Programari

Escola Politècnica Superior

Indice

Indice de Figuras

1. Introduccion

En el contexto actual de la movilidad urbana, los sistemas de micro-movilidad compartida juegan un papel crucial en la optimización del transporte sostenible. Este informe se centra en el desarrollo y las pruebas unitarias de una versión simplificada del caso de uso "Realizar desplazamiento" dentro del Sistema de Micro-movilidad Compartida en el marco de la asignatura Ingeniería de Software. El objetivo principal de este proyecto es implementar la funcionalidad central que permite a los usuarios iniciar y finalizar desplazamientos utilizando vehículos compartidos, sin incorporar en esta fase aspectos como la gestión de paradas, la navegación o asistentes virtuales, los cuales se abordarían en iteraciones posteriores si se llegase a implementar este proyecto ya fuera del marco de la asignatura.

El desarrollo de este sistema se ha llevado a cabo siguiendo una metodología que prioriza la calidad del código y la mantenibilidad del sistema. Para ello, se han aplicado principios de diseño orientado a objetos, destacando los principios *SOLID* y los patrones *GRASP*. Estos principios han guiado la asignación de responsabilidades entre las clases y la estructuración del código, facilitando así una arquitectura robusta y escalable. Además, se ha prestado especial atención a la identificación y eliminación de *code smells*, lo que ha permitido mantener un código limpio y fácil de entender, reduciendo la complejidad y mejorando la eficiencia en el desarrollo y las pruebas.

El proyecto está organizado en varios paquetes, destacando el paquete data, que contiene clases inmutables responsables de almacenar valores esenciales como puntos geográficos, identificadores de estaciones, vehículos y usuarios. Esta separación de responsabilidades no solo sigue el Principio de Responsabilidad Única (SRP) de *SOLID*, sino que también ayuda a evitar la duplicación de código y facilita las pruebas unitarias.

En la sección de Diseño, se detalla la arquitectura del sistema, explicando cómo se han aplicado los patrones *GRASP* para gestionar las interacciones entre los diferentes componentes del sistema y asegurar un bajo acoplamiento y alta cohesión. Asimismo, se describen las estrategias empleadas para detectar y resolver posibles *code smells*, como clases con múltiples responsabilidades o métodos excesivamente complejos, lo que ha contribuido a mejorar la calidad general del código.

La sección de *Tests* aborda la estrategia de verificación implementada, enfocándose en pruebas unitarias que aseguran el correcto funcionamiento de cada componente del sistema. La aplicación de los principios *SOLID* ha facilitado la creación de pruebas más efectivas y eficientes, permitiendo una mayor cobertura del código y una detección temprana de errores.

Finalmente, en la *Conclusión*, se reflexiona sobre los logros alcanzados durante el desarrollo del proyecto, destacando cómo la adherencia a los principios de diseño y la gestión proactiva de code smells han resultado en un sistema robusto y mantenible. Se ofrecen también recomendaciones para futuras mejoras y se subraya la importancia de continuar aplicando buenas prácticas de diseño en las próximas iteraciones del proyecto

2. Diseño

Para empezar el desarrollo de la aplicación se ha modularizado cada apartado de la aplicación en distintos paquetes. Así mismo, estos paquetes se encuentran tanto en el directorio `src` como en el directorio `test`, en el primero se almacenan las implementaciones y en el segundo los tests de esas implementaciones. De este modo, el diseño de paquetes es el siguiente:

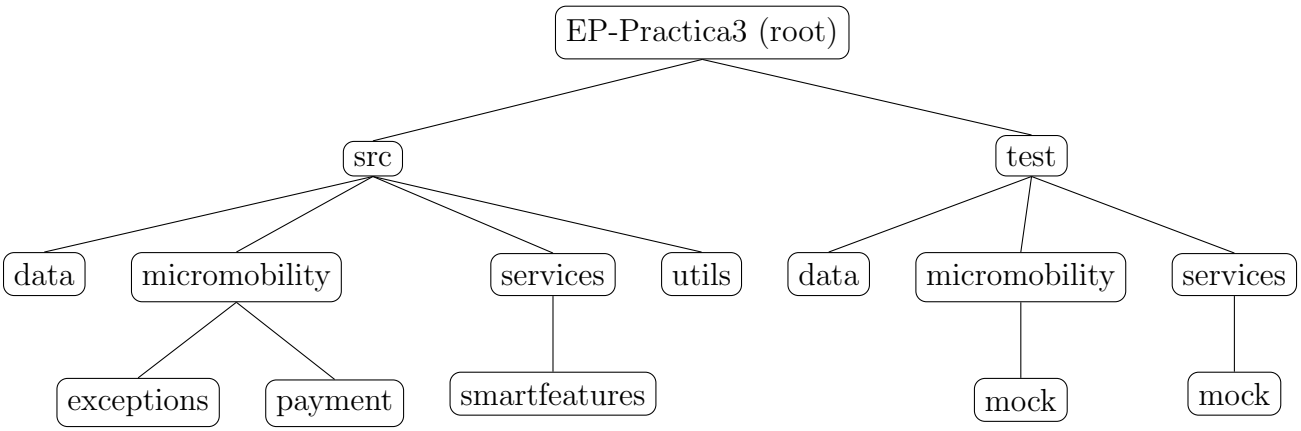

Figura 1: Diagrama estructural del proyecto

1. Arquitectura General

La arquitectura del sistema se ha diseñado siguiendo una estructura modular que facilita la mantenibilidad y escalabilidad del proyecto. Las principales divisiones identificadas son los paquetes `data`, `services`, `micromobility` y, opcionalmente, `micromobility.payment`. Esta organización permite una clara separación de responsabilidades, alineándose con los principios SOLID y los patrones GRASP para asignar adecuadamente las responsabilidades entre las clases.

2. Descripción de los Paquetes Principales

2.1. Paquete `data`

El paquete `data` alberga clases valor que representan entidades fundamentales del dominio, tales como `GeographicPoint`, `StationID`, `VehicleID` y `UserAccount`. Estas clases son

inmutables, lo que significa que una vez creadas, sus estados no pueden ser modificados. Esta inmutabilidad contribuye a la seguridad y consistencia de los datos a lo largo del sistema, cumpliendo con el *Principio de Responsabilidad Única (SRP)* de SOLID al tener cada clase una única responsabilidad de almacenar y gestionar un tipo específico de dato.

Por ejemplo, la clase `GeographicPoint` encapsula las coordenadas geográficas de un punto específico, asegurando que los valores de latitud y longitud sean consistentes mediante la ausencia de métodos setters y la implementación adecuada de los métodos `equals`, `hashCode` y `toString`. De manera similar, las clases `StationID`, `VehicleID` y `UserAccount` validan sus identificadores en el constructor, garantizando que cada instancia cumpla con el formato esperado y evitando así posibles *code smells* como **clases duplicadas** o **validaciones dispersas** en múltiples lugares del código.

Listing 1: Ejemplo de Clase GeographicPoint

```
1 package data;
2 final public class GeographicPoint {
3     private final float latitude;
4     private final float longitude;
5
6     public GeographicPoint(float lat, float lon) {
7         this.latitude = lat;
8         this.longitude = lon;
9     }
10
11    public float getLatitude() {
12        return latitude;
13    }
14
15    public float getLongitude() {
16        return longitude;
17    }
```

```
18
19      @Override
20      public boolean equals(Object o) {
21          boolean eq;
22          if (this == o) return true;
23          if (o == null || getClass() != o.getClass()) return
                false;
24          GeographicPoint gP = (GeographicPoint) o;
25          eq = ((latitude == gP.latitude) && (longitude == gP.
                longitude));
26          return eq;
27      }
28
29      @Override
30      public int hashCode() {
31          final int prime = 31;
32          int result = 1;
33          result = prime * result + Float.floatToIntBits(latitude
                );
34          result = prime * result + Float.floatToIntBits(
                longitude);
35          return result;
36      }
37
38      @Override
39      public String toString() {
40          return "Geographic point: {latitude = " + latitude + "
                longitude = " + longitude + "}";
41      }
42  }
```

2.2. Paquete `services`

El paquete `services` engloba las interfaces y clases que interactúan con servicios externos e internos del sistema. Entre los servicios definidos se encuentran:

- `Server`: Interface que representa el servicio externo de almacenamiento persistente, definiendo métodos como `checkPMVAvail`, `registerPairing` y `stopPairing`.
- `UnbondedBTSignal`: Interface que maneja la comunicación Bluetooth para descubrir estaciones de vehículos mediante el método `BTbroadcast`.
- `QRDecoder`: Interface para decodificar códigos QR de los vehículos a través del método `getVehicleID`.
- `ArduinoMicroController`: Interface que gestiona la comunicación con los microcontroladores incrustados en los vehículos, incluyendo métodos como `startDriving` y `stopDriving`.

La utilización de interfaces en este paquete permite una fácil extensión de funcionalidades sin necesidad de modificar las clases existentes, adheriéndose así al *Principio de Abierto/Cerrado (OCP)*. Además, la inyección de dependencias en clases como `JourneyRealizeHandler` asegura un bajo acoplamiento entre componentes, facilitando la sustitución de implementaciones concretas por otras que cumplan con las mismas interfaces, lo cual es una aplicación directa del *Principio de Inversión de Dependencias (DIP)*.

2.3. Paquete `micromobility`

Este paquete contiene las clases directamente involucradas en la funcionalidad central del sistema de micromovilidad compartida, como `PMVehicle` y `JourneyService`. Además, incluye la clase controladora `JourneyRealizeHandler`, responsable de manejar los eventos del caso de uso "Realizar desplazamiento".

La clase `PMVehicle` gestiona el estado y la ubicación de cada vehículo, implementando métodos como `setNotAvailb`, `setUnderWay` y `setAvailb`. Estos métodos aseguran que las transiciones de estado se realicen de manera controlada, lanzando excepciones cuando las

precondiciones no se cumplen, lo que previene *code smells* como **métodos con múltiples responsabilidades** o **lógica de negocio dispersa**.

Por otro lado, la clase `JourneyService` encapsula toda la información relacionada con un desplazamiento, incluyendo datos como la ubicación de inicio y fin, duración, distancia y costo del servicio. La creación de instancias de `JourneyService` se realiza a través de una fábrica (`JourneyServiceFactory`), lo que facilita la adherencia al patrón GRASP *Information Expert*, asignando la responsabilidad de creación a una clase especializada.

La clase `JourneyRealizeHandler` actúa como controlador (*GRASP Controller*), gestionando las interacciones entre la interfaz de usuario y los servicios del sistema. Esta clase coordina las operaciones necesarias para iniciar y finalizar un desplazamiento, asegurando que las dependencias se inyecten correctamente y que las interacciones con otros componentes se realicen de manera ordenada y coherente. Además, al manejar excepciones específicas, se mejora la robustez del sistema y se facilita el mantenimiento del código.

3. Aplicación de Principios SOLID

Durante el diseño del sistema, se han aplicado rigurosamente los principios SOLID para asegurar una arquitectura robusta y mantenible. El *Principio de Responsabilidad Única (SRP)* se refleja en la clara separación de responsabilidades entre los diferentes paquetes y clases, como se observa en las clases del paquete `data` que únicamente almacenan y gestionan datos. El *Principio de Abierto/Cerrado (OCP)* se ha implementado mediante el uso de interfaces en el paquete `services`, permitiendo la extensión de funcionalidades sin necesidad de modificar las clases existentes.

El *Principio de Sustitución de Liskov (LSP)* asegura que las subclases o implementaciones de interfaces pueden sustituir a sus superclases sin afectar el funcionamiento del sistema, lo cual es fundamental para mantener la integridad del sistema cuando se añaden nuevas funcionalidades. El *Principio de Segregación de Interfaces (ISP)* se ha aplicado creando interfaces específicas para cada servicio, evitando interfaces monolíticas que obliguen a implementar métodos innecesarios y promoviendo una mayor cohesión dentro de las clases.

Finalmente, el *Principio de Inversión de Dependencias (DIP)* se ha logrado mediante la inyección de dependencias en las clases controladoras como `JourneyRealizeHandler`, per-

mitiendo que las clases dependan de abstracciones (interfaces) en lugar de implementaciones concretas. Esto facilita la reutilización de código y mejora la modularidad del sistema, permitiendo una mayor flexibilidad y facilidad para realizar pruebas unitarias.

4. Incorporación de Patrones GRASP

Los patrones GRASP han sido fundamentales para asignar responsabilidades de manera efectiva entre las clases del sistema. La clase `JourneyRealizeHandler` actúa como controlador (*GRASP Controller*), gestionando las interacciones entre la interfaz de usuario y los servicios del sistema. Este patrón facilita la separación de la lógica de negocio de la interfaz de usuario, mejorando la mantenibilidad y la claridad del código.

Además, se ha aplicado el patrón *Information Expert* al asignar la responsabilidad de gestionar la información necesaria a las clases que poseen dicha información. Por ejemplo, la clase `PMVehicle` es responsable de gestionar su propio estado y ubicación, mientras que `JourneyService` maneja los datos relacionados con un desplazamiento específico.

El patrón *Low Coupling y High Cohesion* se ha implementado manteniendo un bajo acoplamiento entre las clases y asegurando que cada clase tenga una alta cohesión interna. Esto se logra mediante la utilización de interfaces y la inyección de dependencias, lo que facilita la reutilización de código y mejora la modularidad del sistema.

5. Gestión de Code Smells

Durante el desarrollo, se ha prestado especial atención a la identificación y eliminación de *code smells* para mantener un código limpio y de alta calidad. Se han aplicado varias estrategias para abordar estos problemas:

Clases God: Se ha evitado la creación de clases con múltiples responsabilidades distribuyendo las funcionalidades entre clases especializadas. Por ejemplo, la clase `JourneyRealizeHandler` se encarga exclusivamente de gestionar los eventos del caso de uso "Realizar desplazamiento", mientras que las clases en el paquete `data` gestionan únicamente los datos.

Duplicación de Código: Se ha eliminado la duplicación de código mediante la reutilización de métodos y clases comunes. Las clases valor en el paquete `data` evitan la repetición

de lógica de validación y gestión de datos en diferentes partes del sistema.

Métodos Largos y Complejos: Se han refactorizado métodos excesivamente largos o complejos dividiéndolos en métodos más pequeños y manejables que cumplen con una única tarea. Por ejemplo, en la clase `JourneyRealizeHandler`, las operaciones de inicio y fin de un desplazamiento se manejan en métodos específicos que encapsulan la lógica necesaria, mejorando la legibilidad y facilitando el mantenimiento.

Nombres Poco Claros: Se han utilizado nombres descriptivos y coherentes para clases, métodos y variables, mejorando la legibilidad del código. Nombres como `checkPMVAvail`, `registerPairing` y `stopPairing` son claros y reflejan su funcionalidad, lo que facilita la comprensión del código por parte de otros desarrolladores.

Comentarios Innecesarios: Se ha buscado que el código sea autoexplicativo, reduciendo la necesidad de comentarios excesivos y favoreciendo la claridad en la implementación. La estructura modular y la adherencia a los principios SOLID y patrones GRASP contribuyen a que el código sea fácil de entender sin depender de comentarios detallados.

Estas prácticas han contribuido significativamente a mantener un código limpio, fácil de mantener y escalable, asegurando la calidad del sistema de micromovilidad compartida.

3. Tests

Para garantizar la correcta funcionalidad y robustez del sistema de micromovilidad compartida, se ha implementado una suite completa de pruebas unitarias. Estas pruebas verifican el comportamiento de cada componente de manera aislada, asegurando que cumplan con los requisitos especificados y manejen adecuadamente distintos escenarios, incluyendo casos de éxito y excepcionales.

1. Importancia de los Tests Dobles

En el desarrollo de software, los *tests dobles* son técnicas esenciales que permiten simular el comportamiento de dependencias externas, facilitando la prueba de componentes individuales sin la necesidad de interactuar con servicios reales o recursos complejos. Los *tests dobles* incluyen **mocks**, **stubs** y **fakes**, cada uno con un propósito específico:

- **Mocks**: Simulan objetos complejos y permiten verificar interacciones específicas entre componentes, como llamadas a métodos y pasaje de parámetros.
- **Stubs**: Proporcionan respuestas predefinidas a llamadas de métodos, sin verificar interacciones, lo que es útil para simular comportamientos simples.
- **Fakes**: Implementan funcionalidades simplificadas que imitan el comportamiento real, pero son más fáciles de configurar y utilizan, permitiendo pruebas más realistas sin la complejidad completa del sistema real.

La utilización de *tests dobles* es crucial por varias razones:

- **Aislamiento de Módulos**: Permite probar cada componente de manera independiente, asegurando que funcione correctamente sin la interferencia de otros módulos.
- **Reducción de Dependencias Externas**: Evita la necesidad de interactuar con servicios reales como servidores remotos o hardware físico durante las pruebas, lo que puede ser costoso, lento o poco práctico.

- **Mejora de la Velocidad de las Pruebas**: Los componentes simulados suelen ser más rápidos y predecibles, agilizando el proceso de pruebas y permitiendo una ejecución más frecuente.
- **Facilitación de la Identificación de Errores**: Al aislar los componentes, es más sencillo identificar la fuente de un error específico, mejorando la eficiencia en el diagnóstico y corrección de problemas.

2. Estrategia de Pruebas Empleada

La estrategia de pruebas adoptada en este proyecto combina diferentes tipos de pruebas para asegurar una cobertura amplia y efectiva de las funcionalidades del sistema:

- **Pruebas Unitarias**: Verifican el correcto funcionamiento de clases y métodos individuales, asegurando que cada unidad de código cumple con su propósito específico.
- **Pruebas de Integración**: Evalúan la interacción entre diferentes módulos y componentes del sistema, garantizando que trabajen juntos de manera coherente y eficiente.
- **Pruebas de Regresión**: Aseguran que las nuevas implementaciones no afecten negativamente las funcionalidades existentes, manteniendo la estabilidad del sistema a lo largo del tiempo.

Para implementar estas pruebas, se han utilizado *tests dobles* que simulan las dependencias externas, permitiendo así una evaluación más precisa y eficiente de cada componente. Esta estrategia ha facilitado la detección temprana de errores y ha contribuido significativamente a la calidad general del sistema.

3. Implementación de Tests Dobles

En este proyecto, se han desarrollado varios *tests dobles* personalizados para simular diferentes aspectos del sistema. A continuación, se describen algunos de los más relevantes:

- **Mocks de Servicios Externos**: Se han creado mocks para simular la interacción con servidores externos, decodificadores de QR y controladores de hardware. Estos mocks

permiten controlar las respuestas y comportamientos de estos servicios, facilitando la prueba de diferentes escenarios sin depender de los servicios reales.

- **Mocks de Controladores**: Los controladores que gestionan la lógica de negocio interactúan con varios servicios y componentes. Utilizando mocks, es posible verificar que estos controladores manejan correctamente las respuestas de los servicios y gestionan adecuadamente los estados internos.

- **Mocks de Hardware**: Para simular interacciones con hardware físico, como controladores Arduino o señales Bluetooth, se han implementado mocks que replican las funcionalidades necesarias, permitiendo probar la lógica sin necesidad de los dispositivos reales.

Estos *tests dobles* permiten una mayor flexibilidad y control sobre los escenarios de prueba, asegurando que se puedan cubrir tanto casos de uso comunes como situaciones excepcionales de manera eficiente.

4. Ejemplos de Pruebas Implementadas

A continuación, se describen algunos ejemplos representativos de pruebas unitarias que utilizan *tests dobles* para verificar el comportamiento del sistema:

4.1. Prueba de Escaneo de QR con Éxito

Esta prueba verifica que el proceso de escanear un QR y iniciar un viaje funcione correctamente. Utiliza mocks para simular la decodificación del QR y la creación del servicio de viaje, asegurando que las interacciones entre los componentes se realicen como se espera.

4.2. Prueba de Finalización de Viaje

Esta prueba verifica que al finalizar un viaje, se llamen correctamente los métodos del servicio de viaje y se calculen los costos asociados. Utiliza mocks para simular la finalización del viaje y el cálculo de costos, garantizando que el sistema maneje adecuadamente estos procesos.

4.3. Prueba de Manejo de Excepciones en el Escaneo de QR

Esta prueba asegura que el sistema maneje adecuadamente una situación en la que el vehículo no está disponible al escanear un QR. Utiliza mocks para simular la indisponibilidad del vehículo y verifica que se lance la excepción correspondiente, evitando la creación de un viaje activo.

5. Cobertura de Pruebas

Las pruebas implementadas cubren una amplia gama de escenarios, incluyendo:

- **Creación y Validación de Entidades**: Verifica que las clases `GeographicPoint`, `StationID`, `UserAccount` y `VehicleID` se crean correctamente, manejan la igualdad y generan los *hash codes* adecuados.
- **Interacciones entre Componentes**: Asegura que el controlador `JourneyRealizeHandler` interactúa correctamente con los servicios de viaje, decodificador de QR y otros componentes simulados.
- **Manejo de Excepciones**: Verifica que el sistema maneja adecuadamente situaciones excepcionales, como QR corruptos o vehículos no disponibles.
- **Estados del Vehículo**: Comprueba que el vehículo `PMVehicle` cambia de estado de manera correcta y que las transiciones inválidas son adecuadamente rechazadas.
- **Simulación de Interacciones de Hardware**: Utiliza mocks como `ArduinoMicroControllerMock` para simular interacciones con hardware físico, permitiendo verificar el comportamiento del sistema sin depender de dispositivos reales.

Esta cobertura asegura que tanto las funcionalidades principales como los comportamientos excepcionales del sistema han sido rigurosamente evaluados, contribuyendo a la robustez y fiabilidad del sistema de micromovilidad compartida.

6. Conclusión

La implementación de una suite robusta de pruebas unitarias, apoyada en el uso de *tests dobles*, ha permitido asegurar la calidad y fiabilidad del sistema de micromovilidad compartida. Estas pruebas no solo verifican el correcto funcionamiento de cada componente de manera aislada, sino que también garantizan que las interacciones entre componentes se realicen de acuerdo con lo esperado. Además, la capacidad de simular escenarios excepcionales mediante mocks ha facilitado la identificación y manejo de errores, contribuyendo a la robustez general del sistema.

La estrategia de pruebas adoptada, que combina pruebas unitarias, de integración y de regresión, junto con el uso efectivo de *tests dobles*, proporciona una base sólida para el mantenimiento y escalabilidad futura del sistema.

7. Workflow

Para garantizar el buen funcionamiento del proyecto se ha integrado un *pipeline* de GitHub llamado *workflow*. Estos *workflows* se definen en el directorio `.github/workflows` y sirven para automatizar procesos mediante la API de GitHub.

Mediante este *workflow* se compilan los archivos de java, se comprueba el estilo del código mediante un linter[1], y se ejecutan todos los tests. En caso que todos los tests se ejecuten de forma existosa, el *workflow* funcionara correctamente.

[1]Un linter es una herramienta que analiza el código fuente para identificar errores de estilo, convenciones o posibles problemas que podrían afectar la calidad del código.

4. Conclusiones

El desarrollo del sistema de micromovilidad compartida ha alcanzado con éxito los objetivos planteados, implementando de manera efectiva la funcionalidad central de realizar desplazamientos. La estructuración modular del código, organizada en paquetes bien definidos, ha facilitado tanto el desarrollo como el mantenimiento del sistema, garantizando una arquitectura robusta y escalable.

La aplicación de los principios SOLID y los patrones GRASP ha sido fundamental para asegurar una distribución adecuada de responsabilidades entre las clases. Estos principios han permitido diseñar un sistema con bajo acoplamiento y alta cohesión, lo que no solo mejora la mantenibilidad del código sino que también facilita su evolución futura. La atención dedicada a la identificación y eliminación de *code smells* ha contribuido a mantener un código limpio y libre de redundancias, reduciendo la complejidad y mejorando la eficiencia en el desarrollo.

La estrategia de pruebas adoptada, centrada en pruebas unitarias y el uso de *tests dobles*, ha sido crucial para garantizar la fiabilidad y robustez del sistema. Los *tests dobles* han permitido simular las interacciones con dependencias externas, facilitando la verificación de cada componente de manera aislada. Esta metodología ha incrementado la cobertura de las pruebas, permitiendo detectar y corregir errores en etapas tempranas del desarrollo, lo que se traduce en un sistema más estable y confiable.

En conclusión, este proyecto ha establecido una base sólida para el desarrollo de pruebas unitarias, combinando buenas prácticas de diseño y estrategias de pruebas que garantizan la calidad, mantenibilidad y escalabilidad del sistema. La adherencia a principios de diseño orientado a objetos ha sido clave para lograr un sistema robusto y confiable.