Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Sistema de trazabilidad de versiones de contenido en flotas embarcadas de transporte inteligente

Autor: Diego Valladolid Clemente

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Sistema de trazabilidad de versiones de contenido en flotas embarcadas de transporte inteligente

Autor: Diego Valladolid Clemente

Tutor: Valentín Cardeñoso Payo

Tutor de Empresa: Álvaro Gamarra Martín

A mi madre, que me ha criado sola y me ha apoyado incondicionalmente.

Gracias por ser mi pilar y apoyarme en mi camino.

Agradecimientos

Quiero expresar mi más profundo agradecimiento a todas las personas que han hecho posible la realización de este proyecto.

En primer lugar, agradezco a mis tutores, tanto de la empresa en la que se ha realizado este proyecto como el encargado por parte de la universidad, que han compartido conmigo su experiencia, conocimiento y paciencia.

También deseo reconocer el apoyo incondicional de mis compañeros de trabajo en GMV. Su profesionalidad, colaboración y entusiasmo han contribuido a crear un ambiente de trabajo estimulante y a impulsar el desarrollo de este proyecto. La sinergia y el compromiso que demuestran día a día han sido una fuente constante de inspiración y ayuda.

Del mismo modo, quiero agradecer a mi grupo de amigos, quienes han estado a mi lado en los momentos buenos y en los difíciles. Su cercanía, alegría y apoyo constante me han dado fuerza para continuar, recordándome siempre la importancia de compartir el camino con quienes te hacen sentir en casa. Gracias por las risas, los consejos y por estar ahí siempre.

Finalmente, no podría dejar de agradecer a mi familia, especialmente a mi madre, cuyo amor, sacrificio y apoyo inquebrantable me han impulsado a seguir adelante en cada etapa de mi formación. Su ejemplo y fortaleza han sido mi mayor motivación.

A todos vosotros, muchas gracias por vuestra confianza y por acompañarme en este recorrido.

Resumen

En los sistemas de transporte inteligente, el despliegue correcto de contenidos como audios, anuncios, paradas, líneas, etc, a los equipos embarcados (OBU) de los buses, es esencial para garantizar el funcionamiento coordinado y actualizado de toda la flota. Actualmente, las revisiones de versiones instaladas en los vehículos se realizan de forma manual, lo cual implica un alto consumo de tiempo, una fuerte dependencia de comprobaciones individuales y una elevada probabilidad de error humano.

Este Trabajo Fin de Grado surge con el objetivo de automatizar el proceso de verificación de actualizaciones, permitiendo identificar de manera precisa qué vehículos han recibido correctamente los contenidos y cuáles no, antes de autorizar un despliegue operativo. De esta forma, se mejora significativamente la trazabilidad y se reduce el riesgo de que un vehículo entre en servicio con versiones obsoletas o inconsistentes.

La solución propuesta se desarrolla sobre una arquitectura distribuida ya existente, en la que participan múltiples componentes de un sistema ya establecido, siendo algunos ejemplo, un gestor de contenidos, conocido como sistema de información al usuario (*SIU*), un generador de archivos (*ArchivosOBU*) y un sistema de transferencia (*Transfer Manager*). El sistema implementado realiza una comparación automatizada entre las versiones esperadas (generadas tras un cambio de contenido) y las versiones reales detectadas en cada equipo, a partir de unos archivos que se van generando a lo largo de todo el workflow de transferencia.

Adicionalmente, este TFG también aborda uno de los desafíos comunes en Ingeniería del Software: la integración de nuevas funcionalidades en sistemas complejos y maduros ya desplegados. En este contexto, se estudian distintas alternativas de arquitectura, se analizan sus implicaciones técnicas y se justifica la elección final de la solución propuesta, considerando factores como la mantenibilidad, escalabilidad y la evolución futura del sistema.

El trabajo incluye el análisis del sistema actual, el diseño e implementación de la lógica de verificación, el modelado de datos asociado, y el desarrollo de una interfaz de usuario orientada a operadores técnicos para la consulta del estado de actualización por vehículo.

El resultado final es una herramienta de trazabilidad que expone esta información a través de una interfaz técnica, accesible por operadores, y que permite controlar de forma visual y automatizada el estado de sincronización de cada vehículo. Este trabajo abarca el análisis del problema actual, el diseño de la solución software, la implementación de la lógica de verificación y la propuesta de una interfaz funcional de consulta orientada a entornos reales de operación.

Abstract

In intelligent transportation systems, the correct deployment of content to on-board units (OBUs) is essential to ensure the coordinated and up-to-date operation of the entire fleet. Currently, the verification of content versions installed on vehicles is performed manually, which implies a high operational cost, a strong dependency on individual checks, and an increased risk of human error.

This Bachelor's Thesis aims to automate the update verification process, enabling precise identification of which vehicles have successfully received the updated content before authorizing them for operational deployment. This significantly improves system traceability and reduces the risk of vehicles entering service with outdated or inconsistent versions.

The proposed solution is built upon an existing distributed architecture composed of several components, such as the content management system (SIU), a file generator (ArchivosOBU), and the transfer system (Transfer Manager). The implemented system performs an automated comparison between the expected versions (generated after a content change) and the actual versions detected on each OBU, based on a set of files produced throughout the update workflow.

Additionally, this project addresses a common challenge in Software Engineering: integrating new functionality into complex, mature, and already deployed systems. Various architectural alternatives are analyzed, and the chosen solution is justified based on key factors such as maintainability, scalability, and future system evolution.

The work includes the analysis of the current system, the design and implementation of the version verification logic, data modeling, and the development of a technical interface for operators to monitor the update status of each vehicle. The final result is a traceability tool that provides a clear and automated view of the synchronization status of the fleet, adapted to the requirements of real operational environments.

Índice general

Índice de cuadros v

Índice de figuras vii

I Objeto, Concepto y Método 1

1. Introducción 3
1.1. Introducción . . . 3
1.2. Motivación . . . 4

2. Objetivos y Alcance 5
2.1. Objetivos . . . 5
2.1.1. Objetivos generales . . . 5
2.1.2. Objetivos específicos . . . 5
2.1.3. Objetivos personales . . . 5
2.1.4. Tareas a realizar . . . 5
2.2. Alcance . . . 6

3. Metodología 7
3.1. Enfoque de desarrollo . . . 7
3.1.1. ¿Qué es Scrum? . . . 7
3.1.2. Pilares fundamentales de Scrum . . . 7
3.1.3. Componentes de Scrum . . . 8
3.1.4. Aplicación al Proyecto . . . 10
3.2. Planificación . . . 10
3.2.1. Sprint 0 - 12/03/2025 - 26/03/2025 . . . 10
3.2.2. Sprint 1 - 26/03/2025 - 08/04/2025 . . . 11
3.2.3. Sprint 2 - 9/04/2025 - 22/04/2025 . . . 11
3.2.4. Sprint 3 - 23/04/2025 - 06/05/2025 . . . 11
3.2.5. Sprint 4 - 07/05/2025 - 20/05/2025 . . . 12
3.2.6. Sprint 5 - 21/05/2025 - 03/06/2025 . . . 12
3.2.7. Sprint 6 - 04/06/2025 - 06/07/2025 . . . 13
3.2.8. Plan de control y Riesgos . . . 13
3.3. Costes . . . 16
3.3.1. Coste humano . . . 16
3.3.2. Costes de Hardware . . . 16
3.3.3. Costes de Software . . . 16
3.3.4. Presupuesto Total . . . 16

II Marco Conceptual y Contexto **17**

4. Marco Contextual **19**
4.1. Entorno Profesional . . . 19
4.2. Contexto operativo del proyecto . . . 19
4.3. Problemática detectada . . . 20
4.4. Justificación del proyecto . . . 20

5. Marco Conceptual y Tecnológico **21**
5.1. Arquitecturas distribuidas . . . 21
5.2. Control de versiones en sistemas software . . . 21
5.3. Transferencia de datos . . . 22
5.4. Sistemas de transporte inteligente . . . 22
5.5. Tecnologías utilizadas . . . 23
5.5.1. Backend y lógica de negocio . . . 23
5.5.2. Frontend . . . 23
5.5.3. Modelado y diseño de sistema . . . 23
5.5.4. Control de versiones y gestión de tareas . . . 24
5.5.5. Pruebas, documentación y validación . . . 24
5.5.6. Conclusión . . . 24

6. Soluciones y Estado del Arte **25**
6.1. Introducción . . . 25
6.2. Soluciones en el ámbito de desarrollo software . . . 25
6.3. Soluciones en el sector Transporte . . . 25
6.4. Alternativas internas en GMV . . . 26
6.5. Justificación de la solución . . . 26

III Desarrollo del Sistema **27**

7. Análisis **29**
7.1. Flujo actual del sistema . . . 29
7.2. Identificación de necesidades . . . 30
7.3. Integración en sistema complejo existente . . . 31
7.3.1. Características del sistema a tener en cuenta: . . . 31
7.3.2. Equilibrio entre integración y viabilidad: . . . 32
7.4. Requisitos . . . 33
7.4.1. Requisitos funcionales . . . 33
7.4.2. Requisitos no funcionales . . . 33

8. Diseño **39**
8.1. Alternativas de arquitectura evaluadas . . . 39
8.1.1. Microservicio Independiente . . . 39
8.1.2. Módulo integrado en backend existente . . . 40
8.1.3. Módulo reutilizable integrado con separación por capas . . . 40
8.1.4. Arquitectura elegida . . . 40
8.2. Diseño . . . 41
8.3. Patrones de Diseño aplicados . . . 41
8.3.1. Singleton . . . 42
8.3.2. Fachada . . . 42

8.3.3. Inyección de dependencias . . . 42
8.3.4. Strategy . . . 43
8.3.5. Scheduled Task . . . 44
8.3.6. Template . . . 44
8.4. Modelado de datos . . . 45
8.5. Diseño de InfoVersionService . . . 47
8.5.1. Descripción General . . . 47
8.5.2. Arquitectura del Microservicio . . . 48
8.5.3. Dependencias entre submódulos . . . 50
8.5.4. Diagrama de clases entre capas . . . 53
8.5.5. Diagramas de Secuencia . . . 56
8.5.6. Consideraciones de extensibilidad, mantenibilidad y escalabilidad . . . 57
8.5.7. Resumen de la arquitectura de InfoVersionService . . . 61
8.6. Interfaz de Usuario . . . 61
8.6.1. Estructura de la interfaz . . . 61

9. Implementación **63**
9.1. Pautas de Estilo . . . 63
9.2. InfoVersionService . . . 64
9.3. Acceso a fuentes de datos . . . 65
9.3.1. Modelo de acceso a datos . . . 65
9.3.2. Integración con ArchivosOBU . . . 65
9.3.3. Integración con Transfer Manager . . . 65
9.4. Exposición de datos - SoaBasicContentManager . . . 66
9.5. Frontend . . . 66
9.5.1. Descripción de la interfaz . . . 67
9.6. Gestión de errores y validaciones . . . 69
9.7. Integración continua y gestión del código . . . 69

10. Pruebas **71**
10.1. Pautas de Estilo . . . 71
10.2. Pruebas Unitarias . . . 72
10.2.1. Cobertura de la aplicación . . . 73
10.3. Pruebas de Integración . . . 74
10.4. Pruebas funcionales . . . 76
10.4.1. CU1 - Registrar versiones . . . 76
10.4.2. CU2 - Comparación de versiones generales . . . 76
10.4.3. CU3 – Consulta específica de versiones . . . 76
10.4.4. CU4 – Consulta de KPIs de la flota . . . 77
10.4.5. Conclusión . . . 77
10.5. Pruebas de rendimiento . . . 77
10.6. Validación con usuarios finales . . . 78

IV Conclusiones **79**

11. Conclusiones y trabajo futuro **81**
11.1. Introducción . . . 81
11.2. Aportaciones realizadas . . . 81
11.3. Valoración del resultado . . . 82
11.4. Mejoras a futuro . . . 82

11.5. Objetivos personales 83

Appendices **85**

Apéndice A. Manual de Instalación **87**
A.1. Requisitos previos 87
A.2. Instalación del servicio InfoVersionService 87
A.2.1. Compilación 87
A.2.2. Instalación como servicio de Windows 88
A.3. Base de datos 88

Apéndice B. Manual de Usuario **91**
B.1. Acceso al módulo 91
B.2. Vista general 91
B.3. Filtros y búsquedas 92
B.4. Consulta detallada por vehículo 92
B.5. Visualización de KPIs 92
B.6. Usabilidad 93

Bibliografía **95**

Índice de cuadros

3.1. Resumen del Sprint 0 . . . 10
3.2. Resumen del Sprint 1 . . . 11
3.3. Resumen del Sprint 2 . . . 11
3.4. Resumen del Sprint 3 . . . 12
3.5. Resumen del Sprint 4 . . . 12
3.6. Resumen del Sprint 5 . . . 13
3.7. Resumen del Sprint 6 . . . 13
3.8. Resumen de todos los sprints . . . 13
3.9. Exposición al Riesgo . . . 14
3.10. R01 - Cambios en los Requisitos . . . 14
3.11. R02 - Estudio de asignaturas pendientes . . . 14
3.12. R03 - Falta de experiencia con herramientas técnicas . . . 14
3.13. R04 - Integración técnica más compleja de lo esperado . . . 15
3.14. R05 - Pérdida de datos o archivos del proyecto . . . 15
3.15. R06 - Problemas de salud . . . 15
3.16. Resumen del presupuesto del proyecto . . . 16

6.1. Carencia identificada y Solución propuesta . . . 26

7.1. Identificación de necesidades . . . 31
7.2. CU1 - Registrar versiones generadas . . . 35
7.3. CU2 - Comparar Versiones . . . 36
7.4. CU3 - Consulta específica de versiones . . . 37
7.5. CU4 - Consulta de KPIs de la flota . . . 37

10.1. Coverage del servicio *InfoVersionService* . . . 73
10.2. Trazabilidad entre requisitos funcionales y pruebas realizadas . . . 77

Índice de figuras

3.1. Pilares de Scrum . . . 8
3.2. Roles de Scrum . . . 8
3.3. Ciclo de eventos y artefactos Scrum[5] . . . 9

7.1. Flujo actual de la gestion de contenidos . . . 30
7.2. Diagrama de casos de uso . . . 33

8.1. Singleton . . . 42
8.2. Fachada . . . 43
8.3. Inyección de Dependencias . . . 43
8.4. Strategy . . . 44
8.5. Template . . . 45
8.6. Modelado de Datos . . . 47
8.7. Diagrama de Paquetes . . . 48
8.8. SubPaquetes Architecture . . . 49
8.9. SubPaquetes Application . . . 50
8.10. Dependencias entre capas . . . 52
8.11. Diagrama de Secuencia ProcessFolder . . . 58
8.12. Diagrama de Secuencia ProcessPackage . . . 59
8.13. Diagrama de Secuencia InsertFileDetails . . . 60
8.14. Diagrama de Secuencia CleanOldRecords . . . 60

9.1. Frontend de la nueva funcionalidad . . . 68

10.1. Paquetes resultado de las pruebas de integracion . . . 75
10.2. Archivos resultado de las pruebas de integracion . . . 75

Parte I

Objeto, Concepto y Método

Capítulo 1

Introducción

1.1 Introducción

Los sistemas de transporte inteligente han evolucionado significativamente en los últimos años, integrando tecnologías de información, automatización y comunicaciones para mejorar la eficiencia operativa, la seguridad y la experiencia del usuario. Una parte esencial de estos sistemas es la correcta distribución y sincronización de contenidos, como configuraciones, archivos multimedia o datos operativos entre otros, en los equipos embarcados (*OBU*, *On-Board Units*) que tiene cada vehículo.

En el contexto de estos sistemas, los procesos de actualización de contenidos a bordo representan un componente crítico para asegurar que todos los vehículos son desplegados bajo las mismas condiciones, con la información más reciente y coherente. Sin embargo, en muchas implementaciones reales, el seguimiento del estado de estas actualizaciones aún se realiza de forma manual, lo que introduce riesgos operativos, posibles errores humanos y una falta de visibilidad en tiempo real sobre el estado de la flota.

Este Trabajo Fin de Grado se enmarca en ese contexto, abordando el diseño e implementación de una solución software que permita automatizar la verificación del estado de actualización de contenidos en los vehículos de una red de transporte inteligente. El sistema desarrollado proporcionará trazabilidad completa de versiones, permitiendo identificar con precisión qué vehículos han recibido correctamente las actualizaciones y cuáles no, facilitando así una operación más segura y eficiente.

1.2 Motivación

Este Trabajo de Fin de Grado surge en el marco de las prácticas profesionales realizadas en GMV[1], una empresa reconocida por sus soluciones tecnológicas en sectores como el transporte, el espacio, la defensa y la ciberseguridad. En el caso de la sede de Valladolid, el enfoque principal es el Transporte Inteligente, ofreciendo servicios a nivel tanto nacional como internacional. Durante el periodo de prácticas, se identificó la necesidad de optimizar el proceso de actualización de contenidos en las flotas de transporte, lo que motivó la elaboración de este proyecto.

Actualmente, en el sistema sobre el que se desarrolla este proyecto, las comprobaciones sobre si un vehículo ha recibido o no una actualización se realizan de forma manual. Esto implica inspeccionar directorios de archivos generados por distintas herramientas (*ArchivosOBU, Transfer Manager*) y cruzar información de manera no automatizada, lo que genera ineficiencias y un elevado margen de error.

Esta situación es especialmente crítica cuando se planifica un despliegue. Por ejemplo, si se lanza una nueva campaña con contenidos actualizados (como anuncios, información al pasajero o configuraciones de red), y un subconjunto de vehículos no ha recibido correctamente los archivos, esos autobuses pueden comportarse de forma diferente al resto: mostrar información incorrecta en los monitores, emitir mensajes obsoletos o incluso fallar en tareas automatizadas. Esto no solo compromete la calidad del servicio, sino que también dificulta la detección y resolución de errores, ya que actualmente no se dispone de una visión centralizada del estado real de cada vehículo.

La motivación principal de este proyecto es, por tanto, automatizar este proceso de verificación, incorporando una solución que lea, interprete y relacione los datos generados por los distintos componentes del sistema, y exponga de forma clara y precisa el estado de actualización de cada vehículo. De esta manera, se podrá garantizar que la flota se encuentra en condiciones homogéneas antes de entrar en servicio.

Desde el punto de vista académico y formativo, este proyecto permite aplicar de manera práctica conocimientos en arquitectura software, integración de sistemas distribuidos, desarrollo backend y frontend, optimización de procesos por computación paralela, así como metodologías de análisis y diseño en un entorno real, complejo y en producción. El resultado es una solución software que proporciona trazabilidad completa de versiones, permitiendo identificar con precisión qué vehículos han recibido correctamente los contenidos y facilitando así una operación más segura, eficiente y controlada.

Capítulo 2

Objetivos y Alcance

En esta sección estarán expuestos los objetivos, tareas y el alcance de este proyecto

2.1 Objetivos

2.1.1 Objetivos generales

El objetivo principal de este Trabajo Fin de Grado es diseñar e implementar una solución software que permita automatizar el proceso de verificación de actualizaciones de contenido en equipos embarcados (*OBU*) dentro de una red de transporte inteligente. La solución debe ofrecer trazabilidad de versiones, integrarse con los sistemas existentes (*ArchivosOBU, Transfer Manager, Gestor de Contenidos*) y proporcionar una interfaz clara para la consulta del estado de actualización por vehículo.

2.1.2 Objetivos específicos

- Mejorar la eficiencia operativa en la gestión de actualizaciones dentro del Gestor de Contenidos de GMV.
- Reducir la necesidad de comprobaciones manuales, optimizando el proceso de despliegue de los equipos.

2.1.3 Objetivos personales

- Aplicar los conocimientos adquiridos a lo largo del grado en un entorno real de desarrollo
- Familiarizarme con las herramientas utilizadas en GMV para el control de versiones y la gestión de proyectos
- Mejorar habilidades de documentación, análisis funcional y validación.
- Adquirir experiencia en la resolución de problemas e implementación de nuevas funcionalidades en entornos con sistemas complejos ya desplegados.
- Profundizar y aprender las tecnologías utilizadas en GMV (C#, React, Servicios de Windows, IIS, etc)

2.1.4 Tareas a realizar

- Definir el trabajo y elaborar una planificación
    - Establecer el alcance del proyecto, definiendo los objetivos generales y específicos.

    - Asignar los recursos necesarios y definir las herramientas a utilizar durante el desarrollo.

- Estudiar el problema
    - Analizar el sistema actual de actualización de contenidos en flotas de transporte y detectar sus principales limitaciones.
    - Definir los requisitos funcionales y no funcionales del sistema de trazabilidad
- Desarrollar nuestra solución
    - Evaluar alternativas arquitectónicas para la integración de la solución.
    - Diseñar la arquitectura lógica y técnica del sistema.
    - Modelar los datos necesarios para realizar la comparación entre las versiones esperadas y las que realmente tiene cada equipo
    - Desarrollar un backend capaz de gestionar dicha información de forma estructurada
    - Diseñar una interfaz de usuario que se integre con la del Gestor de Contenidos actual y sea sencilla y accesible para los operadores técnicos.
- Probarla o realizar experimentos con ella
    - Definir y ejecutar escenarios de prueba que simulen diversas condiciones operativas, incluyendo actualizaciones exitosas, incompletas o fallidas en dispositivos con diferentes versiones y topologías.
    - Recoger y analizar feedback de usuarios finales (clientes y personal técnico) a través de pruebas de usabilidad, para iterar mejoras en la funcionalidad y la interfaz gráfica. Por motivos de tiempo y alcance no se consiguió iterar las mejoras de funcionalidad e interfaz aunque sí se recogió el feedback mediante pruebas de validación.

2.2 Alcance

Este proyecto se centra en el desarrollo de una solución específica para el seguimiento y verificación del estado de versiones de contenido en equipos embarcados dentro de un entorno de transporte inteligente.

El alcance del TFG incluye:

- El análisis y diseño de la arquitectura de trazabilidad.
- La implementación de la lógica de comparación de versiones.
- La creación de una interfaz orientada a operadores técnicos.
- La integración con las herramientas existentes (ArchivosOBU y Transfer Manager) mediante el análisis de archivos generados por estos.

Queda fuera del alcance:

- La modificación directa del sistema de distribución (Transfer Manager)
- La gestión de otros tipos de contenido que no se reflejen en los archivos generados automáticamente.

Capítulo 3

Metodología

3.1 Enfoque de desarrollo

Para el desarrollo del presente Trabajo Fin de Grado se ha adoptado el marco de trabajo Scrum[2], una metodología ágil[3] ampliamente utilizada en la industria del software, especialmente en entornos donde se requiere flexibilidad, colaboración continua y entregas incrementales. Dado que este proyecto se ha desarrollado en el contexto de unas prácticas profesionales en una empresa tecnológica, y en coordinación con un equipo real, la aplicación de Scrum ha permitido alinear el trabajo con las dinámicas y herramientas del entorno profesional.

3.1.1 ¿Qué es Scrum?

Scrum es un marco de trabajo ágil orientado al desarrollo iterativo e incremental de productos complejos. Fue inicialmente planteado para proyectos de software, pero hoy en día se aplica en múltiples disciplinas.

Su principal objetivo es entregar valor de forma continua, a través de ciclos cortos de desarrollo llamados sprints, que en el caso de GMV tienen una duración de una a tres semanas, que permiten inspeccionar y adaptar el trabajo de manera constante.

Este enfoque es especialmente útil en entornos donde los requisitos pueden evolucionar con el tiempo o no están completamente definidos desde el inicio, permitiendo que los equipos respondan a cambios de forma ágil y eficaz.

Scrum no impone una metodología rígida, sino que proporciona roles, eventos y artefactos que ayudan a estructurar el trabajo de forma colaborativa, transparente y adaptable.

Se basa en tres pilares fundamentales que garantizan la transparencia el control del progreso y la mejora continua.

3.1.2 Pilares fundamentales de Scrum

Scrum se sustenta en tres pilares fundamentales[4] que permiten mantener el control, la transparencia y la mejora continua del proceso

- **Transparencia:** Todos los aspectos significativos del proceso deben ser visibles para quienes gestionan los resultados. Todos los artefactos generados deben estar accesibles y ser comprensibles para todo el equipo. Esto garantiza una visión compartida del progreso y del producto.
- **Inspección:** El equipo debe inspeccionar regularmente el progreso hacia el objetivo final con el fin de detectar desviaciones y corregirlas de forma temprana.

- **Adaptación:** Cuando se detectan desviaciones relevantes, el equipo debe estar preparado para ajustar su forma de trabajar, sus tareas o incluso el alcance. Esta capacidad de adaptación constante es lo que permite a Scrum responder a entornos cambiantes.

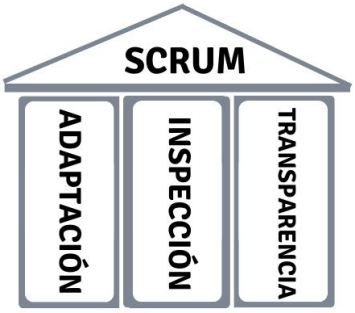


Figura 3.1: Pilares de Scrum

3.1.3 Componentes de Scrum

Scrum está compuesto por roles, eventos y artefactos, que estructuran todo el trabajo del equipo y promueven la entrega continua de valor.

Roles

Hay tres roles principales distinguibles:

- **Product Owner:** es el responsable de maximizar el valor del producto. Gestiona el Product Backlog y prioriza las funcionalidades según el valor para el cliente.
- **Scrum Master:** se encarga de asegurarse de la aplicación de las prácticas Scrum, eliminando impedimentos y ayudando al equipo a mejorar sus procesos. También es el responsable de ser el comunicador entre el equipo y el cliente.
- **Equipo de desarrollo:** es un grupo multidisciplinar el cual se encarga de convertir los elementos del Sprint Backlog en incrementos funcionales.

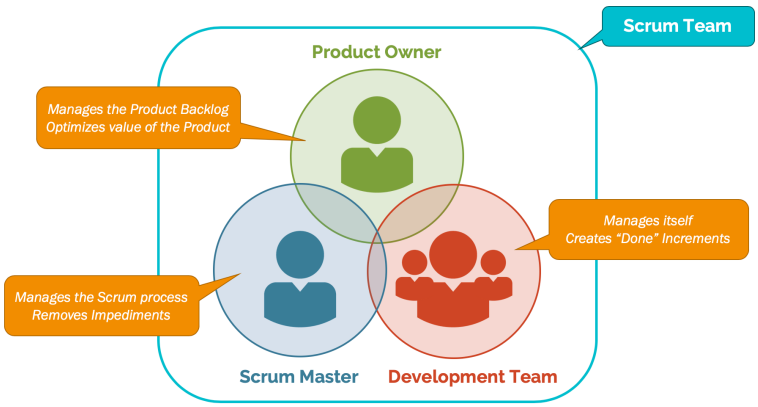


Figura 3.2: Roles de Scrum

Eventos

- **Sprint:** es un periodo de tiempo fijo (normalmente entre 1 y 3 semanas en GMV) en el que se desarrolla un incremento del producto. Contiene a todos los demás eventos y siempre comienza un nuevo sprint inmediatamente después de finalizar el anterior. A lo largo del sprint los requisitos quedan congelados y no se pueden cambiar.
- **Sprint Planning:** se definen los objetivos del sprint y qué tareas del Product Backlog se abordarán.
- **Daily Scrum:** reunión diaria para inspeccionar el progreso, detectar bloqueos y coordinarse con el resto del equipo. Suele durar un máximo de 15 minutos y siempre se realiza a la misma hora y en el mismo lugar. Su principal objetivo es mejorar la comunicación y promover la toma de decisiones eliminando por ende la necesidad de otras reuniones
- **Sprint Review:** al final del Sprint, el equipo presenta todo lo que se ha realizado y recibe feedback al respecto.
- **Sprint Retrospective:** es una sesión interna para reflexionar y proponer mejoras en el proceso. Este evento concluye el sprint actual.

Artefactos

- **Product Backlog:** es una lista priorizada de todo lo que se desea incluir en el producto final. Es dinámica y está en continua evolución. El responsable de este artefacto es el *Product Owner*. El objetivo que se persigue es llamado el *Product Goal*, que hace referencia al estado final o futuro del producto al que se quiere llegar.
- **Sprint Backlog:** es el subconjunto del *Product Backlog* seleccionado para el sprint actual, junto con el plan desarrollado para entregar el *incremento*. Los principales responsables de realizar este trabajo son los desarrolladores. Es una representación del *Sprint Goal*, que es el objetivo que se ha establecido para el sprint actual.
- **Incremento:** Es el resultado del trabajo realizado a lo largo de un sprint. Cada incremento es aditivo a todos los anteriores, fusionandose para acercarse al *Product Goal*. También existe la posibilidad de que se generen múltiples incrementos en un único sprint.

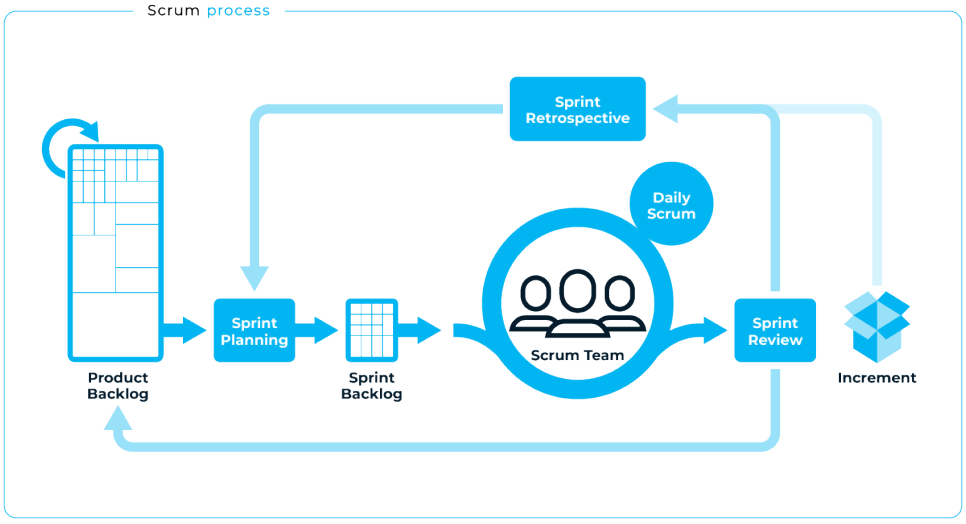


Figura 3.3: Ciclo de eventos y artefactos Scrum[5]

3.1.4 Aplicación al Proyecto

En este TFG, Scrum se ha aplicado de forma adaptada al contexto real de prácticas en empresa, trabajando con un equipo profesional que ya sigue esta metodología.

- El tutor de la empresa tomará tanto el papel de Scrum Master como el de Product Owner. Por otro lado, el estudiante tendrá el papel del equipo de desarrollo, pues será el encargado de hacer todo el desarrollo e implementación propuesto.
- Los Sprints se han definido con un máximo de duración de dos semanas, cada uno de ellos con objetivos claros definidos. Por ejemplo, análisis de requisitos, diseño, desarrollo del backend, etc.
- Se ha participado en reuniones diarias *(Daily Scrums)* para compartir los avances y resolver bloqueos.
- Se usaron herramientas reales del entorno profesional para la planificación, seguimiento y documentación, como *Git (BitBucket), Jira y Confluence*. El *Product Backlog* y el *Sprint Backlog* han sido gestionados por el tutor de la empresa mediante la herramienta anteriormente nombrada (*Jira*).
- Al finalizar cada sprint, se realizaron revisiones de entregables con el equipo técnico y se ajustaron los próximos pasos en función del feedback recibido.

Esta aplicación práctica de Scrum ha permitido que el desarrollo del sistema se adapte a las necesidades reales del entorno, facilitando la integración progresiva con componentes existentes y favoreciendo un desarrollo ágil, trazable y flexible.

3.2 Planificación

En esta sección se mostrará las diferentes planificaciones que se han tomado en cada uno de los sprints realizados. Se han realizado un total de 6 sprints, cada uno de ellos con una duración de 2 semanas aproximadamente. Como aclaración, en la parte de documentación de la memoria se incluyen todas aquellas reuniones que se hayan tenido con el tutor para verificarla.

3.2.1 Sprint 0 - 12/03/2025 - 26/03/2025

Las fases y duración prevista del sprint 0 están representadas en el cuadro 3.1. Este sprint se ha dedicado principalmente a toda la preparación y configuración de aquellas herramientas y utensilios que van a ser necesarios para la realización del proyecto como por ejemplo Visual Studio, Visual Studio Code, SQL Server y las aplicaciones internas de GMV como *ArchivosObu* o el *Content Manager*.

Nombre de actividad	**Tiempo Estimado**	**Tiempo Invertido**	**Estado**
Configuración de entorno de trabajo	15 h	16 h	Completado
Lectura de la documentación inicial proporcionada por la empresa	4 h	4 h	Completado
Adaptación de plantilla del TFG	1 h	1 h	Completado
Documentación de la memoria	10 h	9 h 30 min	Completado
TOTAL	**30 h**	**30 h 30 min**	**Completado**

Cuadro 3.1: Resumen del Sprint 0

3.2.2 Sprint 1 - 26/03/2025 - 08/04/2025

Las fases y duración prevista del sprint 1 están representadas en el cuadro 3.2. Este sprint se centró en conocer en profundidad el sistema actual de actualizaciones de contenidos en los equipos embarcados. Se revisó documentación, se identificaron los actores clave (*ArchivosOBU*, *Transfer Manager*, *SIU*) y en base a todo esto se desarrolló un documento con varias soluciones propuestas para elegir aquella que más beneficiase al desarrollo.

Nombre de actividad	**Tiempo Estimado**	**Tiempo Invertido**	**Estado**
Revisión de documentación y flujos actuales	8 h	10 h	Completado
Toma de requisitos iniciales	2 h	2 h	Completado
Análisis del flujo de actualización	8 h	6h 30 min	Completado
Estudio de soluciones posibles	5 h	5h	Completado
Primeros diagramas preliminares para cada solución	5 h	5h	Completado
Redacción de documento con soluciones propuestas	5 h	5h	Completado
Documentación de la memoria	8 h	9 h	Completado
TOTAL	**41 h**	**42 h 30 min**	**Completado**

Cuadro 3.2: Resumen del Sprint 1

3.2.3 Sprint 2 - 9/04/2025 - 22/04/2025

Las fases y duración prevista del sprint 2 están representadas en el cuadro 3.3. Durante este sprint se definieron los requisitos funcionales y no funcionales, se evaluaron posibles arquitecturas (microservicio independiente vs integración). Se definieron los modelos de datos preliminares, los flujos principales de operación y se comenzaron los primeros bocetos de interfaz de usuario. El diseño se validó con el equipo de trabajo para asegurar la viabilidad técnica y su encaje en la arquitectura real del sistema.

Nombre de actividad	**Tiempo Estimado**	**Tiempo Invertido**	**Estado**
Redacción de requisitos funcionales y no funcionales	4 h	4 h	Completado
Evaluación de arquitectura y selección	8 h	8 h	Completado
Diseño de arquitectura y flujos	10 h	10 h	Completado
Modelado inicial de datos	6 h	5 h	Completado
Elaboración de alternativas para la interfaz	6 h	4 h	Completado
Documentación de la memoria	10 h	10 h	Completado
TOTAL	**44 h**	**41 h**	**Completado**

Cuadro 3.3: Resumen del Sprint 2

3.2.4 Sprint 3 - 23/04/2025 - 06/05/2025

Las fases y duración prevista del sprint 3 están representadas en el cuadro 3.4. En este sprint se iniciaron las tareas de desarrollo del backend, centrándose en la definición del modelo de datos en base de datos y la creación de las primeras estructuras de código. Se implementó la lógica inicial de comparación de versiones

entre contenidos esperados y contenidos actuales. También se comenzó la documentación técnica de las APIs para su futura integración con el frontend.

Nombre de actividad	Tiempo Estimado	Tiempo Invertido	Estado
Implementación del modelo de datos	4 h	4 h	Completado
Estructura inicial del backend	15 h	15 h	Completado
Lógica de comparación de versiones	15 h	13 h	Completado
Documentación técnica del backend en Swagger y Confluence	4 h	4 h	Completado
Documentación de la memoria	10 h	12 h	Completado
TOTAL	**48 h**	**48 h**	**Completado**

Cuadro 3.4: Resumen del Sprint 3

3.2.5 Sprint 4 - 07/05/2025 - 20/05/2025

Las fases y duración prevista del sprint 4 están representadas en el cuadro 3.5. Este sprint se dedicó a conectar el sistema con las fuentes reales de información: los archivos generados por *ArchivosOBU* (conteniendo las versiones esperadas por equipo) y los archivos devueltos por los equipos tras la actualización, gestionados por *Transfer Manager* (que contienen las versiones actuales instaladas). Se desarrollaron módulos de lectura y parsing de estos archivos, y se almacenaron los datos extraídos en la base de datos de manera que se pudiese acceder rápidamente a los datos necesarios para las comprobaciones de versión.

Nombre de actividad	Tiempo Estimado	Tiempo Invertido	Estado
Lectura de archivos generados por ArchivosOBU	5 h	5 h	Completado
Lectura de archivos generados por TransferManager	5 h	5 h	Completado
Desarrollo de parsers y almacenamiento en la base de datos	10 h	10 h	Completado
Integración con el backend Inicial	7 h	4 h	Completado
Documentación de la memoria	10 h	17 h	Completado
TOTAL	**41 h**	**41 h**	**Completado**

Cuadro 3.5: Resumen del Sprint 4

3.2.6 Sprint 5 - 21/05/2025 - 03/06/2025

Las fases y duración prevista del sprint 5 están representadas en el cuadro 3.6. Con la base funcional ya operativa, este sprint se enfocó en la mejora y optimización del backend. Se refactorizó el código para mejorar mantenibilidad, se implementaron filtros y se realizaron pruebas funcionales completas. Además, se preparó el backend para su conexión con la interfaz de usuario, asegurando la disponibilidad de los datos mediante endpoints. Esta fase también incluyó una revisión técnica del sistema en su conjunto y la mejora de la documentación técnica para facilitar la integración con otros módulos.

Nombre de actividad	Tiempo Estimado	Tiempo Invertido	Estado
Refactorización del código	6 h	12 h	Completado
Implementación de filtros	8 h	4 h	Completado
Pruebas funcionales	10 h	13 h	Completado
Revisión y mejora de la documentación técnica	4 h	4 h	Completado
Documentación de la memoria	10 h	10 h	Completado
TOTAL	**38 h**	**43 h**	**Completado**

Cuadro 3.6: Resumen del Sprint 5

3.2.7 Sprint 6 - 04/06/2025 - 06/07/2025

Las fases y duración prevista del sprint 6 están representadas en el cuadro 3.7. Este sprint se centró principalmente en el desarrollo del frontend y en su integración con el backend previamente optimizado. Las tareas incluyeron el diseño de mockups para la interfaz de usuario, el desarrollo e implementación del frontend, y la conexión funcional entre la interfaz y el backend mediante endpoints ya disponibles. Además, se planificaron pruebas funcionales para garantizar una experiencia de usuario fluida y la correcta comunicación entre componentes. Finalmente, se contempló la elaboración de la documentación final del proyecto y su revisión, con el objetivo de dejar el sistema listo para su presentación o entrega. Este sprint se alargó más de lo esperado debido a la convocatoria extraordinaria de la asignatura de Física, teniendo su fecha final prevista para el 17/06/2025.

Nombre de actividad	Tiempo Estimado	Tiempo Invertido	Estado
Diseño de mockups	3 h	3 h	Completado
Implementación del frontend	20 h	17 h	Completado
Conexión con el back	6 h	7 h	Completado
Pruebas funcionales y ajustes	6 h	6 h	Completado
Documentación final de la memoria y revisión	25 h	35 h	Completado
TOTAL	**60 h**	**68 h**	**Completado**

Cuadro 3.7: Resumen del Sprint 6

3.2.8 Plan de control y Riesgos

Esta sección tiene como objetivo analizar los posibles riesgos para este proyecto. Además en el cuadro 3.8 se puede ver un resumen general del tiempo estimado y el invertido en cada *sprint*.

Sprint	Tiempo Estimado	Tiempo Invertido
Sprint 0	30 h	30 h 30 mins
Sprint 1	41 h	42 h 30 mins
Sprint 2	44 h	41 h
Sprint 3	48 h	48 h
Sprint 4	41 h	41 h
Sprint 5	38 h	43 h
Sprint 6	60 h	68 h
Total	**302 h**	**314 h**

Cuadro 3.8: Resumen de todos los sprints

En el cuadro 3.9 se puede ver la tabla que se ha usado para calcular la exposición al riesgo correspondiente

con la probabilidad e impacto del mismo. En este proyecto se han identificado 6 riesgos que pueden analizarse en los cuadros 3.10, 3.11, 3.12, 3.13, 3.14 y 3.15.

Impacto/Prob	**Baja**	**Media**	**Alta**
Bajo	Bajo	Bajo	Medio
Medio	Bajo	Medio	Alto
Alto	Medio	Alto	Alto

Cuadro 3.9: Exposición al Riesgo

ID	R01
Nombre	Cambios en los requisitos
Descripción	Los requisitos pueden cambiar debido a nuevas necesidades detectadas.
Probabilidad	Media
Impacto	Alto
Exposición	Alta
Plan de mitigación	Validar los requisitos con el equipo antes de iniciar cada sprint
Plan de contingencia	Replanificar los sprints y revisar el alcance

Cuadro 3.10: R01 - Cambios en los Requisitos

ID	R02
Nombre	Estudio de asignaturas pendientes
Descripción	La planificación y fechas del proyecto se pueden ver afectadas porque las asignaturas pendientes toman más tiempo del esperado.
Probabilidad	Alta
Impacto	Alto
Exposición	Alta
Plan de mitigación	Planificar y coordinar los estudios de las asignaturas pendientes con la planificación del proyecto
Plan de contingencia	Establecer prioridades y horarios estrictos.

Cuadro 3.11: R02 - Estudio de asignaturas pendientes

ID	R03
Nombre	Falta de experiencia con herramientas técnicas
Descripción	Uso de tecnologías nuevas.
Probabilidad	Media
Impacto	Medio
Exposición	Media
Plan de mitigación	Utilizar los cursos proporcionados por la empresa para aprender dichas tecnologías
Plan de contingencia	Pedir ayuda puntual o buscar soluciones alternativas.

Cuadro 3.12: R03 - Falta de experiencia con herramientas técnicas

ID	R04
Nombre	Integración técnica más compleja de los esperado
Descripción	Puede que la integración con archivosOBU y el Transfer Manager sea más compleja de lo esperado o la estructura de los archivos generados sea costosa de procesar
Probabilidad	Alta
Impacto	Medio
Exposición	Alta
Plan de mitigación	Reestructurar la planificación teniendo en cuenta la complejidad real.
Plan de contingencia	Validar el acceso con el equipo técnico en las etapas tempranas del proyecto.

Cuadro 3.13: R04 - Integración técnica más compleja de lo esperado

ID	R05
Nombre	Pérdida de datos o archivos del proyecto
Descripción	Fallo en el ordenador, disco duro o pérdida de versiones.
Probabilidad	Baja
Impacto	Alto
Exposición	Media
Plan de mitigación	Hacer backups regulares o utilizar sistemas en la nube.
Plan de contingencia	Recuperar todo desde repositorios remotos.

Cuadro 3.14: R05 - Pérdida de datos o archivos del proyecto

ID	R06
Nombre	Problemas de salud
Descripción	Si el desarrollador contrae alguna enfermedad puede afectar a los plazos y estimación del tiempo del proyecto
Probabilidad	Media
Impacto	Medio
Exposición	Media
Plan de mitigación	Añadir algo de holgura en los sprints en caso de que el desarrollador contraiga alguna enfermedad.
Plan de contingencia	Replanificar el proyecto y sus fechas en caso de ser necesario.

Cuadro 3.15: R06 - Problemas de salud

De todos estos riesgos han ocurrido el estudio de asignaturas pendientes debido a la recuperación de Ampliación de Matemáticas y Física y problemas de salud debidos a una conjuntivitis grave que impidió el correcto avance durante un par de semanas. Todo esto llevó a la replanificación del proyecto para su entrega en convocatoria extraordinaria, motivos por los que el Sprint 63.7 tiene una duración mayor.

3.3 Costes

Esta sección cubrirá todo lo relacionado con los costes del proyecto. Estarán calculados tanto los costes humanos como los técnicos y aquellos asociados a la documentación y la defensa.

3.3.1 Coste humano

El puesto que ocupa el estudiante en este proyecto es el de un desarrollador fullstack junior. Este puesto en españa cobra de media un total de 10,77€/h [6], habiendo deducido ya los impuestos. El proyecto ha durado un total de 289 h, por lo que el coste humano del proyecto será de 314hx10,77€/h = **3381,78 €**.

3.3.2 Costes de Hardware

Durante el desarrollo del proyecto se han utilizado un ordenador portátil. El coste asociado a este dispositivo se ha calculado mediante amortización mensual, considerando como vida útil estimada 48 meses o 4 años, que corresponde a una media estándar en entornos profesionales[7]. El portatil utilizado es un Lenovo ThinkPad P14S Gen 4 con un precio de 1793,31€[8] a día 09/04/2025. Esto hace que el precio amortizado aproximado sea de 37,36€/mes. Por lo tanto, al durar el proyecto un aproximado de 3 meses, el coste del portátil ha sido de **112,08€**.

3.3.3 Costes de Software

Respecto al software usado, se ha necesitado la licencia de Microsoft 365 Enterprise para el uso de Microsoft Teams con un costo de 11,70€/mes [9] lo que hace un coste total de **35,1€**. Además, también se ha utilizado la licencia de Visual Studio 2022 Professional con un coste de 45€/mes [10] lo que hace un coste final de **135€**. La última licencia utilizada es la de Astah Professional la cual tiene un coste de 11,99€/mes [11] lo que hace un total de **35,97€**.

3.3.4 Presupuesto Total

En el cuadro 3.16 se puede observar un resumen del presupuesto con el coste final del proyecto.

Nombre	**Precio Parcial**	**Horas | Mes**	**Precio Total**
Trabajo del desarrollador	10.77€/h	314 h	3381,78€
Portátil utilizado	37,36€/mes	3 meses	112,08€
Licencia Microsoft 365 Enterprise	11,70€/mes	3 meses	35,1€
Licencia Visual Studio 2022 Professional	45€/mes	3 meses	135€
Licencia Astah Professional	11,99€/mes	3 meses	35,97€
		Total	**3699,93€**

Cuadro 3.16: Resumen del presupuesto del proyecto

Parte II

Marco Conceptual y Contexto

Capítulo 4

Marco Contextual

4.1 Entorno Profesional

El presente Trabajo Fin de Grado se desarrolla en el marco de las prácticas externas realizadas en la empresa GMV, una multinacional tecnológica con actividad en diversos sectores estratégicos como el transporte, el espacio, la defensa, la ciberseguridad y los sistemas inteligentes.

En concreto, el proyecto se ha llevado a cabo en la sede de GMV Valladolid, especializada en soluciones de Transporte Inteligente. Esta línea de negocio ofrece productos y servicios a operadores de transporte público tanto a nivel nacional como internacional, abarcando desde sistemas embarcados y centros de control, hasta plataformas de información al pasajero y herramientas de análisis y gestión.

El entorno de trabajo en GMV es altamente técnico, multidisciplinar y orientado a la integración de tecnologías avanzadas en sistemas reales, lo que proporciona un marco ideal para el desarrollo de proyectos con aplicaciones prácticas directas, como el que aquí se presenta.

4.2 Contexto operativo del proyecto

En los sistemas de transporte inteligente que gestiona GMV, cada vehículo de la flota cuenta con un equipo embarcado (OBU), encargado de recibir, procesar y mostrar contenidos tales como información al pasajero, contenido multimedia y campañas de comunicación o servicio.

Estos contenidos deben ser actualizados de forma periódica, coherente y controlada en toda la flota. El proceso de actualización implica distintos componentes del sistema, como:

- **SIU (Gestor de Contenidos o Sistema de información al usuario)**: sistema donde se configuran y gestionan los contenidos de cada bus o flota. Contenidos hacen referencia a todo artefacto que pueda contener el bus, desde audios, imágenes, hasta configuraciones sobre rutas, líneas, etc.

- **ArchivosOBU**: servicio de windows que genera los ficheros necesarios para actualizar cada vehículo con los datos actualizados por medio del SIU.

- **Transfer Manager**: sistema encargado de programar y ejecutar la transferencia de ficheros a los vehículos. A su vez, cuando un vehículo es actualizado, genera unos archivos con los contenidos que tiene el vehículo.

4.3 Problemática detectada

Durante el periodo de prácticas, se detectó que, aunque el proceso de generación y envío de contenidos a los equipos está bien definido y automatizado, no existe una solución integrada que permita comprobar de forma centralizada y automática si los contenidos han sido efectivamente recibidos e instalados por cada vehículo, por ejemplo, puede ocurrir que en el momento de la transferencia el sistema estuviese apagado.

Actualmente, estas comprobaciones se realizan de forma manual, revisando directorios, archivos y registros técnicos. Este procedimiento es costoso en tiempo y recursos, tiene alto riesgo de error humano, y dificulta la planificación y verificación de campañas de actualización en tiempo real.

Esto se vuelve especialmente crítico en contextos donde la homogeneidad del contenido es esencial, como en los lanzamientos de nuevas campañas de comunicación, cambios de tarifas o rutas, o con la coordinación con sistemas de información al usuario.

4.4 Justificación del proyecto

La necesidad de contar con una herramienta que permita verificar automáticamente el estado de actualización de cada vehículo es evidente desde el punto de vista operativo, técnico y de calidad del servicio.

Este proyecto surge precisamente como una propuesta de solución a esa necesidad, con el objetivo de automatizar la verificación del estado de actualización por vehículo, reducir la carga operativa asociada a comprobaciones manuales, aumentar la trazabilidad del proceso de distribución de contenidos y facilitar la toma de decisiones antes de sacar a circulación un vehículo.

Al integrarse en el entorno real de GMV, el sistema desarrollado no solo tiene aplicación práctica directa, sino que también puede convertirse en la base de una solución más general orientada a la trazabilidad de versiones, que podría evolucionar hacia un nuevo módulo funcional del ecosistema de productos de la empresa.

Capítulo 5

Marco Conceptual y Tecnológico

5.1 Arquitecturas distribuidas

Una arquitectura distribuida[12] es aquella en la que los componentes del sistema se encuentran físicamente separados, normalmente ejecutándose en diferentes máquinas o nodos, pero cooperan entre sí mediante una red para alcanzar un objetivo común. Este tipo de arquitectura es ampliamente utilizado en sistemas modernos debido a sus ventajas en términos de escalabilidad, disponibilidad, modularidad y resiliencia.

En el contexto de este proyecto, la arquitectura existente dentro de GMV se basa en un modelo distribuido compuesto por múltiples servicios, cada uno con una responsabilidad bien definida. Componentes como el SIU, el servicio ArchivosOBU, el Transfer Manager y los equipos embarcados (OBU) están distribuidos y conectados mediante una infraestructura de red corporativa.

La solución desarrollada en este TFG se incorpora como un nuevo módulo dentro de un servicio ya existente, respetando sus convenciones arquitectónicas y de despliegue. A mayores de este nuevo módulo, también se desarrollará un nuevo servicio de windows que gestione el tratamiento de los archivos de versiones utilizados por GMV. Aunque forman parte de un servicio actual, mantienen un alto grado de modularidad y responsabilidad única: cada función queda claramente delimitada y puede evolucionar de forma independiente sin impactar al resto de componentes. La comunicación con el núcleo del servicio se realiza a través de las mismas interfaces definidas (API REST o intercambio de archivos estructurados), garantizando compatibilidad y cohesión con el ecosistema ya desplegado.

5.2 Control de versiones en sistemas software

El control de versiones tradicionalmente se asocia con el desarrollo de software (por ejemplo, Git para código fuente). Sin embargo, el mismo concepto puede extenderse a contenidos operativos y configuraciones que deben ser gestionadas en múltiples dispositivos distribuidos.

En este proyecto, el control de versiones se aplica a los contenidos que deben ser desplegados y sincronizados entre los distintos vehículos de la flota. Existen dos elementos clave:

- **La versión esperada**, que es la que se genera como resultado de un cambio de contenido en el sistema central.
- **La versión real**, que corresponde a la información devuelta por los dispositivos embarcados una vez aplicadas las actualizaciones.

Comparar ambos tipos de versión permite verificar que los dispositivos están sincronizados, detectar errores o equipos no actualizados y obtener una trazabilidad de qué falta por actualizar.

Esta trazabilidad no solo es útil para garantizar la calidad operativa, sino que también permite disponer de un historial verificable de los cambios aplicados, útil en auditorías, control de versiones de contenido multimedia, o revisiones de configuración en caso de incidentes.

5.3 Transferencia de datos

En los sistemas distribuidos, la transferencia de datos y la sincronización de información entre componentes dispersos geográficamente son aspectos fundamentales para garantizar la coherencia operativa. Estos procesos permiten que distintos nodos del sistema compartan el mismo estado o contenido, y reaccionen adecuadamente ante cambios, manteniendo la fiabilidad y estabilidad del sistema global.

En el contexto de los sistemas de transporte inteligente, donde cada vehículo cuenta con una unidad embarcada (OBU) que opera de forma relativamente autónoma, la necesidad de mantener contenidos actualizados y sincronizados con el sistema central cobra especial relevancia. Dichos contenidos pueden incluir archivos de configuración, campañas de información, datos multimedia o cualquier otro recurso necesario para el funcionamiento diario del servicio.

Uno de los desafíos técnicos principales en este tipo de entornos es que los canales de comunicación pueden ser intermitentes, la disponibilidad de los dispositivos no siempre es constante, y no todos los equipos se encuentran online al mismo tiempo. Por ello, la transferencia de datos debe ser robusta y tolerante a fallos, y el sistema debe disponer de mecanismos que permitan verificar si los datos fueron correctamente entregados y procesados por cada uno de los nodos.

La sincronización, en este contexto, implica confirmar que la información presente en los OBU es idéntica o equivalente funcionalmente a la que se generó en el sistema central. El objetivo del sistema propuesto es precisamente actuar como un mecanismo de verificación automatizado.

5.4 Sistemas de transporte inteligente

Los Sistemas de Transporte Inteligente (ITS, Intelligent Transport Systems)[13] son el conjunto de tecnologías que se aplican al transporte con el objetivo de mejorar la eficiencia, la seguridad, la sostenibilidad y la experiencia del usuario. Estos sistemas combinan tecnologías de la información, telecomunicaciones, automatización y electrónica para optimizar la operación tanto del transporte público como privado.

El concepto de ITS ha sido promovido a nivel global por organismos como la Comisión Europea y la ITS World Congress, dado su papel fundamental en la transformación del transporte hacia un modelo más digital, conectado y centrado en el usuario.

Un sistema ITS completo está formado por diversos elementos interconectados. Entre los más comunes se encuentran:

- **Centro de control (*backoffice*):** Es el cerebro del sistema. Desde aquí es desde donde se gestiona la planificación de las rutas, la supervisión de la flota, el estado del tráfico, la configuración de los vehículos, el contenido mostrado al pasajero, etc.
- **Sistemas embarcados (OBU):** Son los dispositivos instalados a bordo de los vehículos. Reciben instrucciones desde el centro de control y ejecutan funcionalidades como mostrar información al pasajero en las pantallas, emitir mensajes por megafonía, registrar y reportar eventos de operación o gestionar validadores, cámaras, sensores, etc.
- **Infraestructura de comunicaciones:** Es el canal que permite el intercambio de datos entre el centro de control y los vehículos. Puede incluir 4G/5G, Wi-Fi, VPN, redes satelitales u otras tecnologías según el entorno.
- **Interfaces de usuario y herramientas operativas:** Se refiere a las diferentes herramientas diseñadas para que el usuario interaccione con ellas con la posibilidad de cambiar cualquier tipo de información de

manera sencilla y transparente. Por ejemplo, a través de sistemas como el SIU (Sistema de Información al Usuario), los operadores pueden configurar campañas, cargar nuevos contenidos, programar actualizaciones o analizar el comportamiento de la red de transporte.

5.5 Tecnologías utilizadas

Durante el desarrollo del sistema de trazabilidad de versiones de contenidos se han utilizado múltiples tecnologías, herramientas y plataformas. La elección de cada una se ha basado en criterios de compatibilidad con el entorno de GMV, madurez tecnológica, documentación disponible y adecuación al flujo de trabajo ágil adoptado durante el proyecto.

Estas herramientas se agrupan según su área funcional y se describen a continuación, con el enfoque puesto en su aplicación concreta dentro del desarrollo del proyecto.

Cabe destacar además el uso de ChatGPT como herramienta de apoyo puntual durante el desarrollo. Se ha utilizado principalmente para resolver dudas sintácticas o estructurales en C#, obtener ejemplos de pruebas unitarias, mejorar la redacción técnica de algunos apartados del documento y validar estructuras conceptuales. Su uso ha estado siempre supervisado, contrastando los resultados obtenidos con documentación oficial o el comportamiento real del sistema, actuando como un asistente complementario dentro del flujo de trabajo.

5.5.1 Backend y lógica de negocio

El *backend* es el núcleo funcional del sistema, encargado de gestionar los datos, comparar versiones, almacenar resultados y exponer servicios REST para su consulta desde la interfaz. El lenguaje principal del *backend* ha sido C# (.NET), utilizado para desarrollar la lógica de comparación, carga de archivos, y operaciones sobre los datos. Esta elección se alinea con el stack tecnológico ya existente en GMV para otras herramientas.

Respecto a la base de datos, se ha utilizado MySQL, un sistema de gestión de bases de datos relacional utilizado para almacenar las versiones esperadas y las reales por cada componente. Se ha elegido por su fiabilidad, rendimiento y compatibilidad con entornos productivos así como para mantener coherencia con el resto de bases de datos dentro de GMV.

5.5.2 Frontend

El frontend permite la visualización del estado de actualización de los vehículos, de forma sencilla y accesible para técnicos u operadores. Se ha optado por una SPA (Single Page Application) moderna.

Una SPA[14] es un tipo de aplicación web que se carga una sola vez en el navegador y actualiza dinámicamente el contenido sin recargar la página completa. En el contexto de este proyecto se ha logrado mediante el uso de React pues permite construir componentes reutilizables y reactivos para mostrar información en tiempo real sobre los vehículos, sus versiones, y su estado de sincronización.

Como gestor de paquetes para el frontend se ha utilizado NPM, gestionando dependencias, librerías y automatizando scripts de desarrollo y build.

También se ha utilizado ciertas librerías de estilos comunes desarrolladas por y para GMV.

5.5.3 Modelado y diseño de sistema

Durante las fases de análisis y diseño se utilizaron diversas herramientas para crear los diagramas conceptuales, de arquitectura y de base de datos.

Se ha utilizado Draw.io como herramienta para la creación de bocetos y brainstorming iniciales de cada diagrama. Por otro lado, para la creación de los diagramas UML: clases, componentes y secuencia, se ha utilizado Astah Professional permitiendo la documentación interna del *backend*, con las entidades y el ciclo de vida de los datos. Para finalizar esta sección, la generación y validación del esquema de base de datos se ha realizado mediante DBDiagram.io diseñando gráficamente tanto las tablas como las relaciones y campos de la base de datos de forma clara y exportable.

5.5.4 Control de versiones y gestión de tareas

El desarrollo se ha realizado de forma iterativa siguiendo la metodología Scrum. Para ello se utilizaron herramientas profesionales tanto para el seguimiento de tareas como para la gestión del código y la integración continua.

Todo el código realizado está en repositorios privados de GIT en concreto con el uso de Bitbucket para versionar todo el código del *backend* y *frontend*, incluyendo la documentación.

Para la gestión de tareas se utilizó Jira con *boards* de Kanban para organizar los sprints, priorizar tareas y registrar registrar el estado de cada funcionalidad. Además, como medio de comunicación con el equipo y la realización de las reuniones y resolución de dudas se utilizó Microsoft Teams, tanto dentro de la empresa como con el tutor de la Universidad.

El uso de estas herramientas ha garantizado un desarrollo ordenado, profesional y alineado con el trabajo real en empresa.

5.5.5 Pruebas, documentación y validación

Durante el desarrollo y validación del sistema se utilizaron varias herramientas para probar, documentar y verificar el funcionamiento del *endpoint* REST desarrollado.

Para la generación automática de documentación de la API se usó Swagger que permite probar los endpoints desde el navegador, ver esquemas de respuesta y explorar la API de forma interactiva.

Como herramienta de testing de las APIs se utilizó Postman para ejecutar pruebas funcionales, validar casos de error y simular flujos completos.

5.5.6 Conclusión

El conjunto de herramientas y tecnologías utilizadas ha permitido desarrollar un sistema perfectamente alineado con las prácticas profesionales actuales en el ámbito del desarrollo de software para sistemas distribuidos.

Cada decisión tecnológica ha sido tomada en base a criterios de compatibilidad con el entorno de GMV, escalabilidad futura del sistema y facilidad de mantenimiento. Asimismo, el uso de herramientas profesionales para modelado, pruebas y documentación ha permitido mantener la calidad técnica del proyecto desde la fase de análisis hasta la entrega final.

Capítulo 6

Soluciones y Estado del Arte

6.1 Introducción

En este capítulo se analizan las soluciones existentes y las aproximaciones previas al problema abordado en este Trabajo Fin de Grado: la trazabilidad de versiones de contenido en entornos distribuidos, concretamente en flotas de transporte inteligente.

El objetivo es contextualizar la propuesta dentro del panorama actual, identificar posibles referentes o aproximaciones similares y justificar la necesidad de una solución adaptada al entorno real de GMV. Para ello, se han revisado herramientas utilizadas en otros sectores, soluciones genéricas y métodos aplicados en sistemas similares, tanto desde el punto de vista técnico como funcional.

6.2 Soluciones en el ámbito de desarrollo software

En el ámbito del desarrollo de software, existen numerosas herramientas de control de versiones como Git, SVN o Mercurial, utilizadas para gestionar cambios en el código fuente. Estas soluciones permiten mantener un historial de versiones, comparar estados y recuperar versiones anteriores, pero están orientadas exclusivamente al control de archivos de texto o binarios, no al control operativo de contenido desplegado en equipos físicos distribuidos.

Algunas herramientas de CI/CD (Integración y entrega continua), como Jenkins, GitHub Actions, GitLab CI/CD, incluyen mecanismos para verificar despliegues, pero se centran en entornos controlados de servidores, no en entornos con dispositivos como los OBU, donde la conexión puede ser intermitente y los dispositivos tienen comportamiento autónomo.

Por tanto, aunque estos sistemas comparten conceptos clave (comparación de versiones, sincronización, despliegue), no resultan aplicables directamente al problema operativo de una flota de transporte que gestiona contenido no de código, sino operacional y dependiente del estado real del equipo físico.

6.3 Soluciones en el sector Transporte

En el sector del transporte, algunas plataformas ITS comerciales como **INIT[15], Trapeze[16] o TransTrack[17]** ofrecen soluciones avanzadas para gestión de flota, configuración de equipos embarcados y planificación de servicio. Sin embargo, la mayoría de estas plataformas están centradas en la planificación de rutas, la gestión operativa o la monitorización en tiempo real, y no ofrecen trazabilidad detallada sobre el estado del contenido en cada vehículo, especialmente a nivel de comparación entre versión esperada y real.

Además, muchas de estas plataformas se comportan como sistemas cerrados, con capacidades limitadas de personalización o integración con flujos internos como los que utiliza GMV, lo que refuerza la necesidad de una solución a medida.

En entornos similares (como gestión de contenidos multimedia distribuidos), algunas plataformas permiten ver si un dispositivo ha recibido contenido, pero en la mayoría de los casos:

- No hay validación de que el contenido se haya aplicado correctamente.
- No hay comparación con una versión central o esperada.
- La trazabilidad es parcial o no automatizada (Caso inical de GMV).

6.4 Alternativas internas en GMV

Dentro del ecosistema de GMV existen componentes que forman parte del proceso de actualización, como ArchivosOBU o el TransferManager.

Sin embargo, estos componentes no incluyen una funcionalidad de trazabilidad completa. Si bien el Transfer Manager puede registrar el estado de las transferencias, no compara la versión instalada con la que se esperaba ni expone esta información en una interfaz operativa de consulta por vehículo.

Por tanto, actualmente no se cuenta con una solución a este problema y este TFG surge como una nueva solución.

6.5 Justificación de la solución

El análisis del estado del arte demuestra que no existe actualmente una solución específica que cubra de forma directa el problema abordado en este proyecto dentro del contexto de transporte inteligente embarcado. Por tanto, el sistema desarrollado representa una aportación original, necesaria y útil, tanto a nivel técnico como operativo.

A partir del análisis anterior, se justifica la necesidad de una herramienta específica que cubra las carencias detectadas como se indica en el cuadro 6.1.

Carencia	**Solución propuesta**
No existe trazabilidad centralizada	El sistema registra y presenta el estado de cada OBU respecto a cada campaña.
No se automatiza la comparación de versiones	El backend compara versiones esperadas y reales, e informa de diferencias.
No hay interfaz de consulta operativa	Se desarrolla un frontend claro para visualizar el estado de la flota.

Cuadro 6.1: Carencia identificada y Solución propuesta

La solución propuesta no pretende reemplazar sistemas existentes, sino complementarlos con una capa de trazabilidad, alineada con las necesidades reales detectadas durante las prácticas en GMV, y diseñada con visión de escalabilidad y futura integración completa.

Parte III

Desarrollo del Sistema

Capítulo 7

Análisis

7.1 Flujo actual del sistema

En el sistema actual gestionado por GMV, el proceso de actualización de contenidos para los OBU sigue varias etapas clave:

- **Generación de contenido:** Tras una modificación de los contenidos realizada por el usuario en el SIU, un servicio llamado *SoaBasicContentManager*, conocido como el *backend* del SIU, se encarga de registrar este cambio en la base de datos utilizada.
- **Creación de los ficheros a distribuir:** Cada cierto tiempo configurado, una hebra temporizadora del servicio *ArchivosOBU* consulta la base de datos para generar los archivos necesarios a transferir al OBU, organizándolos en un árbol de directorios lógico predefinido. que sigue la siguiente estructura:
  - Fleet
    - Package Type
  - SubFleet
    - FleetId
      - Package Type
  - Particular
    - Bus Tdma
      - Package Type

  Esta estructura jerárquica permite distribuir los archivos de forma eficiente y personalizada. Si existen configuraciones específicas para determinados vehículos, ya sea a nivel de flota, subflota o unidad individual, estos recibirán únicamente los archivos correspondientes a su nivel. En ausencia de personalización, se utilizarán los archivos definidos en el nivel general (Fleet). El nodo Package Type representa el tipo de contenido a distribuir, como configuraciones de líneas, rutas, archivos de audio, video, entre otros. Por otro lado, bus tdma se refiere a un identificador utilizado para los buses.

Cada carpeta de tipo Package Type contendrá los archivos comprimidos que deben enviarse al bus, junto con un fichero de control en formato texto que sigue la siguiente estructura:

```
Version=NUMEROVERSION
ArchivoComprimidoAEnviar1=NUMEROVERSION
ArchivoComprimidoAEnviar2=NUMEROVERSION
```

La línea Version indica la versión general del paquete, mientras que cada entrada ArchivoComprimidoAEnviarX especifica el nombre del archivo comprimido junto con su versión individual. Este fichero facilita la trazabilidad y sincronización de los contenidos distribuidos a cada unidad.

- **Distribución mediante Transfer Manager:** El componente *Transfer Manager* se encarga de programar y ejecutar la transferencia de estos ficheros a los equipos embarcados.
- **Confirmación post-actualización:** Una vez finalizada la instalación de los archivos en el *OBU*, el propio equipo embarcado genera un archivo de retorno que contiene información sobre la versión instalada. Estos archivos son recogidos y almacenados en un directorio central por el *Transfer Manager* en el que se crean nuevos archivos de texto temporales con nombre:

```
packageType#busTdma_fleetId_randomId
```

Estos archivos de texto contendrán el mismo contenido que el mostrado al inicio de esta página pero referidos al contenido **real** que tiene el bus.

En la figura 7.1 se puede ver el flujo actual de la gestión.

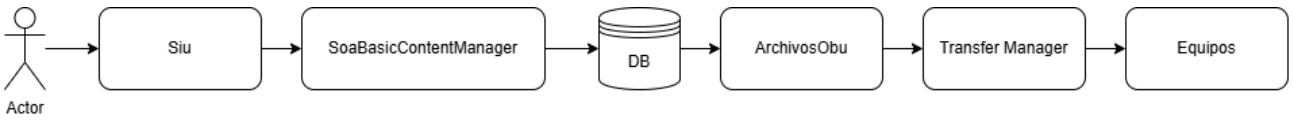


Figura 7.1: Flujo actual de la gestion de contenidos

Actualmente estos archivos generados han de ser consultados manualmente y el sistema propuesto actúa como un verificador automático, conectándose a las rutas donde *ArchivosOBU* y *Transfer Manager* almacenan los archivos generados y devueltos respectivamente. Compara ambas versiones (planificada y actual), y almacena los resultados por vehículo. Esto permite tener una vista global del estado de actualización de la flota, generar alertas e incluso llegar a facilitar reenvíos en caso de errores.

7.2 Identificación de necesidades

Durante el análisis funcional, se identificaron las siguientes necesidades no cubiertas por el sistema actual:

Estas necesidades surgen de un análsis del flujo actual, de entrevistas con el equipo de GMV y de la observación de problemas reales durante el uso del sistema.

Necesidad Detectada	Implicación Operativa
Verificar automáticamente si un OBU ha recibido contenido	Reduce errores manuales y aumenta la fiabilidad
Comparar versiones esperadas vs instaladas	Detecta inconsistencias y evita errores en servicio o a la hora de desplegar buses.
Centralizar la información de actualización	Facilita la supervisión de los estados de actualización de las flotas.
Visualizar el estado por vehículo en una UI	Aumenta la eficiencia de los técnicos y evita la inspección manual.
Tener a disposición distintos filtros por fecha, estado...	Mejora la toma de decisiones operativas.

Cuadro 7.1: Identificación de necesidades

7.3 Integración en sistema complejo existente

Uno de los retos más relevantes de este proyecto ha sido diseñar una solución que pueda integrarse de forma segura, coherente y realista en un sistema complejo y maduro ya existente, como es el ecosistema de transporte inteligente de GMV.

No se trata simplemente de desarrollar una nueva funcionalidad, sino de incorporarla en un entorno que ya funciona en producción, con múltiples sistemas interconectados, procesos establecidos, responsabilidades distribuidas entre equipos y requisitos técnicos bien definidos.

Integrar une nueva funcionalidad en un sistema de este tipo implica afrontar desafíos como:

- Evitar romper el funcionamiento de los componentes existentes.
- Respetar contratos funcionales y estructuras ya desplegadas.
- Adaptarse a tecnologías, convenciones y estándares internos.
- Minimizar los puntos de acoplamiento para facilitar mantenibilidad.
- Garantizar que lo integrado sea entendible, útil y sostenible a largo plazo.

En este proyecto, se ha adoptado un enfoque específico que aborda estos retos de forma profesional. Para ello, lo primero es analizar las características con las que cuenta el sistema, posteriormente y teniendo en cuenta lo analizado, hay que crear una estrategia de integración que pueda aplicarse manteniendo los puntos anteriormente nombrados en este mismo apartado.

7.3.1 Características del sistema a tener en cuenta:

Actualmente, el ecosistema de GMV está compuesto por múltiples microservicios, servicios y componentes, así como sistemas de monitorización, gestión de flota, trazabilidad, y comunicación de hardware embarcado. A esto hay que añadirle los propios equipos embarcados y distintos procesos de seguridad y privacidad asociados.

Uno de los principios fundamentales que ha guiado el desarrollo de este proyecto ha sido la necesidad de diseñar una solución que pueda integrarse de forma no intrusiva en un sistema complejo y consolidado, sin alterar su funcionamiento ni comprometer su estabilidad. En el contexto de GMV, donde gran parte de los servicios están en operación continua y en entornos productivos sensibles, cualquier nueva funcionalidad debe respetar la arquitectura existente, sin introducir riesgos ni modificar componentes críticos.

En primer lugar, se ha asegurado que la herramienta desarrollada no interfiera en el flujo operativo actual de actualización de contenidos, que continúa siendo gestionado exclusivamente por ArchivosOBU y Transfer Manager. Estos componentes mantienen sus responsabilidades intactas: ArchivosOBU genera los ficheros de

contenido a partir de cambios realizados en el SIU, y Transfer Manager los distribuye a los OBU según una lógica ya probada y estable.

Además, la solución propuesta no requiere modificaciones en los dispositivos embarcados (OBU), ni cambios en la forma en la que estos equipos generan los archivos de confirmación tras una actualización. Esto elimina la necesidad de actualizaciones en software embarcado, despliegues masivos o validaciones en campo, que implicarían un coste operativo y un riesgo elevado para los operadores.

Otro aspecto esencial es que el sistema de trazabilidad se alimenta exclusivamente de los archivos ya generados por los sistemas actuales. Estos archivos, tanto los ZIP generados por ArchivosOBU como los logs devueltos por los OBU y almacenados por Transfer Manager, se ubican en estructuras de carpetas y rutas de red bien definidas. El sistema propuesto accede de forma pasiva a estas ubicaciones, sin necesidad de modificar los procesos que las generan, ni de establecer canales nuevos de comunicación o integración directa.

Por último, toda la solución ha sido diseñada para alinearse completamente con los patrones de desarrollo, despliegue y operación ya adoptados por el equipo técnico de GMV. Esto incluye el uso de tecnologías estándar como .NET, MySQL, React, Swagger, Bitbucket y Jira, así como la adopción de buenas prácticas de desarrollo como separación de responsabilidades, desacoplamiento de componentes, y uso de herramientas comunes para testing y documentación.

En conjunto, la solución se comporta como una capa adicional de verificación, completamente autónoma, que funciona en paralelo al sistema productivo. No interfiere en su ejecución, no modifica sus resultados, y no introduce nuevas dependencias en el flujo principal. Este enfoque garantiza una integración respetuosa, segura y coherente, compatible con un entorno empresarial real y adecuada a los requisitos de un sistema en producción.

7.3.2 Equilibrio entre integración y viabilidad:

Uno de los principales retos de este proyecto ha sido alcanzar un equilibrio entre el contexto profesional de integración en un sistema complejo y las limitaciones naturales de un Trabajo Fin de Grado, tanto en tiempo como en recursos.

Este equilibrio ha sido considerado en todas las fases del desarrollo: desde el análisis inicial y la elección de tecnologías, hasta las decisiones de arquitectura y priorización funcional. El objetivo ha sido ofrecer una solución técnicamente útil y realista, que pueda desplegarse en un entorno profesional, pero que al mismo tiempo sea abordable por un estudiante en un periodo de tiempo limitado.

Las decisiones que se han tomado para asegurar la viabilidad han sido:

- Contar con un tiempo de desarrollo limitado a 3 meses para generar un MVP funcional con validación parcial
- La creación de un módulo acoplable o desplegable de manera independiente.
- Evitar la modificación de dependencias críticas como *ArchivosOBU* o el *TransferManager* creando así un acceso pasivo a archivos generados por los sistemas ya existentes
- Uso del stack corporativo (.NET, MySQL, React,etc.)
- Creación del proyecto con la escalabilidad futura en mente, teniendo un código modular y bien documentado para una fácil migración o crecimiento posterior.

Este subproyecto demuestra una de las habilidades más valoradas en entornos reales: la capacidad de introducir mejoras en sistemas maduros sin romper su equilibrio. En un entorno empresarial real no basta con que una solución ”funcione”; debe encajar, ser comprensible para otros desarrolladores, y coexistir con las herramientas, equipos y flujos existentes.

Desde el punto de vista académico, esto también refleja una madurez en el enfoque: no se trata solo de aplicar conocimientos técnicos, sino de pensar como ingeniero/a de software profesional, priorizando la viabilidad, el impacto real y la mantenibilidad de lo que se construye.

7.4 Requisitos

Este apartado recoge todo lo que el sistema debe hacer (funcional) y cómo debe comportarse o estar construido (no funcional), resultado del análisis previo, entrevistas con el equipo técnico, observación del sistema real, y objetivos marcados.

Aunque el desarrollo del proyecto se ha gestionado mediante el marco de trabajo Scrum, utilizando historias de usuario y tareas técnicas para la planificación de sprints y la gestión del backlog, en esta memoria se han estructurado los requisitos según el enfoque clásico: requisitos funcionales y no funcionales. Esta organización permite una mejor trazabilidad documental y facilita la conexión entre los objetivos del sistema, su diseño y las pruebas realizadas.

7.4.1 Requisitos funcionales

Los requisitos funcionales definen el comportamiento observable del sistema. Son las capacidades y servicios que debe proporcionar al usuario o a otros componentes del entorno. Se han encontrado cuatro casos de usos principales que se pueden ver en los cuadros 7.2, 7.3, 7.4 y 7.5. Además en la figura 7.2 se puede ver el diagrama de casos de uso correspondiente al proyecto.

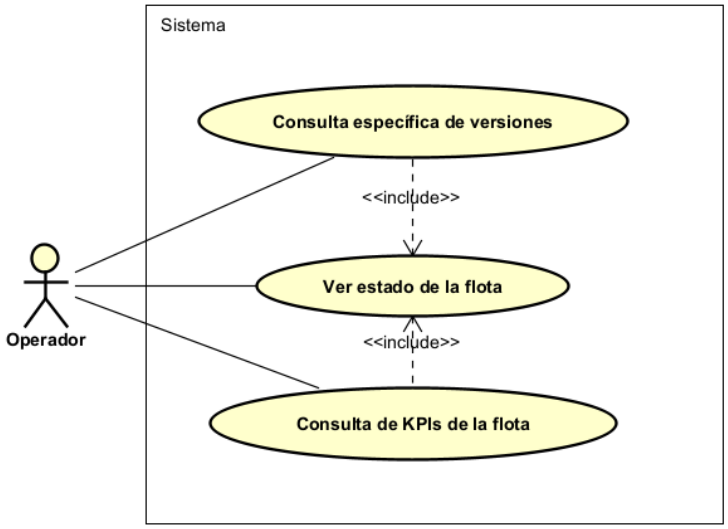


Figura 7.2: Diagrama de casos de uso

7.4.2 Requisitos no funcionales

Los requisitos no funcionales definen restricciones, cualidades y características técnicas que debe cumplir el sistema para ser viable en producción y mantenible en el tiempo. Los requisitos no funcionales identificados son los siguientes:

- **RNF01:** El sistema debe estar desarrollado en el stack tecnológico compatible con GMV: .NET, MySQL, React, etc.
- **RNF02:** El sistema debe integrarse sin alterar los componentes actuales (ArchivosOBU, Transfer Manager, OBU).
- **RNF03:** El *backend* debe exponer una API REST documentada, integrada en el actual SIU.

- **RNF04:** La lógica debe estar desacoplada del backend principal para facilitar su mantenimiento o extracción futura.
- **RNF05:** El tiempo de lectura de los archivos no debe superar los 120 s en condiciones normales (escalable).
- **RNF06:** El tiempo de comprobación por vehículo no debe superar el segundo en condiciones normales (escalable).
- **RNF07:** El sistema debe poder analizar flotas de más de 1000 vehículos sin degradación significativa.
- **RNF09:** El sistema debe mantener un log de errores accesible para diagnósticos posteriores.
- **RNF10:** La implementación deberá ser dividida en 3 partes, un lector de los archivos generados que inserte a BD las versiones, un comprobador automático de versiones esperadas y reales y un frontend integrado con el actual Gestor de Contenidos.

Título	**Resgistrar versiones generadas**
Actor	Sistema interno
Descripción	Tras un tiempo asignado en configuración, el sistema detecta que se ha generado un nuevo conjunto de archivos de contenido y registra la versión esperada y real de cada contenido para cada vehículo.
Precondiciones	PRE-1. Debe existir una nueva campaña o contenido pendiente de distribución. PRE-2. ArchivosOBU ha generado correctamente los ficheros.
Postcondiciones	POST-1. Se almacena en la base de datos las versiones asociadas a cada vehículo.
Flujo Normal	1. Tras un tiempo designado en la configuración, el sistema detecta si hay un nuevo conjunto de archivos o se han actualizado. 2. Se analiza la estructura de las carpetas y se obtienen los nuevos contenidos o versiones actualizadas. 3. Se registran estas versiones en la base de datos.
Excepciones	2.1 Si no se detectan archivos o hay carpetas vacías, el sistema no realiza ninguna acción. Tampoco realiza acciones si todos los archivos siguen igual, es decir, no ha habido actualizaciones de contenido. 2.2 Si los archivos están corruptos o no siguen la estructura esperada, se registra un error y se descarta el procesamiento.

Cuadro 7.2: CU1 - Registrar versiones generadas

Título	**Comparación de versiones**
Actor	Operador
Descripción	El actor solicita ver el estado de sincrionización de los vehículos
Precondiciones	PRE-1. Deben existir registros de versiones esperadas y reales en base de datos PRE-2. El actor deberá estar autenticado en el sistema y tener los permisos correspondientes a este módulo.
Postcondiciones	POST-1. El actor puede ver el estado de sincronización de los vehículos
Flujo Normal	1. El actor solicita ver el estado de sincronización de los vehículos, pudiendo elegir entre varios filtros 2. El sistema compara las versiones esperadas con las reales y detecta inconsistencias. 3. El sistema muestra los datos al actor siguiendo los filtros seleccionados e informa de las inconsistencias detectadas.
Excepciones	3.1 El sistema no encuentra resultados para los filtros → se muestra una tabla vacía con mensaje informativo.

Cuadro 7.3: CU2 - Comparar Versiones

Título	**Consulta específica de versiones**
Actor	Operador
Descripción	El actor solicita ver las versiones específicas de los vehículos
Precondiciones	PRE-1. Deben existir registros de versiones esperadas y reales en base de datos PRE-2. El actor deberá estar autenticado en el sistema y tener los permisos correspondientes a este módulo.
Postcondiciones	POST-1. El actor puede ver las versiones específicas de los vehículos
Flujo Normal	1. El actor solicita ver las versiones específicas de los vehículos 2. El sistema muestra el numero de versión y tipo de paquete esperado y el contenido en el bus por cada tipo de paquete necesario.
Excepciones	3.1 El sistema no encuentra resultados → se muestra una tabla vacía con mensaje informativo.

Cuadro 7.4: CU3 - Consulta específica de versiones

Título	**Consulta de KPIs de la flota**
Actor	Operador
Descripción	El actor solicita consultar los KPIs de la flota
Precondiciones	PRE-1. Deben existir registros de versiones esperadas y reales en base de datos PRE-2. El actor deberá estar autenticado en el sistema y tener los permisos correspondientes a este módulo.
Postcondiciones	POST-1. El actor puede ver información estadística sobre la flota
Flujo Normal	1. El actor solicita consultar los KPIs de la flota 2. El sistema muestra la información estadística correspondiente
Excepciones	3.1 El sistema no encuentra resultados → se muestra una tabla vacía con mensaje informativo.

Cuadro 7.5: CU4 - Consulta de KPIs de la flota

Capítulo 8

Diseño

El presente capítulo describe en detalle el diseño técnico del sistema de trazabilidad de versiones de contenido desarrollado durante este Trabajo Fin de Grado. A partir de los requisitos definidos en el análisis previo, se ha diseñado una arquitectura modular, flexible y alineada con los estándares tecnológicos actuales del entorno de GMV.

El diseño del sistema se ha estructurado teniendo en cuenta los siguientes principios:

- **Modularidad:** cada componente tiene una responsabilidad clara y puede evolucionar de forma independiente.
- **Escalabilidad:** el sistema puede crecer en número de vehículos, campañas o funcionalidades sin pérdida de rendimiento.
- **Integrabilidad:** la solución puede desplegarse como microservicio o integrarse en un backend existente sin romper la arquitectura actual.
- **Mantenibilidad:** el código y los componentes están documentados y organizados para facilitar su evolución y soporte por parte de otros equipos técnicos.

A lo largo del capítulo se describen los aspectos clave del diseño, incluyendo la arquitectura general, el modelo de datos, la lógica de negocio, la interfaz de usuario, y los servicios REST que permiten la interacción con el sistema.

8.1 Alternativas de arquitectura evaluadas

Antes de definir la arquitectura final del sistema, se analizaron diversas alternativas con el objetivo de encontrar la solución que ofreciera el mejor equilibrio entre integrabilidad, viabilidad técnica, esfuerzo de desarrollo y alineación con el sistema actual.

A continuación se detallan las tres alternativas principales que se evaluaron:

8.1.1 Microservicio Independiente

La primera opción evaluada fue desarrollar la solución como un microservicio independiente. Esta alternativa implicaba diseñar un servicio completamente desacoplado del resto de la infraestructura, con su propio backend, base de datos, endpoints REST documentados, sistema de control de versiones y, opcionalmente, su propia interfaz gráfica. Desde el punto de vista arquitectónico, esta opción representaba una solución moderna y alineada con las tendencias actuales en desarrollo distribuido, y además se ajustaba bien al enfoque basado

en microservicios que ya existe en el entorno técnico de GMV. Al tratarse de un componente autónomo, esta solución ofrecía ventajas significativas en términos de escalabilidad, posibilidad de reutilización en otros contextos, independencia de despliegue y menor impacto sobre el resto del sistema. Sin embargo, el coste asociado al diseño, implementación y despliegue de un microservicio completo resultaba elevado dentro del contexto temporal y técnico de un TFG. Desarrollar un microservicio desde cero implica definir y configurar toda la infraestructura asociada (entorno de ejecución, autenticación, integración en CI/CD, monitorización, logging, etc.), lo que suponía una carga adicional difícilmente asumible sin desviar el foco del proyecto hacia aspectos de infraestructura más que funcionales. Además, este enfoque complicaba la validación final con datos reales, al requerir una integración más extensa con el ecosistema en producción.

8.1.2 Módulo integrado en backend existente

La segunda opción considerada consistía en integrar la funcionalidad directamente dentro del backend de un sistema ya existente, concretamente en el SIU (Sistema de Información al Usuario), que es uno de los componentes clave del ecosistema GMV para la gestión de campañas y contenidos. Esta alternativa resultaba especialmente atractiva desde el punto de vista de la eficiencia: permitía reutilizar la infraestructura ya desplegada, aprovechar los mecanismos existentes de autenticación, permisos, gestión de sesiones, configuración de entornos y acceso a bases de datos, y reducir el esfuerzo requerido en la fase de despliegue. Asimismo, facilitaba una validación rápida de la funcionalidad desarrollada y la incorporación inmediata en el flujo de trabajo del usuario final. No obstante, este enfoque tenía también sus limitaciones. La más relevante era el alto grado de acoplamiento que generaba con el backend principal, lo cual comprometería la mantenibilidad futura del sistema y dificultaría su evolución como módulo independiente. Además, existía el riesgo de alterar la lógica del SIU en producción, introducir dependencias técnicas difíciles de aislar, o generar un solapamiento entre responsabilidades funcionales que no estaba alineado con los principios de responsabilidad única.

8.1.3 Módulo reutilizable integrado con separación por capas

Como tercera opción se planteó un enfoque intermedio, basado en el desarrollo de una lógica completamente modular, diseñada desde el inicio para ser desacoplada, pero que pudiera integrarse temporalmente dentro de un backend existente para facilitar su despliegue y validación. Esta alternativa permitía trabajar con una arquitectura clara, basada en capas (acceso a datos, lógica de comparación, servicios REST), utilizando tecnologías compatibles con el ecosistema de GMV (C#, .NET, PostgreSQL, React) y respetando las convenciones internas de desarrollo y estilo. La solución se diseñó de forma que pudiera empaquetarse e integrarse en el backend del SIU como un módulo más, pero manteniendo sus dependencias y servicios bien delimitados, con el objetivo de que, en el futuro, pudiera ser extraída y desplegada como microservicio con un esfuerzo reducido. Este enfoque ofrecía el mejor equilibrio entre los factores evaluados: reducía el esfuerzo inicial de despliegue, permitía validar funcionalmente el sistema dentro del entorno de prácticas, y a su vez garantizaba una base técnica sólida para su futura evolución como componente independiente. Además, encajaba de forma natural en los tiempos y recursos disponibles en el desarrollo de un TFG, al evitar complicaciones logísticas y técnicas asociadas a un microservicio completo, sin renunciar a los principios de diseño modular y mantenible.

8.1.4 Arquitectura elegida

Tras evaluar estas tres alternativas, se optó por implementar la solución siguiendo el enfoque modular de integración controlada, priorizando la coherencia con el sistema actual, la viabilidad académica y el potencial de escalabilidad futura. Esta decisión ha permitido centrar el desarrollo en la funcionalidad principal, la trazabilidad de versiones, sin desatender los aspectos clave de integrabilidad, sostenibilidad y alineación con el entorno profesional en el que se enmarca este proyecto.

8.2 Diseño

La arquitectura del sistema diseñado en este Trabajo Fin de Grado se ha definido siguiendo un enfoque modular y desacoplado, con el objetivo de facilitar su integración en un entorno complejo y en producción como el de GMV. A diferencia de una arquitectura de microservicios clásica, se ha optado por un enfoque más pragmático y alineado con los estándares internos de la empresa, basado en la separación funcional entre el procesamiento de datos y su exposición a través de servicios ya existentes.

La solución se articula en torno a cuatro grandes componentes:

- **InfoVersionService:** Un servicio de Windows independiente, responsable de realizar el procesamiento de los datos (lectura de archivos, tratado de los datos y generación de resultados).
- **SoaBasicContentManager:** backend del SIU ya existente que expone endpoints para ser consumidos por el SiuFront. En este caso contará con un nuevo endpoint que permitirá consultar resultados generados gracias al servicio de trazabilidad de InfoVersionService, tratarlos y exponer su información al front.
- **Base de datos:** compartida entre InfoVersionService y el SoaBasicContentManager.
- **SiuFront:** Frontend encargado del sistema de información al usuario que se encarga de todo lo relacionado con los contenidos en las flotas. En este caso, se ha añadido un nuevo componente llamado ”Fleet Status”, en el que se podrá consultar el estado de actualización de toda la flota.

Esta arquitectura garantiza una integración controlada y no invasiva en el sistema actual. El nuevo servicio opera de forma autónoma, ejecutándose en segundo plano como una tarea de sistema que puede programarse periódicamente y configurarse para tener más funcionalidades, como por ejemplo, el borrado de históricos con cierta antigüedad. A su vez, los datos generados (versiones esperadas, versiones reales y comparaciones) se almacenan en una base de datos que sigue el modelo de datos definido para este proyecto, véase la sección 8.4. El backend del SIU ha sido extendido con un nuevo endpoint REST, que se encarga de consultar esa base de datos, tratar los datos y exponer los resultados al frontend, sin que el SIU tenga que hacerse responsable de la lógica de comparación o de la gestión de archivos.

Este diseño ofrece múltiples ventajas desde el punto de vista técnico y organizativo. Por un lado, permite desacoplar la lógica pesada de procesamiento, que podría evolucionar en volumen y complejidad, de los servicios interactivos que deben mantener tiempos de respuesta cortos. Por otro, garantiza una integración segura y progresiva en el ecosistema de GMV: el SIU accede únicamente a datos procesados en un nuevo componente desacopable, mientras que el nuevo servicio puede desplegarse, actualizarse o incluso detenerse sin afectar a la operativa central.

La interfaz de usuario, desarrollada en React, no accede directamente al servicio de Windows, sino que consume el nuevo endpoint del SIU, manteniendo la experiencia de usuario unificada y coherente con el resto de la aplicación. Esto permite aprovechar el sistema de autenticación, permisos y diseño gráfico ya existente, sin duplicar funcionalidades ni introducir nuevas dependencias en el cliente.

Desde el punto de vista del ciclo de vida del software, esta arquitectura también favorece el mantenimiento: las actualizaciones del servicio de procesamiento pueden desplegarse de forma independiente, y el almacenamiento de los resultados en una base de datos relacional permite auditoría, reuso, y análisis posterior sin necesidad de reprocesar archivos.

En resumen, se trata de una arquitectura híbrida y realista, que separa claramente responsabilidades entre procesamiento y presentación, respeta el funcionamiento del sistema actual, y se adapta a las limitaciones temporales y técnicas de un Trabajo Fin de Grado, sin renunciar a la calidad y la profesionalidad en el diseño.

8.3 Patrones de Diseño aplicados

Durante el diseño e implementación del sistema se han aplicado varios patrones de diseño clásicos, ampliamente utilizados en el desarrollo de software moderno. Estos patrones han permitido estructurar el código

de forma más modular, mantenible y extensible, además de facilitar su integración en el ecosistema técnico existente.

A continuación se describen los principales patrones utilizados:

8.3.1 Singleton

El patrón Singleton garantiza que una clase tenga una única instancia accesible globalmente y proporciona un punto centralizado de acceso a ella. Este patrón es útil para clases que representan configuraciones o recursos compartidos[18]. Su estructura se puede ver en la figura 8.1.

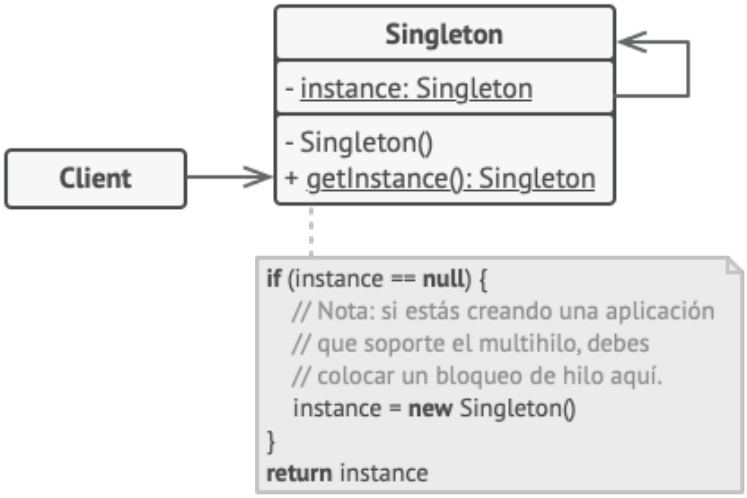


Figura 8.1: Singleton

En cuanto a la aplicación en el proyecto, todas las configuraciones internas del nuevo servicio (InfoVersionService), son almacenadas por medio de la implementación de un singleton clásico con propiedad estática Instance. Almacena parámetros de configuración como rutas de las carpetas utilizadas por el Transfer Manager y ArchivosOBU en la generación de archivos, intervalos de ejecución de la lectura de archivos y opciones de limpieza. Esta instancia se accede desde distintas partes del código sin necesidad de pasarla como dependencia explícita.

8.3.2 Fachada

El patrón Fachada proporciona una interfaz de alto nivel que oculta la complejidad de múltiples subsistemas. Permite al cliente interactuar con una sola entrada simplificada, sin conocer la estructura interna. Reduce el acoplamiento entre cliente y subsistemas, facilita el uso del sistema por parte de otros módulos y mejora la legibilidad del flujo general[19]. Se puede ver su estructura en la figura 8.2.

A nivel de aplicación, se proporciona la fachada *VersionProcessingService* para agrupar múltiples responsabilidades internas como la gestión de carpetas, ejecución de estrategias de lectura de archivos, escritura en base de datos, etc, y exponerlas a través de métodos tan simples como Start() o Stop().

8.3.3 Inyección de dependencias

La Inyección de Dependencias es un patrón de arquitectura que permite suministrar las dependencias de una clase desde el exterior, en lugar de crearlas internamente. Se implementa habitualmente mediante contenedores de inversión de control (IoC), mejorando así el desacoplamiento entre componentes, facilitando el uso de mocks en tests y promoviendo el principio de inversión de dependencias[20]. Su estructura se puede ver en la figura 8.3.

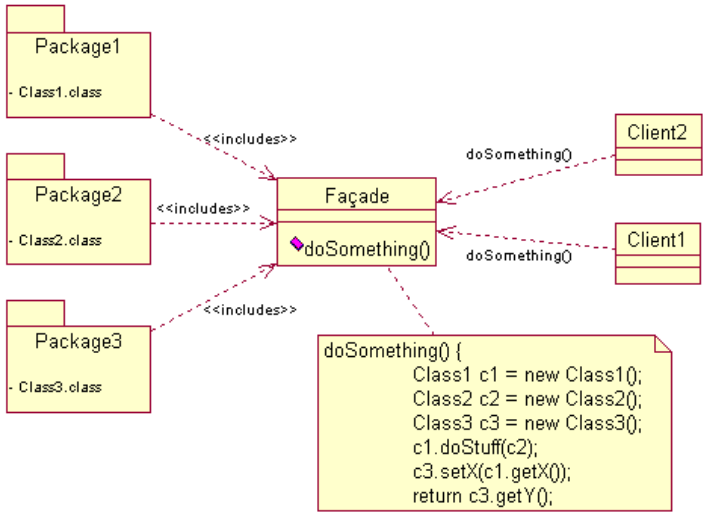


Figura 8.2: Fachada

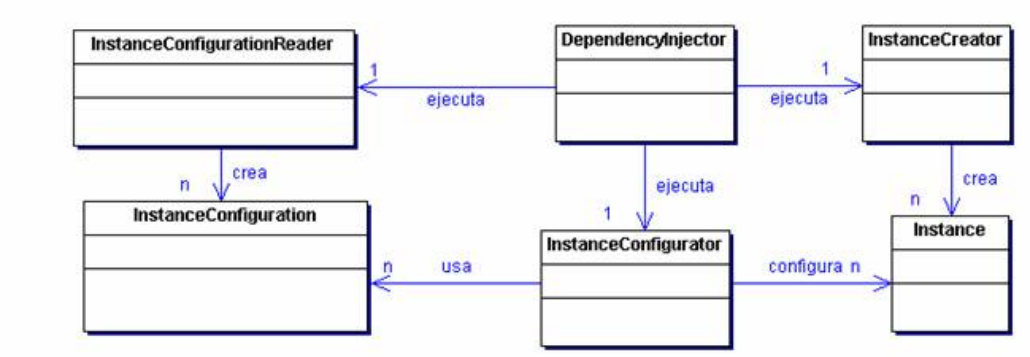


Figura 8.3: Inyección de Dependencias

En el contexto de la aplicación, se ha creado un contenedor de servicios denominado *ServiceCollectionExtension* en el que se registran todas las interfaces e implementaciones. Luego, son inyectadas automáticamente en el resto de clases consumidoras evitando así la necesidad de uns instanciación directa.

8.3.4 Strategy

El patrón Strategy permite definir una familia de algoritmos o comportamientos intercambiables y encapsularlos en clases separadas que comparten una misma interfaz. El cliente delega el comportamiento concreto a una estrategia en tiempo de ejecución, lo que permite modificar dinámicamente la lógica sin cambiar el cliente. Esto elimina bloques de código condicionales extensos, favorece el principio abierto/cerrado permitiendo agregar nuevas estrategias sin modificar las existentes y facilita el testing unitario de cada estrategia individual[21]. Su estructura se puede ver en la figura 8.4.

En el contexto del proyecto, en el servicio *InfoVersionService*, se manifiesta por medio de una interfaz llamada *IFolderProcessor* que define el contrato para los distintos procesadores de carpetas, siendo estas sus implementaciones. Estas implementaciones encapsulan lógicas distintas de navegación de carpetas según el tipo de agrupación (Flota, SubFlota, Particular). El servicio selecciona e invoca estas estrategias en tiempo de ejecución sin tener que conocer sus diferencias internas.

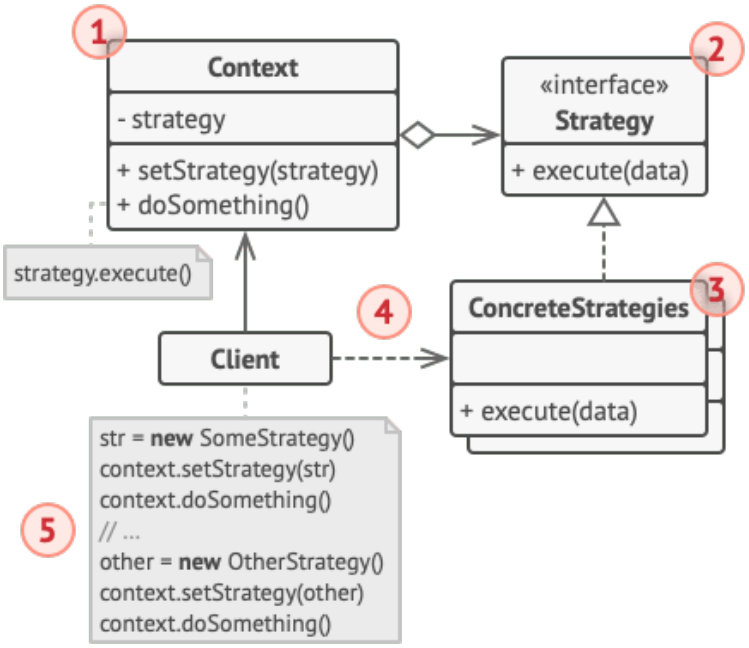


Figura 8.4: Strategy

8.3.5 Scheduled Task

Este patrón se utiliza cuando se necesita ejecutar tareas a intervalos fijos o programados, como limpiezas, verificaciones o sincronizaciones automáticas. Permite la ejecución periódica sin intervención del usuario, encapsula lógica recurrente en módulos reutilizables y mejora la automatización y monitorización del sistema[22].

La aplicación al proyecto se realiza mediante una clase llamada *ScheduledTask* que encapsula toda la lógica de temporización. Esta clase permite crear tareas que se ejecutan cada ciertos intervalos de tiempo. En este caso la lectura de los archivos de versiones generados y el borrado de registros con antigüedad mayor a la configurada. Ambas tareas definidas como callbacks configurables separadas del código principal.

8.3.6 Template

Este patrón define el esqueleto de un algoritmo en una clase base, dejando la implementación de pasos concretos a subclases. La clase abstracta contiene la lógica común, pero permite que ciertos pasos sean personalizados [23].

La estructura general de este patrón puede verse en la figura 8.5.

La clase abstracta *FolderProcessorBase* implementa el método Process(), que define el flujo general del procesamiento (recorrido de carpetas, identificación de OBUs, procesado de paquetes, registro de resultados). Las subclases *(FleetProcessor, etc.)* redefinen métodos como *GetFoldersToProcess()* y *GetBusIds()* según el contexto, permitiendo adaptar el flujo a distintos tipos de entrada sin alterar la lógica general.

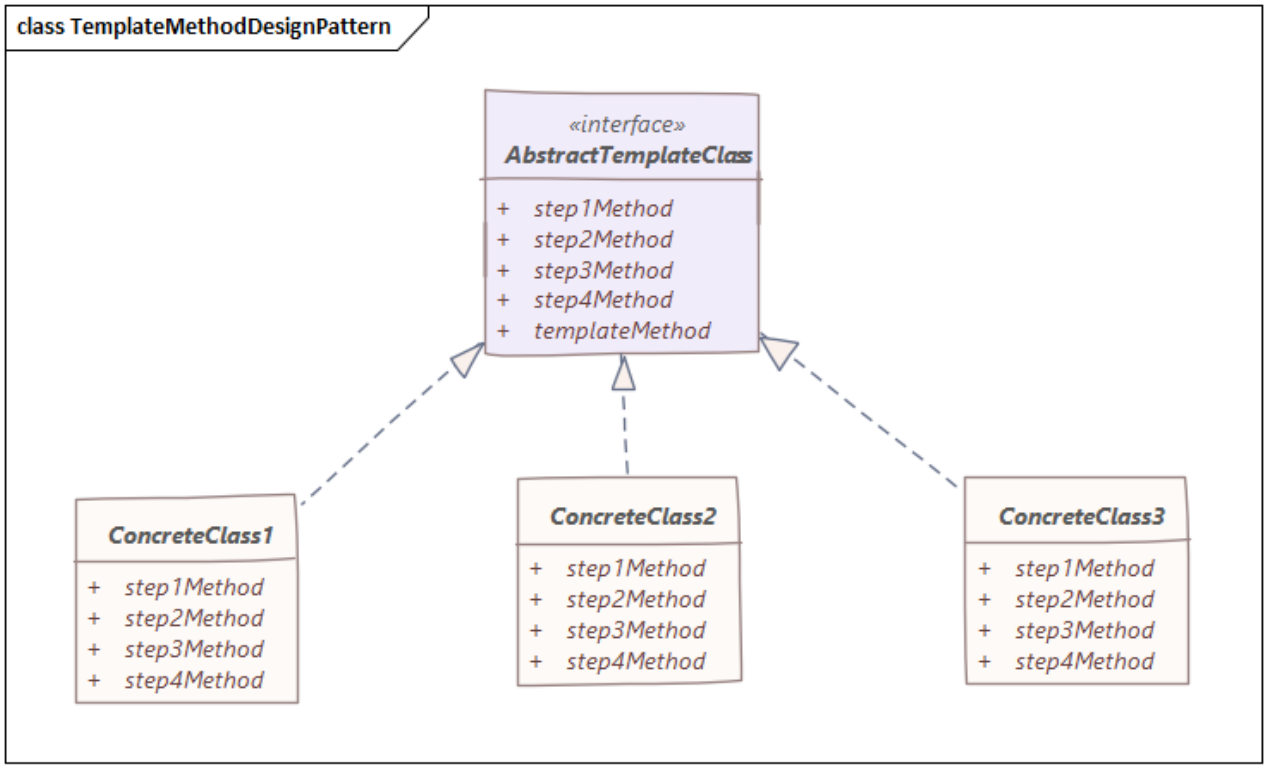


Figura 8.5: Template

8.4 Modelado de datos

El sistema diseñado para la trazabilidad de versiones de contenidos requiere almacenar de forma persistente la información que fluye durante todo el proceso: versiones generadas por los sistemas emisores (ArchivosOBU) y las versiones detectadas por los equipos embarcados (a través de Transfer Manager)

Para ello, se ha definido un modelo de datos relacional centrado en tres entidades principales: Bus, PackageVersion y File. Estas entidades se relacionan entre sí mediante identificadores de vehículo, y paquete, permitiendo consultar el estado histórico y actual de cada unidad embarcada respecto a una determinada actualización. Se puede consultar el modelo de datos en la figura 8.6

Antes de analizar cada clase con sus atributos cabe destacar que todos ellos empiezan por una letra que indica su tipo primitivo. Esto se hace así por convenio en GMV.

Las clases con las descripciones de sus atributos son las siguientes:

Bus: Clase que representa los datos de los buses.

- iIdAutobus: Se refiere al identificador interno de GMV respecto al bus.
- sSideCode: Equivale a un identificador del bus conocido por el cliente.
- sMatricula: Matrícula del vehículo.

BusPackage: Clase que representa los datos relacionados con los paquetes generales de versiones.

- iIdPackage: Identificador del paquete.
- iIdAutobus: Identificador del bus que al que pertenece este paquete.
- packageType: Se refiere al tipo de paquete de contenidos, es decir el tipo de ficheros y contenidos que almacena dicho paquete. A día de hoy, GMV consta con Topología para todo lo relacionado a rutas, líneas, paradas, trayectos, etc. Ecodriving para todo lo relacionado a la conducción eco, Pis_configuration para las configuraciones internas del bus y Pis_data para distintos contenidos como mensajes, audios, imágenes, etc.
- bInBus: indica si el paquete se refiere a una version esperada (False) o Actual (True).
- iPackageVersion: Versión del paquete
- dtFechaRegistro: Fecha en la que InfoVersionService procesó el paquete a base de datos.

File: Hace referencia a los ficheros, es decir contenidos, internos de cada paquete. Por ejemplo, una línea en concreto o una de las muchas imágenes que se pueden mostrar.

- sName: Nombre del fichero/contenido.
- iIdPackage: Paquete al que pertenece dicho contenido.
- iVersion: Versión del fichero/contenido.

Este modelo de datos permite almacenar de forma estructurada el histórico de versiones por vehículo, hacer trazabilidad por campaña, comparar datos desde distintas fuentes, y ofrecer información confiable al operador sobre el estado de la flota. Su diseño relacional permite consultas eficientes, control de integridad referencial y posibilidad de extensión futura. Por otro lado, este modelo de dominio es un equivalente al utilizado para la base de datos pues los datos se tratan de igual manera.

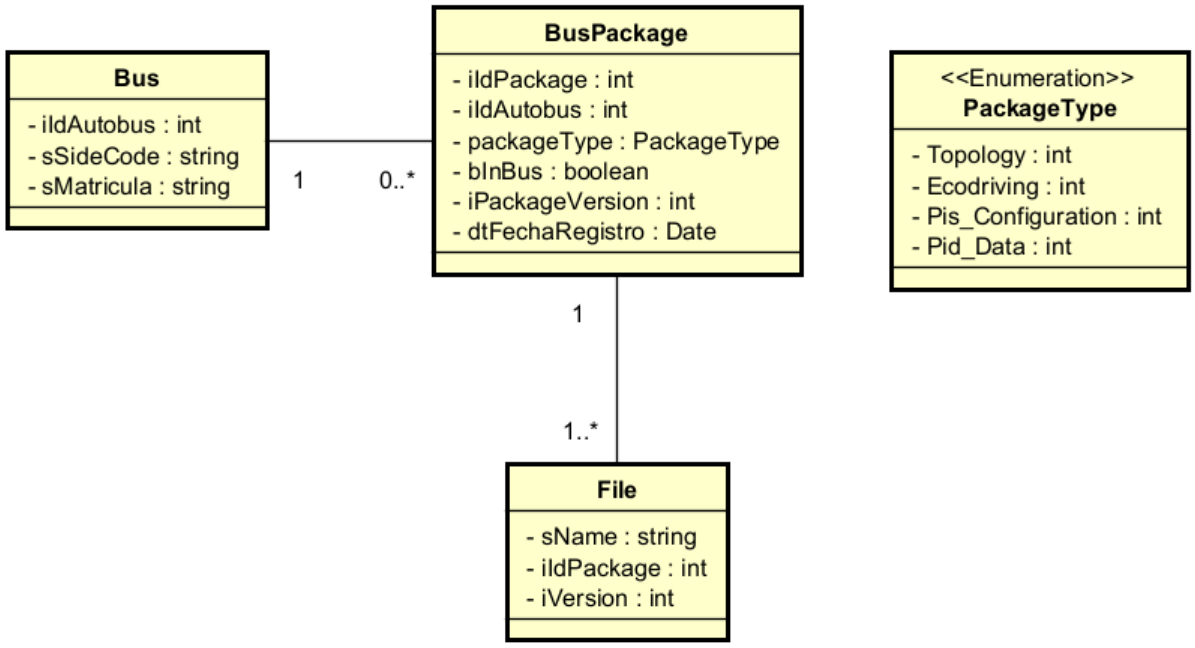


Figura 8.6: Modelado de Datos

8.5 Diseño de InfoVersionService

El único componente del sistema cuyo diseño se analiza en profundidad es InfoVersionService, ya que constituye el núcleo funcional (core) de la solución propuesta. Este servicio es responsable de procesar los datos brutos, aplicar la lógica de negocio y generar los resultados que permitirán determinar el estado de sincronización de los vehículos. En contraste, las extensiones realizadas tanto en el SIU Frontend como en el SoaBasicContentManager son modificaciones mínimas, limitadas a la incorporación de nuevos componentes de interfaz o métodos auxiliares que se encargan exclusivamente de consultar y visualizar la información generada por InfoVersionService. Estas piezas actúan como consumidores de datos, sin aportar complejidad adicional desde el punto de vista arquitectónico o algorítmico.

8.5.1 Descripción General

InfoVersionService es un servicio de backend desarrollado como una aplicación de tipo Windows Service, diseñado para ejecutarse de forma autónoma y periódica en segundo plano. Su propósito principal es analizar las versiones de contenido destinadas a los vehículos, comparar dichas versiones con las efectivamente instaladas, y generar registros estructurados que posteriormente serán consultados por el SIU para su visualización y control.

Este servicio constituye el núcleo funcional (core) de la solución propuesta. A diferencia de otros componentes del sistema que se limitan a consumir y mostrar los datos, InfoVersionService es responsable de implementar la lógica de negocio central: lectura de archivos, interpretación de estructuras, aplicación de estrategias de procesamiento, persistencia de datos y evaluación de sincronización. Por ello, su diseño se ha abordado con especial atención a la separación de responsabilidades, el uso de patrones de diseño reutilizables y la extensibilidad futura.

El servicio se estructura siguiendo una arquitectura multicapa, que divide claramente las responsabilidades entre los distintos módulos del sistema. La lógica se organiza en paquetes que responden a distintas capas funcionales: Application, Domain, Infrastructure, Configuration, y ServiceHost. Esta organización favorece el

mantenimiento del sistema, permite testear cada capa de forma aislada y facilita una futura migración hacia arquitecturas más distribuidas si el sistema crece.

La ejecución del servicio se basa en un mecanismo de tareas programadas (ScheduledTask 8.3.5), que lanza automáticamente dos procesos fundamentales:

- El procesamiento de carpetas, donde se recorren las estructuras generadas por ArchivosOBU y Transfer Manager.
- La limpieza de datos antiguos, que garantiza la sostenibilidad del sistema en el tiempo y la gestión eficiente del almacenamiento.

Durante el procesamiento, se aplican diferentes estrategias de interpretación de carpetas (por flota, subflota, individual, del TransferManager), seleccionadas dinámicamente mediante inyección de dependencias, y basadas en una jerarquía común de clases. Esta estructura modular facilita añadir nuevas formas de organización de archivos sin necesidad de modificar la lógica central.

Finalmente, los resultados obtenidos se persisten en una base de datos SQL, utilizando una capa de acceso a datos intermedia (Dbm/DAO) que encapsula los detalles técnicos de la persistencia y facilita el uso de pruebas o simulaciones. Estos resultados son posteriormente accesibles desde el SIU a través de un nuevo endpoint expuesto en su backend.

8.5.2 Arquitectura del Microservicio

En GMV se sigue un estándar a la hora de crear e implementar tanto servicios como nuevos microservicios. Esta plantilla es la que se ha seguido para el desarrollo de InfoVersionService. En la figura 8.7 se puede ver el diagrama de paquetes de InfoVersionService.

Dentro del paquete general InfoVersionService, se encuentran 5 paquetes que corresponden a proyectos .NET. Por simplicidad no se han incluido los paquetes relacionados con los Test pero cada uno de estos paquetes lleva a su vez asociado un paquete Test encargado de los test unitarios correspondientes.

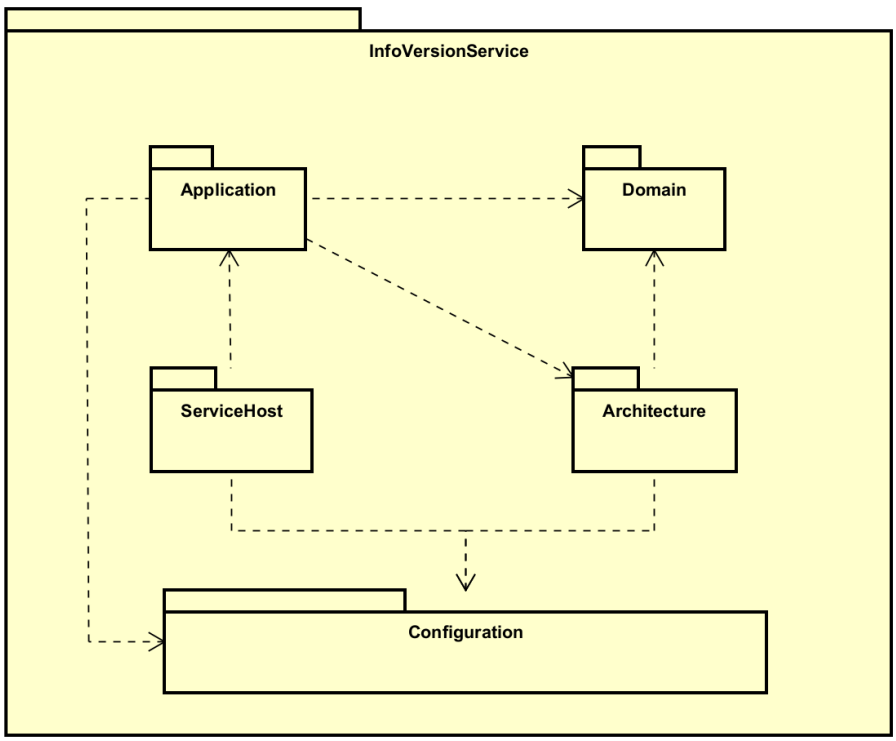


Figura 8.7: Diagrama de Paquetes

Se centrará la atención en los paquetes mostrados en el diagrama de paquetes:

- El paquete **Configuration** es el que contiene toda la información relacionada con la configuración de la aplicación, desde la cantidad de tiempo que ha de pasar entre cada tipo de tarea ejecutada por el servicio, hasta la base de datos a la que se accede y las rutas en las que ArchivosOBU o el TransferManager generan sus archivos. Este paquete internamente por el momento no consta con más subpaquetes, pero es algo que a futuro podría ocurrir en caso de que el servicio escalase y se quisiesen extraer dependencias de configuración.
- El paquete **Domain** contiene las clases que representan las estructuras de datos utilizadas por el servicio para encapsular información como se puede ver en la figura 8.6. Este paquete está diseñado para ser independiente de la lógica de negocio y de la persistencia, y sirve como puente de comunicación entre capas. Todas las clases de este paquete son esencialmente objetos de transferencia de datos (DTOs) o entidades inmutables que encapsulan información del dominio. Este paquete tampoco consta con subpaquetes pues no tiene una alta complejidad. En caso de que la complejidad aumentase se podrían crear nuevos paquetes para evitar alto acoplamiento.
- El paquete **ServiceHost** contiene la clase principal *InfoVersionService*, que representa el punto de entrada del servicio Windows. Este paquete se encarga de inicializar la aplicación, resolver las dependencias mediante inyección, arrancar el procesamiento y gestionar el ciclo de vida del servicio (inicio/parada). No contiene lógica funcional del sistema, ya que su responsabilidad es únicamente servir como contenedor de ejecución.
- El paquete **Architecture** encapsula la lógica de acceso a datos. Cada clase en este paquete representa una entidad persistida y contiene métodos de acceso, actualización y consulta a través de DAOs internos. Este paquete se comunica directamente con la base de datos SQL utilizada por el sistema, y constituye la capa de infraestructura de acceso a persistencia.

  En la figura 8.8 se puede ver la estructura interna.

  Este paquete está organizado internamente en subpaquetes por dominio funcional, tales como Bus, File, PackageType y BusPackage, cada uno con sus propias entidades persistentes (DbmXxx) y mecanismos de acceso (DaoXxx). Esta organización favorece la escalabilidad del sistema, ya que permite extender nuevas áreas de persistencia sin interferir con otras.

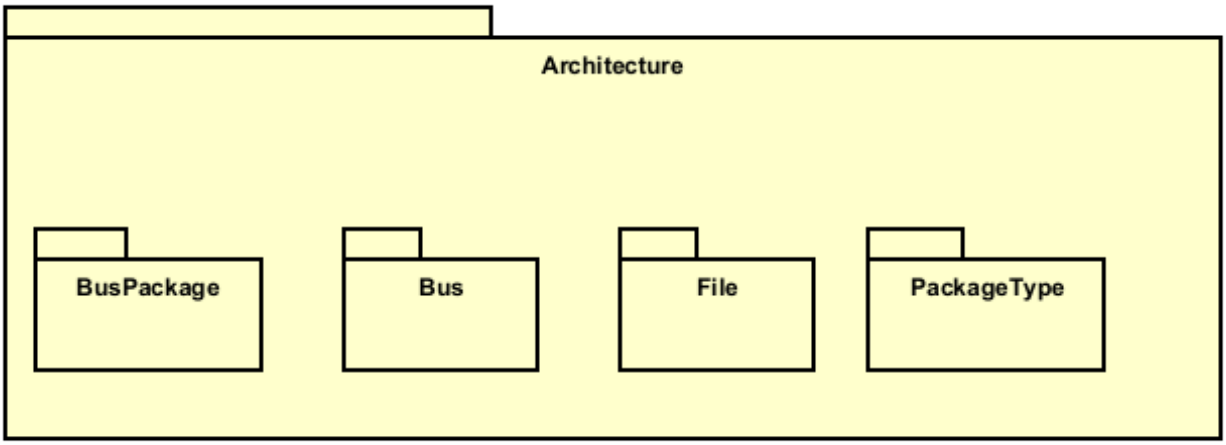


Figura 8.8: SubPaquetes Architecture

- El paquete **Application** constituye el núcleo funcional de la lógica de alto nivel del sistema. En él se centraliza la coordinación de las tareas principales que ejecuta el servicio, estructuradas a su vez en tres

subpaquetes: *Processors*, *Services* y *Utilities*. Esta división permite separar claramente la orquestación de flujos, la ejecución de algoritmos específicos y las operaciones auxiliares de apoyo. En la figura 8.9 se puede ver la estructura interna.

El análisis en profundidad de los paquetes es el siguiente:

- El subpaquete **Processors** contiene la lógica específica para la lectura y procesado de las carpetas generadas por sistemas como *ArchivosOBU* y *TransferManager*. Aquí se definen los distintos procesadores concretos (FleetProcessor, *SubFleetProcessor*, *ParticularProcessor*, *TMProcessor*) que heredan de una clase base común y comparten una misma interfaz, *IFolderProcessor*. Este subpaquete implementa el patrón *Strategy* 8.3.4, permitiendo que el servicio seleccione dinámicamente la estrategia adecuada para procesar cada carpeta según su estructura lógica. Asimismo, se aplica el patrón *Template Method* 8.3.6 mediante la clase *FolderProcessorBase*, que define un flujo estándar para el recorrido de carpetas y delega los detalles específicos en las subclases.
- El subpaquete **Services** agrupa los servicios encargados de aplicar la lógica de negocio de más alto nivel. Aquí se encuentran clases como *VersionProcessingService*, que coordina toda la ejecución periódica del sistema (programación de tareas, ejecución, supervisión), o *BusProcessingTracker*, que gestiona el estado de los vehículos procesados para evitar repeticiones innecesarias. También se incluye el uso del patrón *Façade* 8.3.2, ya que estos servicios encapsulan las interacciones entre los distintos componentes internos, ofreciendo un punto de entrada único para el procesamiento principal.
- El subpaquete **Utilities** contiene funciones auxiliares específicas del módulo de aplicación que no pertenecen a lógica de dominio puro, pero tampoco son genéricas del sistema completo (como las que irían en un *Util* general). Se incluyen aquí operaciones de soporte como validaciones especializadas, lógica de temporización avanzada para tareas programadas (*ScheduledTask* 8.3.5), y ayudas reutilizables propias del servicio, pero separadas de la infraestructura técnica o la lógica de negocio central.

Esta estructura modular del paquete **Application** favorece el mantenimiento, mejora la organización de responsabilidades y permite extender fácilmente la funcionalidad del sistema sin comprometer la estabilidad del núcleo existente.

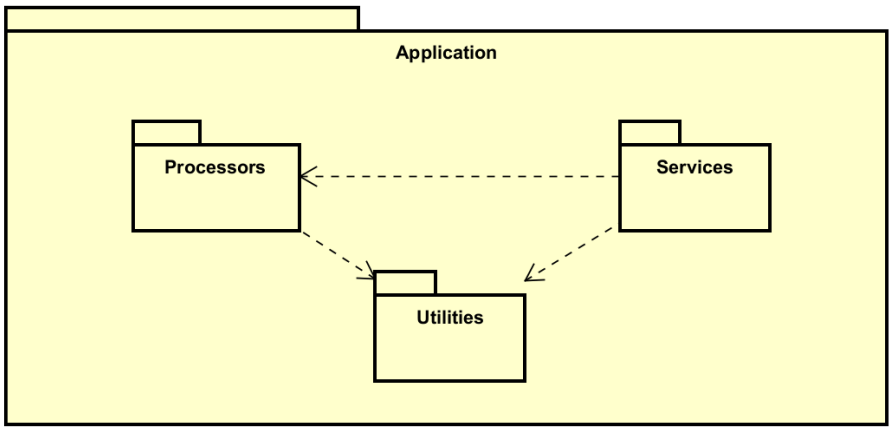


Figura 8.9: SubPaquetes Application

8.5.3 Dependencias entre submódulos

En el diseño interno de InfoVersionService, los submódulos se estructuran de forma que respetan una dirección de dependencia descendente: los módulos de alto nivel (como ServiceHost y Application) consumen los

de nivel inferior (Model, Configuration, Dbm, Util), pero nunca al revés. Esta orientación permite mantener un diseño modular, desacoplado y fácilmente testeable.

En la figura 8.10 se puede ver un diagrama con las dependencias entre las distintas capas.

A continuación se describen las principales dependencias entre submódulos, agrupadas por nivel:

- Nivel Superior **ServiceHost**: Este módulo es el punto de entrada de la aplicación (clase *InfoVersionService*) y depende directamente de:
  - **Application.Services:** para iniciar/parar el procesamiento (*VersionProcessingService*).
  - **Configuration:** para leer los parámetros necesarios de arranque.

  No contiene lógica propia ni es dependido por otros módulos.
- Nivel Intermedio **Application**: El módulo de aplicación se divide en tres submódulos funcionales con relaciones claras.
  - **Application.Services**: el cual depende tanto de *Application.Processors* para invocar los distintos procesadores de carpetas y archivos, como todos los *Dbm* para leer/escribir resultados y *Domain* como estructura intermedia de datos.
  - **Application.Processors:** el cual únicamente depende de Domain para la creación de los datos intermedios.
  - **Application.Utilities:** No se muestra en el diagrama pues no tiene dependencias externas pero es utilizado por el resto de submódulos de *Application* para tareas comunes como temporización entre otras.
- Nivel de infraestructura **Architecture:** Esta capa depende solamente de *Domain* para transformar DTOS en entidades persistentes y viceversa, y de *Configuration* para todas las configuraciones de BD necesarias.
- Módulos Transversales (**Domain y Configuration**): Estos módulos se utilizan en los demás, pero no dependen de ninguno, respetando su carácter transversal.
  - Model define estructuras puras de datos
  - Configuration contiene las clases relacionadas con la configuración del servicio como por ejemplo un singleton con variables como las rutas de los archivos, tiempos de procesado, etc.

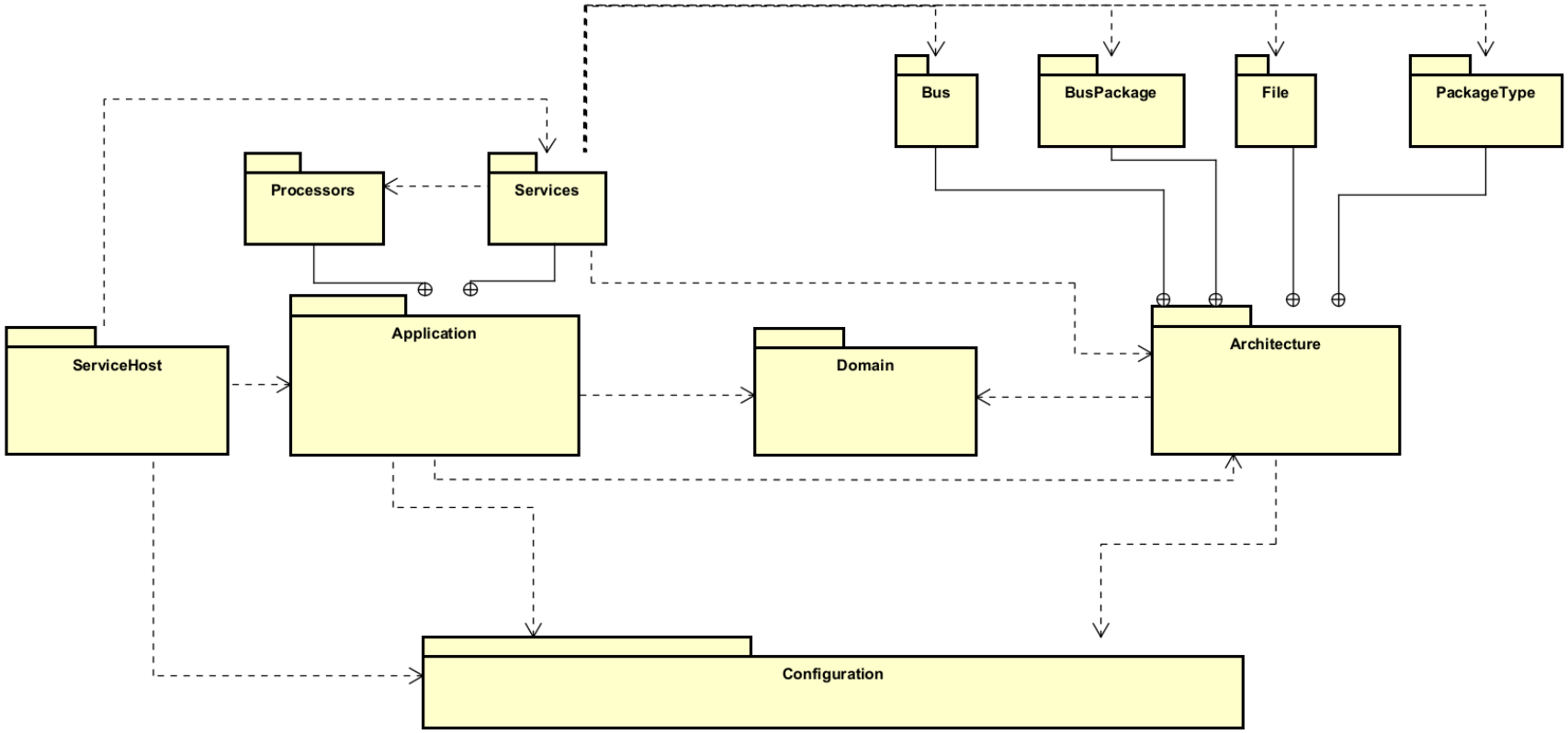


Figura 8.10: Dependencias entre capas

8.5.4 Diagrama de clases entre capas

La comprensión estructural de un sistema software no se limita únicamente al análisis de sus paquetes o capas funcionales, sino que requiere también un entendimiento claro de cómo se relacionan e interactúan las clases que lo componen. En este apartado se presenta un diagrama de clases intermodular, cuyo objetivo es representar las principales entidades del sistema InfoVersionService, su distribución en las diferentes capas arquitectónicas, y las relaciones que se establecen entre ellas.

El enfoque adoptado para este diseño responde a los principios de la arquitectura en capas, donde cada nivel funcional (orquestación, procesamiento, negocio, persistencia) tiene responsabilidades claramente delimitadas. A su vez, se han aplicado principios de diseño orientado a objetos como la inversión de dependencias, la separación de responsabilidades (SRP), y el uso de interfaces para garantizar un bajo acoplamiento y una alta cohesión.

Este diagrama es especialmente útil para ilustrar cómo fluye la información en el sistema desde la activación del servicio hasta la escritura en la base de datos. También permite visualizar cómo se integran distintos patrones de diseño identificados previamente, como Strategy, Template Method y Façade y cómo contribuyen a estructurar un sistema escalable y mantenible.

A lo largo de este apartado se explicará el rol de cada clase clave, agrupadas por capa funcional, así como las asociaciones y dependencias que existen entre ellas. Esta representación refuerza el diseño modular del sistema y pone en evidencia las buenas prácticas aplicadas durante su desarrollo.

En la figura 8.5.4 se puede ver el diagrama de conexión de capas por medio de las clases principales.

A continuación, se explicará el rol de las clases clave:

- **ServiceHost**
  - **InfoVersionService:** Es la clase principal del servicio de Windows. Se encarga de gestionar el ciclo de vida en el sistema (OnStart, OnStop), resolver dependencias iniciales y delegar la ejecución al componente *VersionProcessingService*. No contiene lógica de negocio.
- **Application.Services**
  - **VersionProcessingService:** Es el componente central de orquestación. Programa y lanza las tareas periódicas de procesado y limpieza, inicializa las estrategias de procesamiento (IFolderProcessor) y mantiene el control de ejecución global. Aplica el patrón Façade 8.3.2.
  - **PackageService:** Se encarga de la lógica de análisis de archivos y generación de datos estructurados. Es invocado por los procesadores para interpretar los archivos encontrados en disco y construir objetos Package y File. Valida formatos, extrae metadatos relevantes, determina tipos de paquete y gestiona las operaciones necesarias para registrar la información de versiones en el sistema.

    Es un punto clave de conexión entre el recorrido físico del sistema de ficheros y el modelo de datos interno. Aplica validaciones específicas, delega la persistencia en la capa de *Architecture* y colabora estrechamente con *BusProcessingTracker*.
- **Application.Processors**
  - **IProcessor:** Interfaz que define el contrato común para todos los procesadores de carpetas. Permite aplicar el patrón Strategy 8.3.4 y sustituir dinámicamente la lógica según el tipo de carpeta.
  - **FolderProcessorBase:** Clase abstracta que implementa el patrón Template 8.3.6. Define el flujo general de procesamiento (recorrido de carpetas, obtención de vehículos, evaluación de versiones) y deja los detalles a implementar por las subclases concretas.
  - **FleetProcesssor, SubFleetProcessory ParticularProcessor:** Implementaciones específicas del procesador que manejan estructuras diferentes de carpetas o fuentes (ArchivosOBU). Cada clase implementa su propia lógica para interpretar rutas y extraer identificadores de OBU.

  - **TMProcessor:** Es una implementación específica del *IProcessor* relacionada únicamente con los archivos generados por el *TransferManager*. Esto se debe a que los archivos generados no siguen la misma estructura que *ArchivosOBU*, en este caso se encuentran todos en la misma carpeta y se identifican por el nombre del archivo mientras que *ArchivosOBU* los distribuye en diferentes carpetas ya sea por flota, bus, etc.

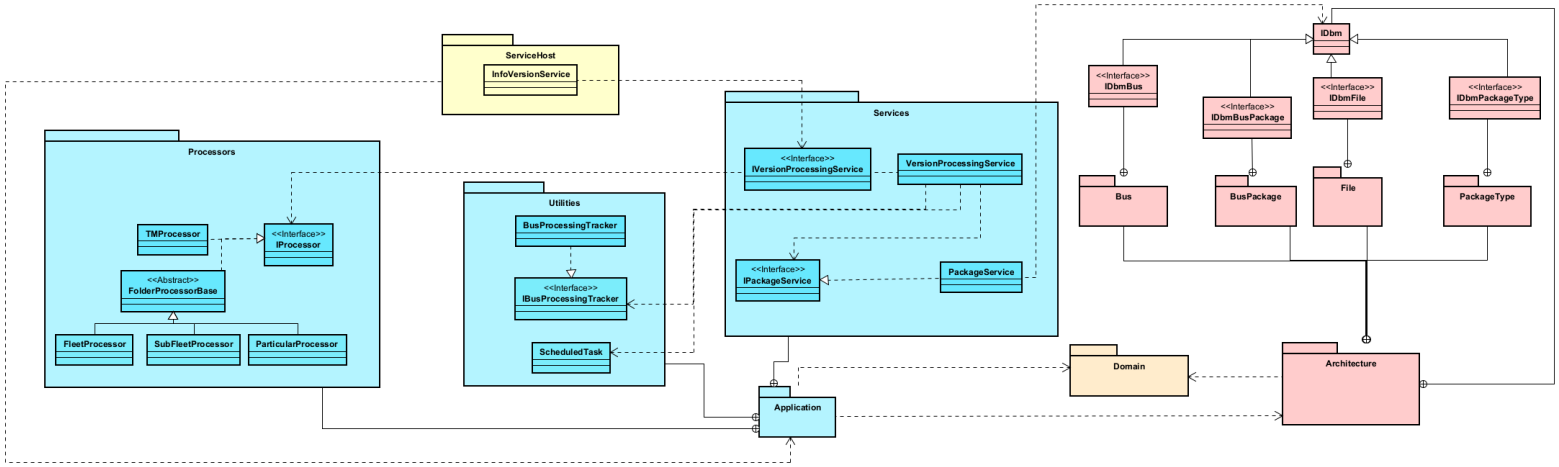
- **Application.Utilities**

  - **BusProcessingTracker:** Clase que actúa como mecanismo de control en memoria, evitando que un mismo vehículo se procese múltiples veces dentro de una misma ejecución. Esencial para mantener eficiencia y evitar duplicidad de resultados. Esto se hace principalmente para que si ArchivosOBU genera versiones para toda la flota pero dentro de esa misma flota hay ciertos buses particulares con versiones distintas, no tengan tanto la versión de la flota como la particular, si no la que le corresponda en cada caso.

- **Domain:** Este apartado se puede consultar en profundidad en el capítulo 8.4.

- **Architecture:** Esta capa es la encargada de mapear y acceder a las distintas tablas relacionadas con cada modelo definido en el capítulo 8.4. Consta de una interfaz general llamada IDbm que tiene los métodos generales de todos los data managers de la que heredan las interfaces e implementaciones concretas para cada tipo de dato..

La correcta separación de responsabilidades entre estas clases, junto con el uso sistemático de interfaces, clases abstractas y servicios, garantiza que el sistema sea fácilmente extensible, testeable y mantenible. Este diseño modular permite añadir nuevos tipos de procesamiento, ajustar los criterios de comparación o cambiar la fuente de datos sin alterar la arquitectura general del sistema.


ServiceHost
InfoVersionService
Processors
TMProcessor
<<Interface>>
IProcessor
<<Abstract>>
FolderProcessorBase
FleetProcessor
SubFleetProcessor
ParticularProcessor
Utilities
BusProcessingTracker
<<Interface>>
IBusProcessingTracker
ScheduledTask
Services
<<Interface>>
IVersionProcessingService
VersionProcessingService
<<Interface>>
IPackageService
PackageService
Application
IDbm
<<Interface>>
IDbmBus
<<Interface>>
IDbmBusPackage
<<Interface>>
IDbmFile
<<Interface>>
IDbmPackageType
Bus
BusPackage
File
PackageType
Domain
Architecture

8.5.5 Diagramas de Secuencia

En este apartado se describen los flujos dinámicos de ejecución que se producen durante el funcionamiento normal del sistema. A diferencia de los diagramas de clases (que representan la estructura estática), los diagramas de secuencia muestran la interacción temporal entre objetos o componentes, permitiendo visualizar cómo fluye la información y qué clases participan activamente en cada etapa del proceso. Dado que InfoVersionService se compone de tareas cíclicas y procesos automatizados, se han identificado dos flujos representativos:

- **Proceso de comparación de versiones (Tarea principal):** Este diagrama representa lo que ocurre cuando el servicio lanza una tarea periódica para analizar las carpetas generadas por ArchivosOBU o Transfer Manager, extraer las versiones, compararlas, y registrar tanto el resultado como los paquetes con sus archivos y versiones. Los diagramas de secuencia se pueden ver en las figuras 8.11, 8.12 y 8.13.

  En estos diagramas de secuencia se representa el flujo de ejecución correspondiente al método *Process-Folders()* de la clase *VersionProcessingService*, que constituye el núcleo de procesamiento cíclico de versiones en el sistema.

  El sistema dispone de una colección de estrategias de procesado que implementan la interfaz *IProcessor*, las cuales son inyectadas dinámicamente mediante el mecanismo de inyección de dependencias (véase la sección 8.3.3). Cada una de estas estrategias se encarga de recorrer una estructura de carpetas específica, generada previamente por los servicios *ArchivosObu* o *TransferManager* (Véase el capítulo 7).

  Las carpetas a procesar son determinadas a través del método *GetFoldersToProcess()*. Durante este recorrido, cada procesador identifica y almacena referencias a los directorios asociados a paquetes concretos (PackageType según el análisis realizado), en este caso, correspondientes a los paquetes generales de contenidos.

  Posteriormente, cada estrategia invoca el método *GetBusIds()*, cuya implementación varía en función del tipo de procesador (por ejemplo, FleetProcessor, SubFleetProcessor, entre otros), con el objetivo de determinar los identificadores de los vehículos implicados.

  Una vez se ha procesado la carpeta principal del procesador actual, se realiza una llamada al método *ProcessPackage()* de la clase *PackageService*, el cual se encarga de procesar los distintos directorios, es decir, tipos de paquete, encontrados en los pasos anteriores. Estos directorios constan con un archivo de texto de control(Véase el capítulo 7). Este método se encarga de encontrar estos archivos de texto dentro de cada paquete para así procesar su contenido, crear los correspondientes *BusPackages* y *Files* e insertarlos en los buses conseguidos por medio de *GetBusIds()*

  Con el fin de simplificar el diagrama de secuencia y mejorar su legibilidad, se ha optado por representar únicamente una de las implementaciones de *IProcessor*, concretamente *FleetProcessor*. Esta representación se encuentra encapsulada dentro de un bloque loop, que indica que dicha lógica se ejecuta de forma iterativa para cada uno de los procesadores registrados en el sistema.

  Este enfoque representa de forma precisa el uso del patrón Strategy 8.3.4, dejando implícito que el comportamiento sería equivalente para otras implementaciones como *SubFleetProcessor*,*ParticularProcessor* o *TMProcessor*. Además, se han simplificado los diagramas para evitar algunas implementaciones que no tienen gran relevancia.

- **Proceso de limpieza de registros antiguos (Mantenimiento periódico)**: Este diagrama de secuencia representa el flujo correspondiente al proceso de limpieza periódica de registros antiguos, llevado a cabo por el método *CleanOldRecords()* de la clase *VersionProcessingService*. Esta funcionalidad forma parte de las tareas programadas que ejecuta el servicio de manera autónoma, y su objetivo es eliminar de la base de datos aquellos paquetes de versión cuya antigüedad supere al umbral designado en el paquete de configuración.

  El proceso se ejecuta en segundo plano mediante una instancia de *ScheduledTask* 8.3.5 que, tras el intervalo de tiempo invoca al método *CleanOldRecords()*.

El diagrama de secuencia se puede ver en la figura 8.14

Cabe señalar que, con el fin de mejorar la legibilidad de los diagramas de secuencia presentados, se ha optado por omitir explícitamente el tratamiento de posibles excepciones o errores. No obstante, en la implementación real, dichos mecanismos de control están debidamente contemplados para garantizar la robustez del sistema.

8.5.6 Consideraciones de extensibilidad, mantenibilidad y escalabilidad

Uno de los objetivos fundamentales en el diseño de InfoVersionService ha sido garantizar que el sistema sea fácilmente extensible, mantenible y escalable a medio y largo plazo. La implementación se ha construido desde el inicio siguiendo principios de diseño sólido y arquitectura limpia, con el fin de facilitar la evolución del sistema sin introducir efectos colaterales ni comprometer su estabilidad.

- **Extensibilidad:** El sistema ha sido diseñado para que nuevos comportamientos o funcionalidades puedan incorporarse sin modificar el código existente, sino añadiendo nuevos módulos de forma controlada. Destacan las siguientes decisiones que favorecen la extensibilidad:
  - El uso del patrón Strategy 8.3.4 permite incorporar nuevos procesadores (*IProcessor*) para tratar estructuras de carpetas distintas (por ejemplo, nuevos formatos generados por otros sistemas), sin necesidad de modificar los procesadores actuales ni la lógica central del servicio.
  - La separación entre lógica de recorrido de carpetas y lógica de interpretación de archivos permite que los cambios en la estructura de carpetas o en los tipos de contenido se puedan abordar de forma independiente.
  - La inyección de dependencias hace que sea posible sustituir servicios concretos por otros (por ejemplo, una nueva implementación de almacenamiento) sin modificar la lógica de negocio.
  - El uso de *ScheduledTask* 8.3.5 permite la creación de nuevas tareas en segundo plano que se coordinen automáticamente con el resto de tareas ya creadas, sin modificar la lógica actual del sistema.

  Estas decisiones permiten incorporar nuevos procesadores dedicados al versionado de cualquier tipo de contenido e, incluso, extender su ámbito más allá de lo puramente documental.
- **Mantenibilidad:** El sistema cumple principios clave de diseño orientado a objetos como:
  - **Responsabilidad única (SRP):** cada clase tiene una responsabilidad claramente definida (por ejemplo, los procesadores se encargan solo del procesado de carpetas, mientras que PackageService crea los paquetes de versiones y los inserta en base de datos).
  - **Bajo acoplamiento y alta cohesión:** las clases están organizadas en capas y paquetes funcionales, con dependencias claras y bien aisladas. Ninguna clase accede directamente a capas externas (por ejemplo, lógica de negocio a base de datos), sino que lo hace a través de interfaces.
- **Escalabilidad:** Aunque actualmente el servicio se ejecuta como una única instancia Windows, su arquitectura está preparada para escenarios futuros que puedan requerir más carga o mayor paralelismo:
  - El diseño por tareas permite separar los distintos procesos (procesamiento vs limpieza) y ejecutarlos de forma independiente.
  - El modelo de datos puede escalar horizontalmente con soporte de índices, particiones por campaña o flota, y almacenamiento distribuido si fuera necesario.

Finalmente, el desacoplamiento total entre InfoVersionService y los sistemas que consumen sus resultados (como el SoaBasicContentManager o el SiuFront) permite que cada uno de estos componentes pueda evolucionar de forma independiente, sin generar bloqueos funcionales ni dependencias rígidas.

Este diseño modular no solo facilita la integración inicial con el gestor de contenidos, sino que convierte a InfoVersionService en una solución genérica y reutilizable para cualquier funcionalidad relacionada con el control de versiones de contenido en GMV, más allá del ámbito específico del SIU.

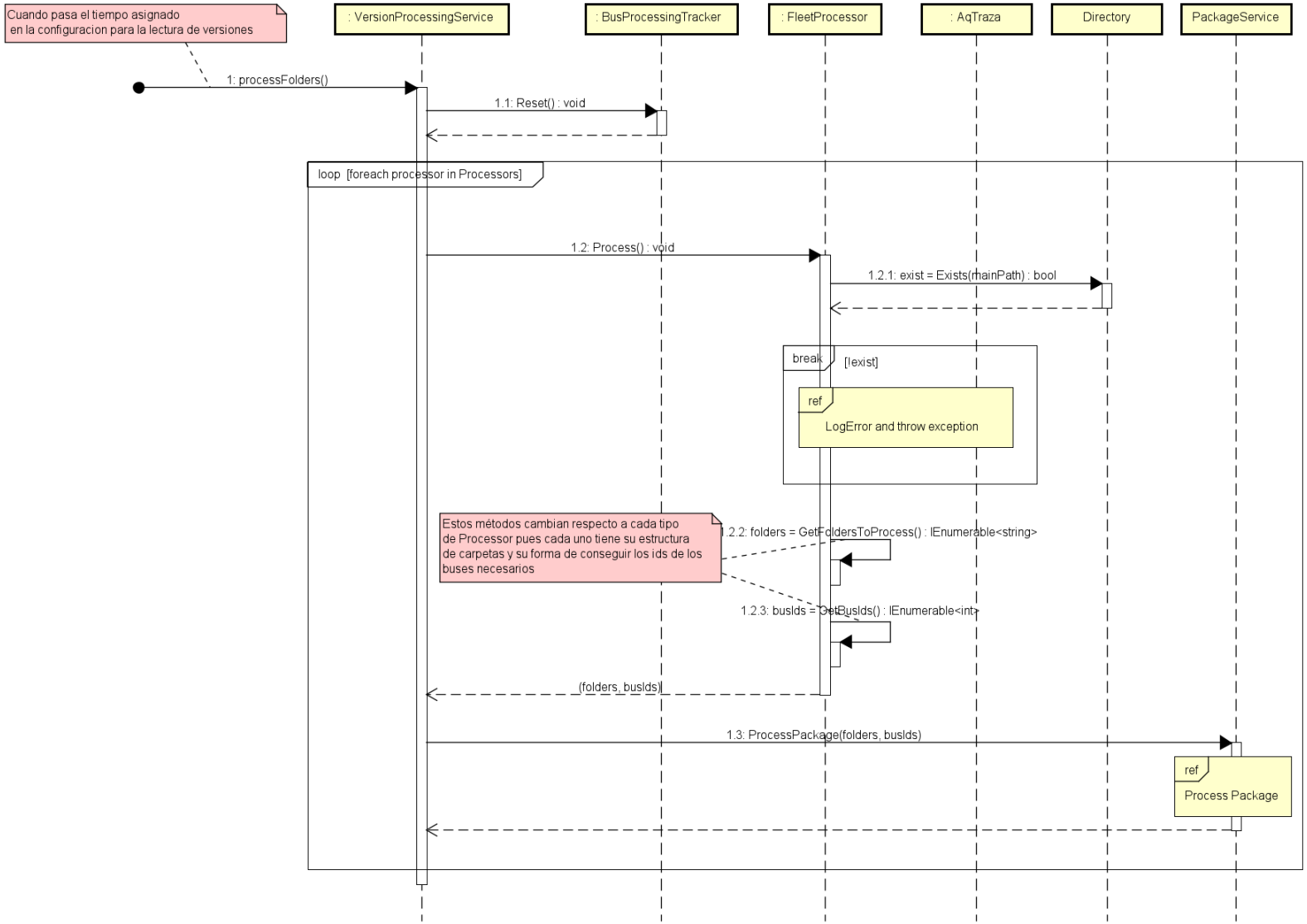


Figura 8.11: Diagrama de Secuencia ProcessFolder

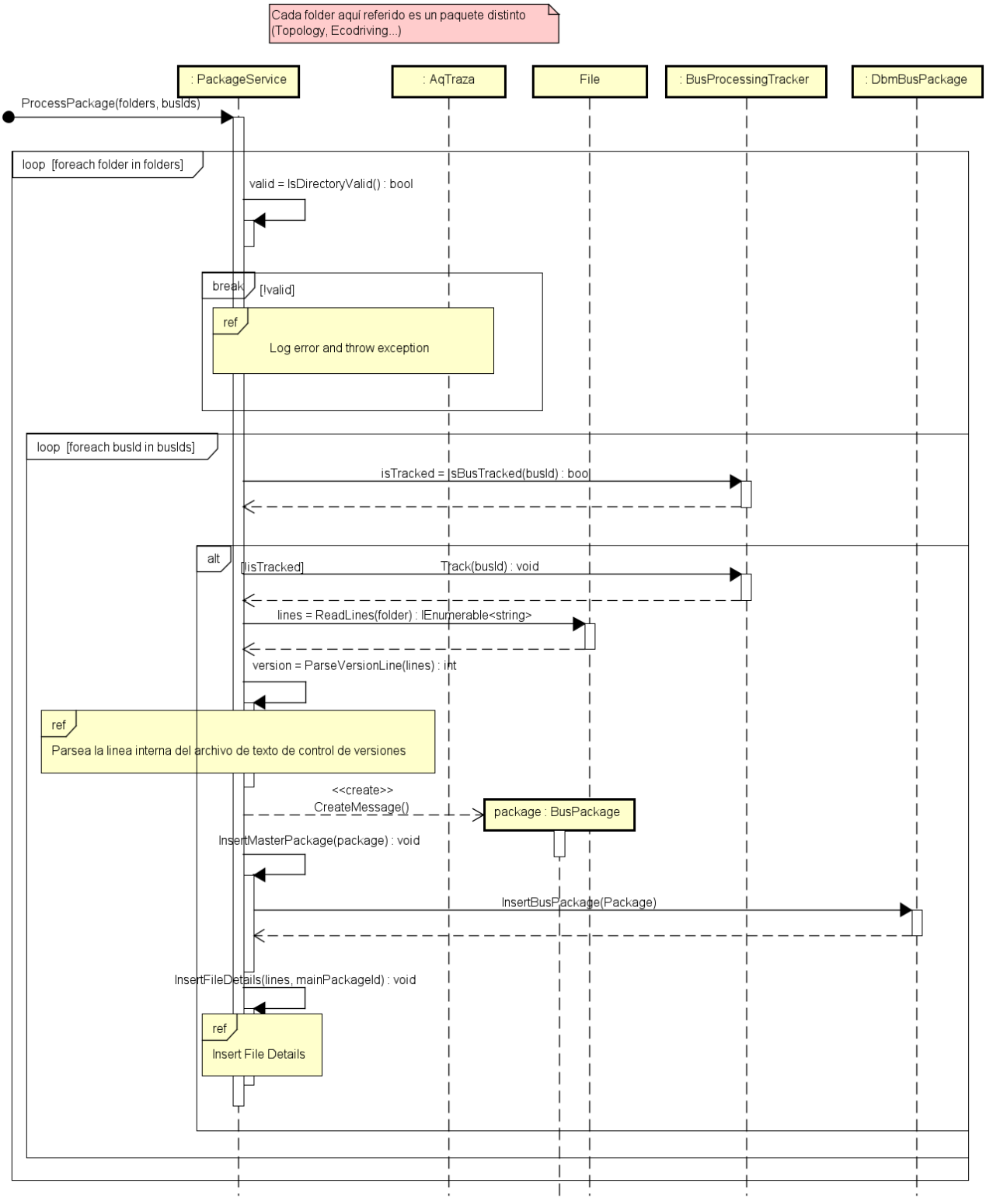


Figura 8.12: Diagrama de Secuencia ProcessPackage

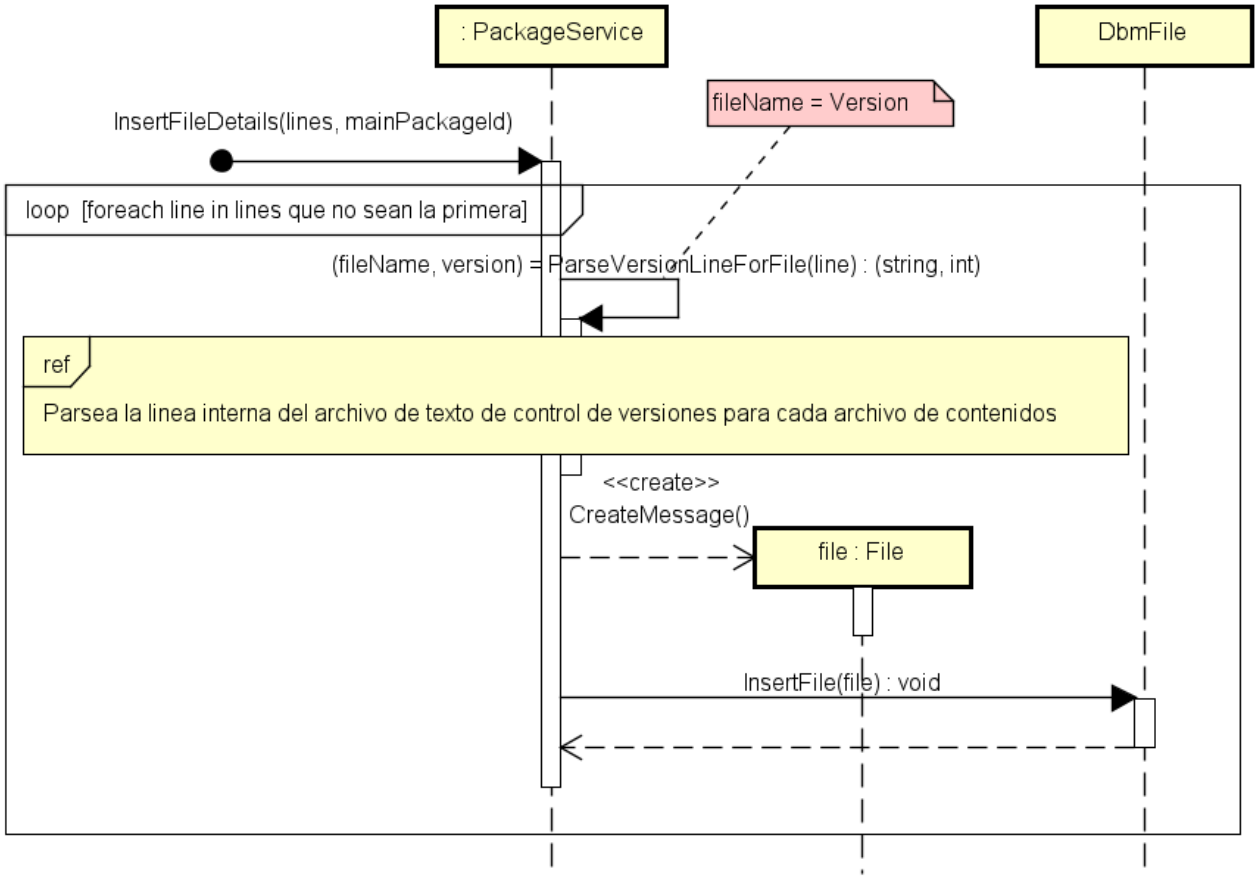


Figura 8.13: Diagrama de Secuencia InsertFileDetails

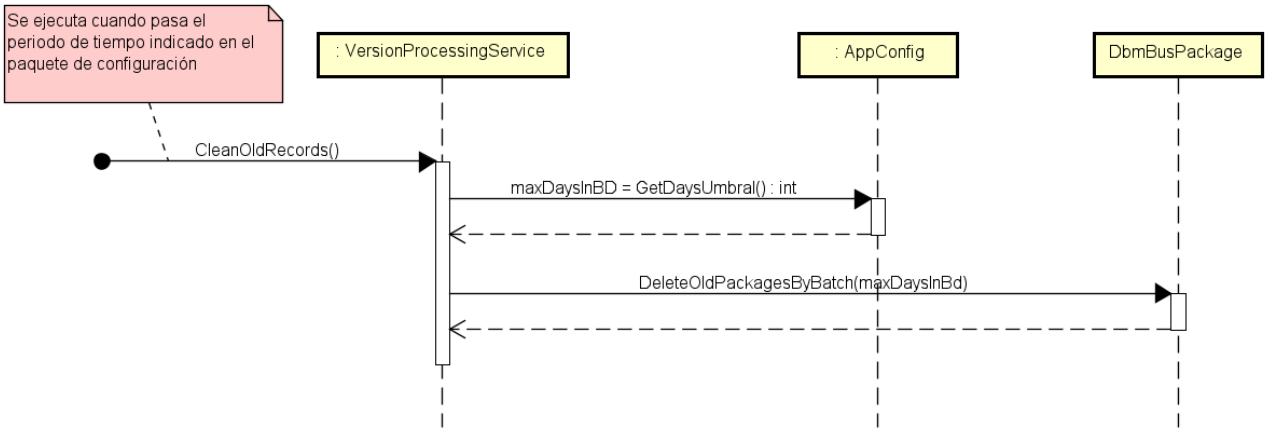


Figura 8.14: Diagrama de Secuencia CleanOldRecords

8.5.7 Resumen de la arquitectura de InfoVersionService

En resumen, el módulo InfoVersionService se estructura en torno a un servicio Windows encargado de ejecutar periódicamente tareas programadas de análisis y limpieza. Este servicio se apoya en una arquitectura basada en inyección de dependencias, separación en capas y procesadores para interpretar archivos, registrar versiones y mantener la trazabilidad de forma desacoplada. Su diseño modular permite tanto la escalabilidad a otros tipos de paquetes como su integración con nuevos sistemas sin alterar el núcleo del servicio.

8.6 Interfaz de Usuario

La solución desarrollada incluye una interfaz gráfica integrada en el sistema SIU, específicamente en su módulo de administración técnica, con el objetivo de proporcionar a los operadores una visión clara y centralizada del estado de actualización de contenidos por vehículo.

Este componente frontend permite consultar, de manera visual y ordenada, la información generada por el servicio InfoVersionService, proporcionando trazabilidad sobre las versiones esperadas y reales instaladas en cada OBU, así como posibles inconsistencias detectadas durante el proceso de sincronización.

Aunque la lógica de control y generación de datos reside completamente en el *backend*, la interfaz de usuario representa una parte esencial del sistema desde el punto de vista de la operación técnica y toma de decisiones. La información expuesta permite validar si una flota está en condiciones de ser desplegada, detectar errores en la distribución de archivos, o confirmar la correcta instalación de contenidos críticos.

Dado que el SIU es un sistema complejo, con estilos y flujos propios ya consolidados, la interfaz desarrollada respeta tanto la línea visual existente como los patrones de interacción definidos previamente, de forma que se integra de forma transparente en la experiencia del usuario técnico.

8.6.1 Estructura de la interfaz

La interfaz desarrollada se ha diseñado como un nuevo componente visual dentro del entorno ya existente del SIU, siguiendo la arquitectura técnica y visual propia del sistema. Su objetivo es proporcionar a los usuarios técnicos una vista detallada y centralizada del estado de actualización de versiones por vehículo, de forma que puedan verificar la sincronización de manera rápida y fiable antes de autorizar despliegues operativos.

La vista principal es la tabla de resultados, este es el componente principal de la interfaz, una tabla interactiva que muestra una entrada por cada vehículo y tipo de paquete configurado en el sistema. Esta tabla representa el estado actual de sincronización para cada combinación, mostrando los siguientes campos principales:

- Vehículo: Mostrando tanto su sideCode como su matrícula (Véase el capítulo 8.4)
- Tipo de Paquete: corresponderán a distintas columnas, tantas como tipos de paquete haya introducidos en el sistema, por ejemplo, contenido multimedia, configuración, topología, etc. Algunos de estos paquetes pueden estar formados por subpaquetes, por ejemplo, topología engloba a líneas, rutas, trayectos, correspondencias, etc. Estas columnas mostrarán una X o un V dependiendo del estado de actualización del paquete en el bus.
- Versión de configuración del usuario: Esta versión hace referencia al número de versión que **conoce** el usuario, no el interno generado por *ArchivosObu* y *TransferManager*.
- Versión esperada e instalada: se mostrarán como un tooltip en las columnas de paquetes correspondientes, evitando así la saturación de la interfaz.
- Fecha de última transmisión.

Además de este listado, la interfaz incluye elementos de interacción pensados para mejorar la usabilidad, como filtros, ordenación por columnas (alfabética, por fecha, por estado), e incluso exportación a excel.

En la misma vista principal, además de la tabla de resultados, hay un pequeño panel con KPIs que darán información acerca del % de buses sincronizados, la cantidad exacta que no lo están y la cantidad exacta disponibles para salir a despliegue.

En la figura 9.1 puede verse un ejemplo de esta interfaz.

Capítulo 9

Implementación

Este capítulo describe el proceso de implementación de la solución diseñada, detallando las herramientas utilizadas, el entorno de desarrollo, así como las decisiones técnicas adoptadas en los distintos componentes del sistema: el servicio InfoVersionService, su integración con las fuentes de datos externas, la interfaz gráfica implementada en el SIU y los mecanismos de control y validación.

La implementación se ha realizado siguiendo los principios definidos en los capítulos anteriores, aplicando una arquitectura en capas, utilizando patrones de diseño reutilizables y asegurando la mantenibilidad del sistema a medio y largo plazo.

Por temas de privacidad, en este capítulo no se expondrá código concreto del software desarrollado en este proyecto.

9.1 Pautas de Estilo

Debido a que el nuevo servicio de windows es muy probable que se acabe utilizando por múltiples proyectos y escalando a futuro, se ha decidido seguir ciertas pautas de estilo para la mejor legibilidad y cohesión del código.

Las principales pautas a seguir son las siguientes:

- Los atributos privados deben estar precedidos por _.
- Las interfaces han de empezar siempre por I.
- No debe haber espacios colgantes en el código.
- A la hora de declarar las variables se ha de poner el tipo concreto y no utilizar la inferencia de tipos de C# u React (var).
- El nombre de los atributos públicos ha de empezar por mayúscula.
- Todos los nombres han de estar en inglés.
- El nombre de todos los métodos han de empezar por mayúscula.
- Se deberán crear regiones que permitan organizar el código.

9.2 InfoVersionService

La implementación de *InfoVersionService* comenzó tras definir la arquitectura lógica del sistema y seleccionar el patrón de diseño por capas, que posteriormente se consolidó en el módulo InfoVersionService. Esta parte del sistema se desarrolló como un servicio Windows autónomo utilizando C# sobre .NET 4.7.2, ya que esta tecnología permitía integrar fácilmente tareas programadas, acceso a ficheros del sistema, gestión de procesos en segundo plano y trazabilidad de eventos, todo ello dentro del entorno tecnológico utilizado en GMV.

El desarrollo se abordó de manera incremental, siguiendo las fases previamente establecidas durante el diseño. En primer lugar, se construyó la base del servicio, incluyendo la clase *InfoVersionService*, que define los métodos de arranque y parada del sistema, así como los puntos de entrada para la ejecución de tareas. A partir de ahí, se desarrolló el núcleo funcional del servicio en la clase *VersionProcessingService*, responsable de orquestar el comportamiento global. Esta clase implementa el control del ciclo de vida, la inicialización de tareas programadas y la coordinación de los distintos componentes del sistema, incluidos los procesadores de carpetas, los servicios de análisis de versiones y el acceso a base de datos.

Una vez establecida la estructura básica, se diseñó la lógica de procesamiento de carpetas. Para ello se definió la interfaz *IProcessor* y una clase abstracta común, *FolderProcessorBase*, que establece el flujo genérico de ejecución para cualquier procesador. Esta estructura permitió aplicar el patrón Template Method 8.3.6 y Strategy 8.3.4 y garantizar que todos los procesadores compartieran la misma secuencia de pasos, al tiempo que permitía a cada uno implementar únicamente la lógica específica de su contexto. Posteriormente se añadieron las implementaciones concretas, como *FleetProcessor*, *SubFleetProcessor* o *TMProcessor*, que se encargan de interpretar distintas estructuras de carpetas generadas por herramientas externas como *ArchivosOBU* o *Transfer Manager*.

A continuación, se abordó la implementación del análisis de versiones, que se concentró en la clase *PackageService*. Este servicio fue diseñado para recibir los archivos encontrados por los procesadores, analizarlos y generar estructuras de versión esperada y detectada por cada vehículo y tipo de contenido. Una vez obtenida esta información, se construyen los resultados de comparación, que posteriormente se persisten en base de datos. Esta lógica se desarrolló de forma desacoplada del recorrido físico de carpetas, siguiendo los principios de separación de responsabilidades, con el objetivo de facilitar tanto su mantenimiento como su reutilización.

La persistencia de datos se implementó mediante clases específicas de acceso a base de datos agrupadas en el paquete *Architecture*, como *DbmBus*, *DbmPaqueteAutobus* o *DbmFile*. Siendo éstas implementaciones de sus respectivas interfaces que heredan de un *Dbm* general con distintos métodos comúnes. Estas clases encapsulan las operaciones de inserción, actualización y consulta sobre las tablas necesarias, y su uso se limita exclusivamente a la capa de servicios, siguiendo una lógica de acceso indirecto a través de interfaces. Además, se añadió una tarea periódica de limpieza de registros antiguos, que elimina información cuya antigüedad supera el umbral definido en configuración, manteniendo así la base de datos optimizada.

Tras el desarrollo inicial y las primeras pruebas con volúmenes moderados, se detectó que, en escenarios reales de despliegue, el sistema podría llegar a almacenar millones de registros de versiones de contenido en base de datos, especialmente en flotas grandes o con múltiples tipos de paquetes por vehículo. Esta previsión llevó a introducir optimizaciones significativas en la capa de persistencia y en la gestión del rendimiento general del sistema. En particular, se incorporó el uso de operaciones de inserción masiva (bulk insert) para reducir la latencia en el guardado de grandes volúmenes de datos, y se implementó paralelismo controlado en el procesamiento de carpetas para mejorar el aprovechamiento de los recursos del sistema. Además, se añadieron índices específicos en base de datos sobre columnas clave para acelerar las consultas y operaciones de mantenimiento, asegurando la escalabilidad y estabilidad del servicio ante cargas elevadas.

Durante toda la implementación se prestó especial atención a la robustez y tolerancia a fallos. Cada parte del sistema incluye trazas generadas por AqTraza, una librería capaz de generar logs de manera sencilla, lo que permite registrar información de diagnóstico, advertencias y errores. Asimismo, se envolvieron todos los puntos críticos con bloques de control de excepciones, de modo que una carpeta malformada o un archivo erróneo no pueda detener el funcionamiento global del servicio. Toda la configuración, incluidos los intervalos de ejecución, rutas de carpetas, tipos de contenido y parámetros de limpieza, se centralizó en la clase *AppConfig*,

lo que facilita la adaptación del sistema a diferentes entornos sin necesidad de recompilar.

En definitiva, la implementación del backend se ajustó en todo momento a los principios de diseño definidos durante la fase de análisis y arquitectura, y permitió materializar la solución planteada de forma modular, estable y fácilmente extensible. El resultado final fue un servicio autónomo, ejecutable en segundo plano, que genera de forma continua los datos necesarios para controlar la trazabilidad de contenidos en los vehículos, sirviendo como núcleo funcional del sistema.

9.3 Acceso a fuentes de datos

El núcleo funcional de *InfoVersionService* se basa en la capacidad de comparar la información que se espera que esté instalada en los vehículos (versiones esperadas) con la información realmente instalada (versiones detectadas o en bus). Para ello, el sistema debe integrarse con dos fuentes externas clave del ecosistema GMV: *ArchivosOBU* y *Transfer Manager*. Ambas herramientas generan archivos estructurados en rutas compartidas, y es precisamente sobre estas estructuras donde opera el servicio.

A diferencia de otros sistemas que se comunican mediante APIs o colas de mensajes, InfoVersionService está diseñado para trabajar mediante acceso directo al sistema de archivos, consumiendo de forma no intrusiva la información generada por las herramientas existentes. Esta decisión garantiza una integración sencilla, robusta y sin interferencias, respetando completamente el flujo actual de generación y distribución de contenidos en los vehículos.

En los siguientes subapartados se detalla cómo se ha modelado este acceso, y cómo se interpreta la información procedente tanto de ArchivosOBU (como fuente de versiones esperadas), como de Transfer Manager (como fuente de versiones realmente instaladas).

9.3.1 Modelo de acceso a datos

El servicio *InfoVersionService* no interactúa con *ArchivosOBU* ni con *Transfer Manager* a través de APIs o comunicación directa, sino que accede a ellos mediante el sistema de archivos compartido, donde ambas herramientas depositan sus resultados. Este enfoque simplifica la integración y garantiza una mínima intrusión en los sistemas existentes. Las rutas de acceso se definen en la configuración (*AppConfig*) y son utilizadas por los procesadores para recorrer carpetas, identificar vehículos y leer los archivos correspondientes.

Cada procesador conoce la estructura de las carpetas que le corresponde analizar, y aplica lógica específica para interpretar los nombres de archivos, carpetas y rutas. El acceso a los datos se realiza de forma asíncrona y tolerante a errores: si una carpeta no está disponible en el momento del escaneo, se registra una advertencia en el sistema de trazas, pero el flujo de ejecución continúa.

9.3.2 Integración con ArchivosOBU

ArchivosOBU genera periódicamente los archivos de configuración y contenido que deben ser enviados a los vehículos. Estos archivos se organizan en una estructura de carpetas jerárquica (por ejemplo, por campaña, fecha o tipo de paquete), y contienen la versión esperada de cada elemento por vehículo o por grupo de vehículos.

InfoVersionService recorre estas carpetas para determinar qué versión se supone que debe tener cada OBU en el momento actual. Para ello, los procesadores interpretan los nombres y contenidos de los archivos, extraen metadatos relevantes y devuelven los directorios de los paquetes encontrados para que *PackageService* los analicé y genere los objetos de tipo *BusPackage* y *File*, que luego se utilizan como base de comparación frente a la versión detectada. Esta integración no requiere ninguna modificación en *ArchivosOBU*, ya que se limita a consumir la salida que este genera de forma natural.

9.3.3 Integración con Transfer Manager

El *Transfer Manager*, por su parte, genera carpetas distintas en las que deposita los archivos que han sido realmente instalados en los vehículos tras una operación de sincronización. Estos archivos reflejan el estado

actual del contenido en los OBU y son la fuente principal para determinar la versión real detectada.

A diferencia de *ArchivosOBU*, que trabaja de forma programada, la información proveniente de *Transfer Manager* puede llegar con cierto retardo, de forma desordenada o incompleta. Por tanto, el sistema debe estar preparado para manejar situaciones en las que no se haya recibido aún confirmación de todos los vehículos o en las que falten datos. Esta lógica se encuentra centralizada en los procesadores y en *PackageService*, que crea los objetos de tipo *BusPackage* y *File* asignando esta vez el bInBus a 1, indicando que el paquete se encuentra en activo en el bus.

La integración con *Transfer Manager* también se realiza de forma completamente pasiva, sin necesidad de modificar su comportamiento ni interferir con su ciclo de vida. Esto permite que ambos sistemas evolucionen de forma independiente, manteniendo una arquitectura desacoplada y robusta.

9.4 Exposición de datos - SoaBasicContentManager

Este apartado describe cómo se integró la lógica de consulta desde el sistema SIU, exponiendo los datos generados por *InfoVersionService* mediante un nuevo endpoint RESTFUL implementado dentro del servicio *SoaBasicContentManager*.

Con el objetivo de permitir que los datos generados por InfoVersionService puedan ser consultados por la interfaz de usuario del SIU y por otros posibles sistemas de soporte, fue necesario implementar un nuevo endpoint RESTFUL dentro del backend del SIU, concretamente en el componente SoaBasicContentManager.

Este endpoint actúa como capa de exposición de datos, consultando directamente la base de datos relacional donde InfoVersionService ha dejado registrados los resultados del procesamiento de versiones. De esta forma, se evita cualquier dependencia directa entre el SIU y el servicio autónomo, manteniendo la arquitectura modular y desacoplada.

La implementación consistió en crear un nuevo método HTTP GET que responde a peticiones bajo la ruta /api/LoadFleetStatusRF, el cual devuelve un listado estructurado de objetos DTO que representan el estado de sincronización por vehículo y tipo de paquete. Estos objetos incluyen campos como la matrícula y el sideCode del bus, el tipo de contenido, la versión esperada, la versión detectada, el estado de comparación y la fecha de última comprobación. La estructura de respuesta se diseñó para adaptarse directamente a los requisitos de la tabla en el frontend, evitando necesidad de postprocesamiento.

Se añadió la lógica necesaria para poder mostrar los KPIs requeridos, % buses sincronizados y cantidad exacta de buses sincronizados y no sincronizados.

Internamente, el endpoint utiliza una clase de servicio propia del backend del SIU que se conecta al modelo de datos ya existente, accediendo a las tablas que InfoVersionService actualiza periódicamente. Esta integración se desarrolló siguiendo las convenciones del backend del SIU, reutilizando los patrones existentes para control de errores, validación de permisos y serialización de respuestas.

Gracias a este diseño, el SIU puede consumir los datos de forma transparente, actualizada y con el mínimo acoplamiento posible. Además, este endpoint podría ser reutilizado en el futuro por otras herramientas o paneles, al estar basado en una interfaz REST abierta.

9.5 Frontend

La implementación de la interfaz de usuario se llevó a cabo dentro del propio proyecto frontend del SIU, desarrollado en React y mantenido mediante una arquitectura modular basada en componentes. Dado que el SIU es una plataforma consolidada y en producción, uno de los objetivos principales fue asegurar una integración visual y funcional coherente, reutilizando los estilos, comportamientos y librerías ya disponibles en la plataforma.

El nuevo componente se construyó como una tabla técnica de consulta para operadores, ubicada en el área de administración. Esta tabla muestra la trazabilidad de versiones por vehículo, permitiendo a los usuarios visualizar qué buses tienen instaladas las versiones correctas y cuáles presentan inconsistencias. La estructura

base se desarrolló reutilizando el componente genérico de tabla interactiva del SIU, al que se añadieron las columnas específicas (Véase la sección 8.6).

Para asegurar la flexibilidad de uso, la tabla se complementó con filtros por tipo de paquete, estado y vehículo, así como con la posibilidad de exportar los resultados a Excel. La lógica de interacción y transformación de datos se encapsuló en un contenedor encargado de gestionar el estado, disparar las llamadas al backend y procesar la respuesta para presentarla de forma amigable. Se aplicaron además buenas prácticas como la paginación automática y gestión de errores visibles para el usuario.

Desde el punto de vista técnico, la comunicación entre la interfaz y el backend se implementó mediante llamadas fetch asíncronas al endpoint /api/LoadFleetStatusRF expuesto por SoaBasicContentManager. La respuesta, en formato JSON, se deserializa y transforma en un array de objetos intermedios que alimentan la tabla. El frontend está preparado para manejar errores de red, respuestas vacías o estados incompletos, garantizando así una experiencia de usuario robusta.

Adicionalmente, se implementaron mecanismos para interpretar y representar gráficamente el estado de sincronización mediante iconos e indicadores de color. Esto permite al operador detectar rápidamente problemas sin necesidad de examinar todas las columnas. El diseño se validó internamente por el equipo técnico y se ajustó para mantener la coherencia visual con otros módulos del SIU.

En conjunto, la interfaz desarrollada permite consultar de forma eficiente y visual el estado de sincronización de contenidos, cumpliendo su propósito funcional sin introducir complejidad adicional ni comprometer la estructura ya existente del sistema.

9.5.1 Descripción de la interfaz

La figura 9.1 muestra un ejemplo sobre la interfaz. Consta con un panel de consulta del estado de sincronización de contenidos por vehículo, integrado dentro de la interfaz principal del SIU. Esta vista técnica permite al operador visualizar de forma consolidada el estado de cada vehículo en relación con los distintos tipos de contenido que deben estar correctamente instalados: rutas, patrones, paradas, correspondencias, mensajes y contenidos multimedia.

La tabla central se estructura por filas, cada una correspondiente a un vehículo identificado por su ID interno (sideCode) y su alias visible (matrícula). Las columnas indican para cada tipo de contenido un tick verde en caso de que esté sincronizado o una x roja en caso de que no lo esté. A la izquierda de esta indicación, se muestra un número que indica la configuración conocida por el usuario a la que hace referencia dicha versión. Además de esto, el indicador tiene una badge circular sin contenido que solo muestra un color. Este color indica lo siguiente:

- Gris: Es referido a que la versión es una aplicada con configuración ALL, o lo que es lo mismo, FLEET. Es decir, una configuración global para todas las flotas.
- Azul: Es referido a que la versión es una aplicada con configuración FLEET, que en este caso se refiere a SUBFLEET internamente. Es decir, una configuración para una flota en concreto.
- Morado: Se refiere a las configuraciones de versión PARTICULAR.

En la parte superior se incluye un resumen visual con indicadores clave de flota, como el porcentaje de vehículos correctamente sincronizados, el número total de vehículos OK y los elementos pendientes.

Además, la interfaz ofrece opciones avanzadas de interacción: distintos tipos de filtrado, búsqueda por texto, exportación de resultados y configuración de columnas visibles

La integración visual con el SIU es completa, reutilizando el estilo, colores y componentes ya existentes en el resto de la plataforma. Esto garantiza una experiencia de usuario coherente y facilita la adopción del nuevo módulo por parte del personal técnico.

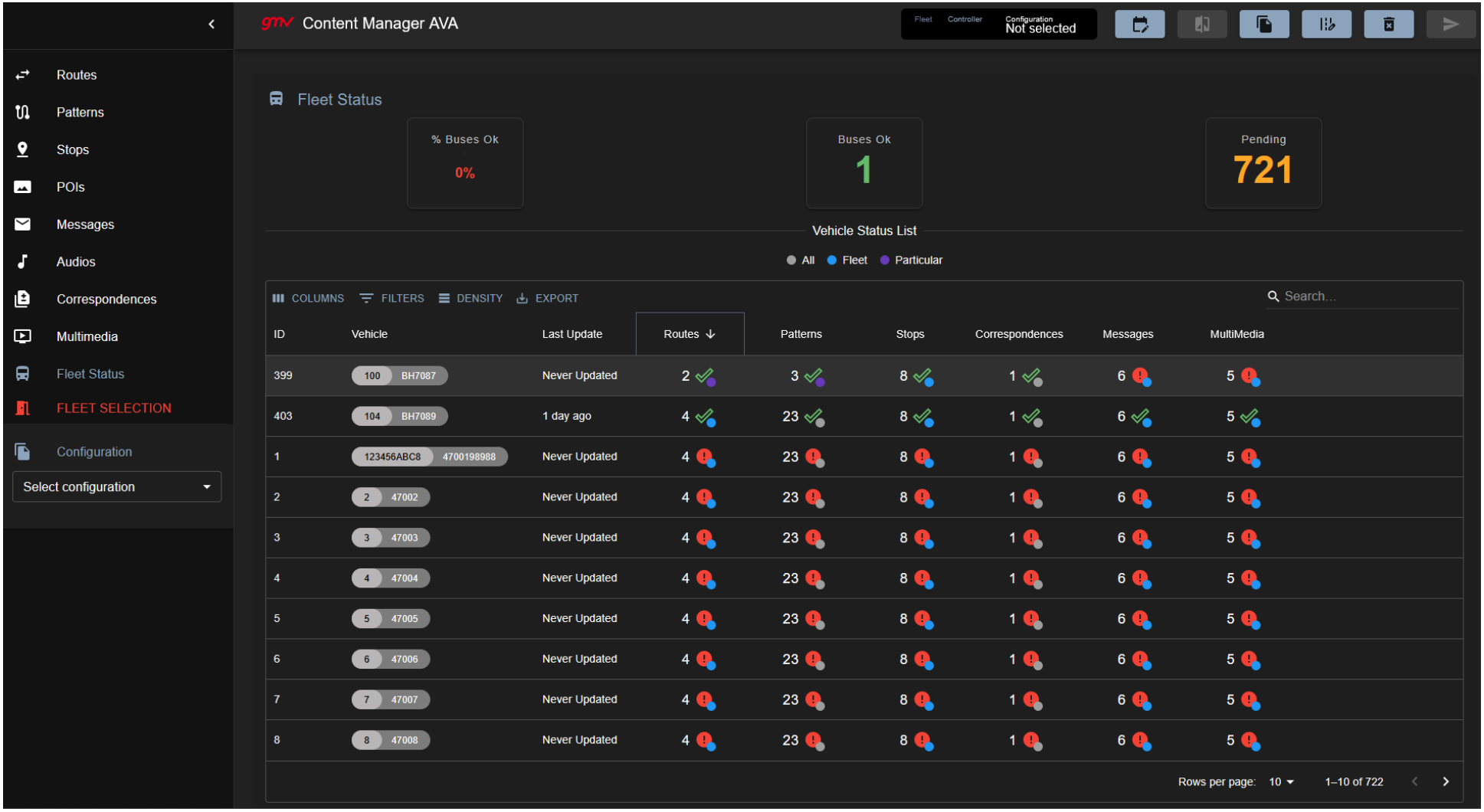


Figura 9.1: Frontend de la nueva funcionalidad

9.6 Gestión de errores y validaciones

La robustez del sistema ha sido una prioridad durante todo el proceso de desarrollo, especialmente al tratarse de un servicio autónomo encargado de analizar información crítica y no controlada directamente, como los archivos generados por *ArchivosOBU* y *Transfer Manager*. Por este motivo, se implementaron múltiples mecanismos de gestión de errores y validaciones, distribuidos tanto en el backend como en la interfaz de usuario, con el objetivo de garantizar la estabilidad del sistema y proporcionar visibilidad ante fallos o datos incoherentes.

En el backend, todas las operaciones que pueden verse afectadas por errores externos (como accesos a rutas de red, lectura de archivos o llamadas a base de datos) se encapsulan en bloques try/catch para evitar que un fallo puntual detenga el procesamiento completo. En cada uno de estos casos, las excepciones son capturadas, y se registra un mensaje detallado mediante el sistema de trazas AqTraza, especificando el tipo de error, su contexto y su origen. Esto permite auditar con precisión los fallos ocurridos, facilitando el diagnóstico sin comprometer la continuidad del servicio. Además, se diferencian explícitamente los mensajes de advertencia (por ejemplo, carpetas vacías o rutas no encontradas) de los errores críticos, lo que permite priorizar su tratamiento.

En cuanto a la validación, *InfoVersionService* incluye controles internos para comprobar que las rutas existen, que los archivos son accesibles, y que la estructura esperada de las carpetas y ficheros se mantiene. Por ejemplo, si se encuentra una carpeta con nombre no válido, se descarta del procesamiento y se registra como incidencia. Asimismo, antes de insertar versiones en base de datos, se valida que los campos mínimos requeridos estén presentes (identificador de vehículo, tipo de paquete y versión).

En el frontend también se aplican validaciones y gestión de errores a distintos niveles. En primer lugar, la llamada al endpoint REST del backend se realiza de forma asíncrona, y está preparada para detectar errores de red, respuestas mal formadas o estados no esperados. En caso de error, se muestra al usuario técnico un mensaje informativo no intrusivo, que le permite reintentar o continuar con la navegación.

En caso de producirse errores críticos, como la falta de acceso a rutas de red o problemas de escritura en base de datos, el sistema está diseñado para no detener su ejecución. Los errores se registran mediante trazas detalladas en el sistema de logging, y el servicio continúa su ejecución con el siguiente ciclo programado. Esto garantiza una alta disponibilidad y evita la interrupción completa del proceso ante fallos puntuales, favoreciendo una operación más resiliente en entornos reales.

Por último, se han implementado validaciones visuales en los datos mostrados, destacando de forma clara las inconsistencias o desincronizaciones mediante colores e iconos. Esto permite al operador identificar posibles problemas sin necesidad de analizar manualmente las versiones.

En conjunto, todos estos mecanismos aseguran que el sistema pueda funcionar de forma autónoma, fiable y predecible, incluso ante escenarios incompletos, errores de entrada o fallos temporales en las herramientas externas. Además, la visibilidad ofrecida por las trazas y los indicadores visuales facilita enormemente el mantenimiento y la supervisión del sistema por parte de los operadores.

9.7 Integración continua y gestión del código

El desarrollo del sistema se realizó siguiendo un modelo de integración continua dentro del entorno de trabajo de GMV, utilizando herramientas corporativas como *Bitbucket* para el control de versiones y *Jenkins* como servidor de automatización para la ejecución de tareas asociadas a los commits y despliegues.

Durante el ciclo de desarrollo, cada funcionalidad o corrección fue implementada en una rama independiente, siguiendo la convención *feature/, hotfix/ o release/*, y posteriormente integrada en la rama principal (*master*) mediante pull requests. Cada pull request requería una revisión por parte de otro miembro del equipo, lo que garantizaba la calidad del código, la conformidad con las convenciones del proyecto y la detección temprana de posibles errores o duplicidades.

El repositorio estaba alojado en Bitbucket Server, lo que permitió una gestión estructurada del código fuente, con control de permisos, histórico de versiones y seguimiento de incidencias. Se utilizaron etiquetas y comentarios en las revisiones para facilitar la trazabilidad de cada cambio, así como para documentar las decisiones tomadas durante el desarrollo.

Por otro lado, el sistema estaba integrado con un servidor *Jenkins*, encargado de ejecutar pipelines automáticos cada vez que se realizaban integraciones en la rama principal. Estos pipelines incluían tareas como la compilación del servicio (*InfoVersionService*), la ejecución de pruebas automáticas, el análisis estático de código y la generación de artefactos preparados para despliegue. En caso de errores en la compilación o fallos en los tests, *Jenkins* notificaba automáticamente al responsable del commit para su revisión.

Este enfoque de integración continua permitió mantener un flujo de trabajo estable, detectar errores de forma temprana y reducir significativamente el tiempo entre el desarrollo y la validación. Además, la trazabilidad completa de cada cambio facilita el mantenimiento a largo plazo y la colaboración entre distintos miembros del equipo.

Capítulo 10

Pruebas

Una vez finalizada la implementación de los distintos componentes del sistema, se procedió a validar su correcto funcionamiento mediante una serie de pruebas distribuidas en distintos niveles. El objetivo de este proceso fue verificar que el sistema se comporta conforme a los requisitos funcionales y no funcionales establecidos, así como garantizar su robustez ante situaciones inesperadas o datos erróneos.

Las pruebas realizadas incluyen desde validaciones unitarias de componentes clave hasta pruebas de integración completas entre los módulos del sistema, incluyendo la base de datos, el servicio de procesamiento (*InfoVersionService*) y la interfaz gráfica integrada en el *SIU*. Adicionalmente, se llevaron a cabo pruebas manuales y funcionales con escenarios simulados para asegurar que la información mostrada en el panel de versiones refleja fielmente la situación real de cada vehículo.

En los apartados siguientes se detallan los distintos tipos de pruebas realizadas, su metodología, los resultados obtenidos y los criterios aplicados para determinar la validez de cada uno de los bloques del sistema.

10.1 Pautas de Estilo

En la implementación de las pruebas, así como en el desarrollo general del sistema, se aplicaron diversas pautas de estilo con el objetivo de mejorar la cohesión, la legibilidad, la consistencia estructural y la mantenibilidad del código fuente. Estas buenas prácticas no solo facilitaron la escritura y depuración del código durante el desarrollo, sino que también permiten a futuros desarrolladores comprender y ampliar la lógica con mayor facilidad. Estas son algunas de las principales pautas seguidas:

- **Nombres descriptivos**: Se utilizaron identificadores claros y significativos para clases, métodos, variables y archivos de test, facilitando su comprensión sin necesidad de revisar la implementación interna.
- **Separación de responsabilidades**: Cada método de prueba se centró en validar un único comportamiento específico, siguiendo el principio *Arrange-Act-Assert* para mantener una estructura coherente. Este principio hace que los métodos de prueba se dividan en tres partes:
    - **Arrange:** es aquella parte en la que los datos son preparados para utilizarse en las pruebas
    - **Act:** es aquella parte en la uqe se invoca el método o parte de código que se quiere probar.
    - **Assert:** es aquella parte en al que se verifiacn los resultados de los métodos activados en la parte *Act*.

  Estas partes deberán estar indicadas en el código del método de prueba por medio de comentarios.
- **Uso de mocks y stubs**: Se emplearon objetos simulados para aislar dependencias externas, evitando efectos colaterales y garantizando que las pruebas unitarias fueran deterministas y reproducibles.

- **Asserts claros y específicos**: En lugar de realizar múltiples verificaciones en una única prueba, se fragmentaron en pruebas más pequeñas y precisas, facilitando la localización de errores en caso de fallo.

- **Reutilización de lógica auxiliar**: Se agruparon funciones comunes de inicialización o creación de objetos mock en clases base o métodos utilitarios compartidos, evitando duplicación innecesaria.

- **Formato y convenciones consistentes**: Se respetaron las reglas de estilo del equipo (Véase la sección 9.1), aplicando herramientas de linting y formato automático en el IDE.

10.2 Pruebas Unitarias

Las pruebas unitarias se centraron en validar el comportamiento de los componentes críticos del sistema de forma aislada, especialmente aquellos que contenían lógica de negocio independiente de la infraestructura. El objetivo fue asegurar que cada clase, método o funcionalidad ejecutaba correctamente su responsabilidad, y reaccionaba adecuadamente ante entradas válidas, valores extremos o condiciones inesperadas.

Las pruebas se realizaron principalmente sobre clases del servicio *InfoVersionService*, en particular sobre los procesadores, el servicio de empaquetado (*PackageService)* de versiones, y las clases auxiliares de validación de versiones o generación de rutas. Dado que muchas de estas clases fueron diseñadas para trabajar mediante interfaces y con dependencia explícita de servicios externos (por ejemplo, acceso a disco o base de datos), fue posible aplicar fácilmente técnicas de mocking para aislar las pruebas.

Las pruebas se desarrollaron utilizando el entorno de testeo integrado de Visual Studio y el framework de pruebas por defecto de .NET, xUnit. Las aserciones comprobaban tanto los valores devueltos como los efectos secundarios esperados (por ejemplo, la inserción de versiones o el rechazo de carpetas con nombres inválidos).

Un ejemplo concreto de prueba unitaria implementada fue la validación de métodos como GetFoldersToProcess en entornos simulados, comprobando que el sistema respondía correctamente ante rutas inexistentes, carpetas vacías o nombres mal formateados.

Estas pruebas fueron fundamentales para poder refactorizar e introducir nuevas funcionalidades sin comprometer el comportamiento ya establecido, además de servir como documentación viva del comportamiento esperado del sistema.

Además de la validación individual de comportamientos, se definió una estrategia general de cobertura, basada en los principios de testeo en componentes desacoplados y priorización por criticidad. Se dio mayor énfasis a clases con lógica de decisión (procesadores y servicios), minimizando el número de tests sobre utilidades triviales.

Para asegurar la calidad de los tests, se cumplió con los siguientes criterios:

- Las pruebas deben ejecutarse de forma determinista, sin depender del estado del sistema.

- El resultado de la prueba debe ser binario (éxito o fallo claro), sin necesidad de interpretación ambigua.

También se tuvo en cuenta la aplicabilidad del enfoque **test-first** en ciertos métodos clave, especialmente durante el desarrollo de *PackageService*, asegurando que el comportamiento del método se ajustara desde el principio a los requisitos funcionales.

Para mantener las pruebas automatizadas, se integraron en el entorno de desarrollo de Visual Studio, permitiendo su ejecución continua en local, y se incluyeron en el pipeline de Jenkins dentro del sistema de integración continua de GMV. Esto garantiza que ningún cambio en el código principal pueda incorporarse a ramas estables sin pasar por las pruebas correspondientes.

Un ejemplo de una de estas pruebas se puede ver en el código 10.1. No interpretar este código como el real de la aplicación, ya que esto es un mero ejemplo de uno de los tests.

```
[Fact]
public void GetFoldersToProcess_ShouldReturnOnlyValidDirectories()
{
    // Arrange
    var mockFileSystem = new Mock<IFileSystem>();
    mockFileSystem.Setup(fs => fs.GetDirectories("/base"))
        .Returns(new[] { "/base/valid1", "/base/empty", "/base/invalid" });

    mockFileSystem.Setup(fs =>
        fs.DirectoryExists("/base/valid1")).Returns(true);
    mockFileSystem.Setup(fs =>
        fs.DirectoryExists("/base/empty")).Returns(true);
    mockFileSystem.Setup(fs =>
        fs.DirectoryExists("/base/invalid")).Returns(false);

    mockFileSystem.Setup(fs => fs.HasContent("/base/valid1")).Returns(true);
    mockFileSystem.Setup(fs => fs.HasContent("/base/empty")).Returns(false);

    var folderService = new FolderService(mockFileSystem.Object);

    // Act
    var result = folderService.GetFoldersToProcess("/base");

    // Assert
    Assert.Single(result);
    Assert.Contains("/base/valid1", result);
}
```

Listing 10.1: Ejemplo de prueba unitaria

10.2.1 Cobertura de la aplicación

En esta sección se analiza el grado de cobertura alcanzado por las pruebas automáticas desarrolladas. Dado que el nuevo endpoint REST y el componente de interfaz en el frontend presentan una cobertura completa del nuevo código incorporado (100 % de las líneas modificadas o añadidas han sido verificadas mediante pruebas), el análisis se centrará principalmente en el núcleo funcional del sistema: el servicio *InfoVersionService*.

Este módulo concentra la mayor parte de la lógica de negocio y procesamiento de datos, y por tanto representa el componente más crítico en términos de fiabilidad y robustez. Se detallarán a continuación los porcentajes de cobertura alcanzados en las clases principales (Véase el cuadro 10.1), así como los criterios utilizados para seleccionar los bloques de código sujetos a validación mediante pruebas unitarias.

Paquete	Líneas cubiertas	Líneas a cubrir	Lineas totales	Porcentaje
Application	340	343	779	99.1 %
Architecture	392	430	709	91.1 %
Domain	38	38	50	100 %
Configuration	51	51	95	100 %
ServiceHost	20	20	32	100 %
TOTAL	841	882	1665	95.35 %

Cuadro 10.1: Coverage del servicio *InfoVersionService*

Criterios para la selección del código sujeto a pruebas unitarias

La selección de bloques de código para ser cubiertos mediante pruebas unitarias no se realizó de forma arbitraria, sino siguiendo una serie de criterios técnicos y de valor añadido que aseguran que el esfuerzo de testeo se focaliza sobre los puntos más relevantes del sistema:

- **Complejidad lógica**: se priorizaron métodos que incluyeran condiciones, bifurcaciones (*if*, *switch*), estructuras de iteración o lógica de validación. Cuanto mayor era la complejidad del fragmento, mayor fue su prioridad para ser cubierto.

- **Impacto funcional**: se dio prioridad a aquellos componentes cuyo fallo pudiera afectar de forma crítica al funcionamiento global del sistema, como los encargados de filtrar los directorios, crear los paquetes, generar datos para insertar en base de datos o preparar estructuras de respuesta.

- **Facilidad de desacoplamiento**: en los casos en los que ciertas clases no estaban diseñadas inicialmente para ser testeables (por ejemplo, acopladas directamente a estructuras estáticas), se propuso su refactorización hacia una estructura más modular y testable.

- **Volatilidad esperada**: se cubrieron también componentes que se espera que puedan cambiar o escalar en el futuro, para facilitar su refactorización sin riesgo de regresión funcional.

Este enfoque permitió obtener una cobertura coherente, centrada en maximizar el valor de las pruebas, en lugar de perseguir únicamente métricas cuantitativas. De esta forma, se garantiza que los elementos más sensibles del sistema están protegidos ante errores y se refuerza la calidad del código base.

Por último, cabe destacar que el umbral de cobertura recomendado internamente por GMV para desarrollos de este tipo se sitúa por encima al 75 %. En este proyecto, se ha alcanzado un **95.35 %** de cobertura total sobre el módulo *InfoVersionService*, lo que supone un valor significativamente superior al mínimo esperado. Este alto porcentaje no solo se refleja en volumen, sino también en calidad, ya que —como se ha indicado anteriormente— la cobertura se concentra en las clases con mayor complejidad lógica y relevancia funcional dentro del sistema.

10.3 Pruebas de Integración

Además de las pruebas unitarias, se realizaron pruebas de integración para validar el comportamiento conjunto de los diferentes componentes del sistema. Estas pruebas fueron fundamentales para asegurar que la lógica desarrollada en el servicio *InfoVersionService* interactúa correctamente con los recursos externos como la base de datos, las carpetas generadas por *ArchivosOBU* y *Transfer Manager*, así como con el backend del *SIU* a través del nuevo endpoint implementado.

El objetivo principal de estas pruebas fue comprobar la coherencia del flujo completo, desde la lectura de carpetas reales hasta la persistencia de datos y su posterior consulta desde la interfaz web. Para ello, se simularon escenarios realistas en un entorno de desarrollo controlado, utilizando estructuras de carpetas reales con archivos representativos, así como un entorno de base de datos parcialmente poblado con datos de prueba.

Uno de los casos más representativos fue la integración entre los procesadores del sistema (*FleetProcessor, TMProcessor*, etc.) y el servicio de creación de paquetes de versiones (*PackageService*). Se verificó que los procesadores detectaban correctamente los paquetes disponibles en las rutas compartidas, los analizaban conforme al formato esperado y generaban entradas válidas para ser insertadas en base de datos. Véase la figura 10.1 y 10.2 para ver los resultados al consultar las tablas de la base de datos.

Results Messages

	iIdPaquete	iIdAutobus	iIdTipoPaquete	bEnBus	iVersionPaquete	dtFechaRegistro
1	648	403	3	1	2	2025-06-27 09:54:24.713
2	647	401	3	1	0	2025-06-27 09:54:24.620
3	646	403	2	1	5	2025-06-27 09:54:24.603
4	645	402	2	1	2	2025-06-27 09:54:24.583
5	644	401	2	1	2	2025-06-27 09:54:24.567
6	643	723	2	0	5	2025-06-27 09:54:24.523
7	642	722	2	0	5	2025-06-27 09:54:24.510
8	641	721	2	0	5	2025-06-27 09:54:24.493
9	640	720	2	0	5	2025-06-27 09:54:24.480
10	639	719	2	0	5	2025-06-27 09:54:24.467
11	638	718	2	0	5	2025-06-27 09:54:24.457
12	637	717	2	0	5	2025-06-27 09:54:24.443
13	636	716	2	0	5	2025-06-27 09:54:24.417
14	635	715	2	0	5	2025-06-27 09:54:24.397
15	634	714	2	0	5	2025-06-27 09:54:24.380
16	633	713	2	0	5	2025-06-27 09:54:24.367
17	632	712	2	0	5	2025-06-27 09:54:24.353
18	631	711	2	0	5	2025-06-27 09:54:24.340
19	630	710	2	0	5	2025-06-27 09:54:24.323
20	629	709	2	0	5	2025-06-27 09:54:24.310
21	628	708	2	0	5	2025-06-27 09:54:24.297
22	627	707	2	0	5	2025-06-27 09:54:24.280

Figura 10.1: Paquetes resultado de las pruebas de integracion

	sNombre	iIdPaquete	iVersion
1	line_user_amp.dat.zip	646	2
2	par_amp.dat.zip	646	1
3	par_user_amp.dat.zip	646	2
4	trav_amp.dat.zip	646	1
5	trav_user_amp.dat.zip	646	3
6	trav_user_amp.dat.zip	645	3
7	trav_amp.dat.zip	645	1
8	par_user_amp.dat.zip	645	2
9	par_amp.dat.zip	645	1
10	line_user_amp.dat.zip	645	2
11	line_user_amp.dat.zip	644	2
12	par_amp.dat.zip	644	1
13	par_user_amp.dat.zip	644	2
14	trav_amp.dat.zip	644	1
15	trav_user_amp.dat.zip	644	3
16	trav_amp.dat.zip	643	1
17	par_user_amp.dat.zip	643	8
18	par_amp.dat.zip	643	1
19	par_amp.dat.zip	642	1
20	par_user_amp.dat.zip	642	8
21	trav_amp.dat.zip	642	1
22	trav_amp.dat.zip	641	1

Figura 10.2: Archivos resultado de las pruebas de integracion

Posteriormente, se validó que la tabla de resultados reflejaba correctamente esta información al consultarse desde el endpoint expuesto por *SoaBasicContentManager*.

También se realizaron pruebas sobre el ciclo completo de limpieza de registros antiguos, confirmando que el sistema eliminaba correctamente los paquetes cuya antigüedad excedía el umbral configurado, **sin afectar a las entradas activas o recientes**.

En lo relativo a la capa de presentación, se confirmó que los datos mostrados en la tabla de versiones del SIU correspondían exactamente con los datos almacenados en base de datos. Se probaron filtros, ordenaciones y combinaciones de datos, asegurando que no se produjeran inconsistencias entre lo visualizado y lo realmente procesado.

Las pruebas de integración se realizaron de forma manual con validaciones cruzadas en base de datos y en los archivos del sistema, pero también se prepararon scripts reutilizables que permitieron automatizar algunos de los casos más críticos.

Estas pruebas confirmaron que la solución propuesta funciona como un conjunto cohesionado, y que todos los módulos interactúan correctamente en condiciones reales de uso, sin dependencia directa entre ellos, lo que facilita el mantenimiento y la escalabilidad futura del sistema.

10.4 Pruebas funcionales

Las pruebas funcionales se llevaron a cabo con el objetivo de verificar que el sistema desarrollado cumple con los requisitos funcionales definidos previamente en el capítulo de análisis. Estas pruebas validan la lógica completa de los flujos descritos en los casos de uso, desde la ejecución automática del procesamiento de archivos hasta la consulta por parte del operador en la interfaz del SIU.

Dado que el sistema está dividido en tres bloques principales —servicio de procesamiento (*InfoVersionService*), API de exposición (nuevo endpoint en *SoaBasicContentManager*) y frontend del *SIU*— las pruebas funcionales cubrieron el comportamiento de extremo a extremo, asegurando que los datos generados por el backend fueran consistentes con los resultados presentados al usuario.

Para cada caso de uso definido, se prepararon escenarios de prueba reales, utilizando archivos generados por *ArchivosOBU* y *Transfer Manager* en una estructura simulada, y registros iniciales en base de datos controlados. A continuación se detallan las validaciones realizadas por cada uno de los casos de uso:

10.4.1 CU1 - Registrar versiones

Se validó que, al introducir nuevos archivos de configuración en las rutas monitorizadas por *InfoVersionService*, el sistema los detecta automáticamente tras el intervalo de espera configurado. Se comprobó que las versiones extraídas se correspondían con los contenidos reales, que los registros eran correctamente insertados en base de datos, y que el sistema omitía archivos mal formateados o carpetas sin cambios. Además, se probaron situaciones de error controlado (archivos corruptos o nombres inválidos), verificando que el sistema las registraba como trazas sin interrumpir la ejecución.

10.4.2 CU2 - Comparación de versiones generales

Desde la interfaz del *SIU*, se simuló el comportamiento de un operador solicitando la comparación de versiones. Se probaron distintos filtros (por tipo de paquete, por estado de sincronización, por OBU) y se comprobó que el sistema mostraba correctamente las inconsistencias entre versiones esperadas y detectadas, representadas con iconos visuales e indicadores de estado. Se incluyó también una validación de comportamiento ante ausencia de datos (tabla vacía con mensaje informativo).

10.4.3 CU3 – Consulta específica de versiones

Esta prueba consistió en verificar que, al seleccionar un vehículo concreto, el sistema mostraba de forma desglosada las versiones de cada tipo de contenido instalado, junto con su correspondiente versión esperada.

Se validó que la información aparecía de forma clara y precisa, y que los datos mostrados coincidían con los registros existentes en base de datos. También se verificó la respuesta del sistema cuando no existían registros para un vehículo determinado.

10.4.4 CU4 – Consulta de KPIs de la flota

Se probó la funcionalidad de resumen estadístico del estado de sincronización de la flota. En particular, se verificó el cálculo correcto del porcentaje de vehículos sincronizados, el total de vehículos afectados, y la correcta visualización de estos KPIs en la cabecera del módulo. También se realizaron pruebas sobre campañas con datos incompletos o inconsistentes, confirmando que los cálculos se ajustaban a los datos disponibles sin producir errores en la visualización.

10.4.5 Conclusión

En todas las pruebas funcionales se respetaron los permisos de acceso definidos para el operador, validando que el acceso al módulo está restringido a usuarios autenticados y con los permisos habilitados. Todas las acciones fueron validadas manualmente y cruzadas con los datos reales insertados en base de datos, asegurando la trazabilidad del comportamiento del sistema frente a los requisitos funcionales previamente definidos.

En el cuadro 10.2 se puede ver un pequeño resumen de los CU y las pruebas funcionales realizadas.

Código	**Nombre del caso de uso**	**Pruebas funcionales realizadas**
CU1	Registrar versiones	Inserción automática de versiones desde carpetas. Comprobación de inserción en base de datos, validación de detección de cambios, exclusión de carpetas sin actualizaciones o con errores.
CU2	Comparación de versiones	Verificación de las inconsistencias entre versiones esperadas y detectadas, aplicación de filtros, visualización de estados con iconos, tabla vacía cuando no hay coincidencias.
CU3	Consulta específica de versiones	Visualización detallada por vehículo: tipo de contenido, versión detectada y versión esperada. Verificación con registros reales. Gestión de casos sin datos.
CU4	Consulta de KPIs de la flota	Validación de indicadores: porcentaje de sincronización, vehículos afectados, estadísticas por tipo de contenido. Comprobación de resultados parciales o faltantes.

Cuadro 10.2: Trazabilidad entre requisitos funcionales y pruebas realizadas

10.5 Pruebas de rendimiento

Dado el contexto real de uso del sistema en flotas de gran tamaño, uno de los objetivos clave ha sido asegurar que el servicio mantiene un rendimiento aceptable, escalable y predecible incluso con volúmenes elevados de datos. Esta necesidad queda reflejada directamente en varios de los requisitos no funcionales definidos, como RNF05, RNF06 y RNF07, que establecen umbrales de rendimiento concretos: menos de 120 segundos para el procesamiento global, menos de 1 segundo por vehículo y soporte para más de 1000 vehículos sin degradación significativa.

Para validar estos objetivos, se diseñaron y ejecutaron pruebas de rendimiento simulando entornos con datos reales generados por *ArchivosOBU* y *Transfer Manager*. Se crearon estructuras de carpetas que representaban distintos tamaños de flota (100, 500, 1000 y 1500 vehículos), replicando versiones, archivos multimedia y configuraciones diversas, a fin de obtener una visión representativa de la carga real esperada.

Inicialmente, durante el desarrollo del sistema, se detectó que al incrementar el volumen de paquetes generados (por encima de los 200.000 **contenidos**), la inserción en base de datos se volvía significativamente más lenta, y el proceso completo podía superar los límites establecidos. Como resultado de este análisis, se introdujeron varias optimizaciones orientadas al rendimiento, entre ellas:

- **Paralelización del procesamiento** mediante múltiples *IProcessor* ejecutados en serie pero desacoplados, evitando cuellos de botella al tratar carpetas independientes. Además, se paralelizaron los distintos bucles que recorren las carpetas y las líneas del archivo de control de versiones.
- **Inserciones en lote (bulk insert)** para reducir el número de transacciones individuales y minimizar la latencia de comunicación con la base de datos.
- **Indexación en las tablas clave** para acelerar las consultas posteriores realizadas por el frontend y el servicio de comparación de versiones.

Tras aplicar estas mejoras, se repitieron las pruebas con el conjunto más exigente (más de 1500 vehículos y 20000 contenidos, llegando en algunos casos a superar el millón de archivos de contenido procesables). Los resultados obtenidos demostraron que en un **89 % de las ejecuciones**, el tiempo medio de procesamiento por vehículo se situó en torno a los **520 ms**, mientras que el tiempo total de análisis completo de toda la flota no llegó al umbral.

En el pequeño porcentaje restante de ejecuciones, los tiempos se vieron afectados ligeramente por picos de uso del sistema, pero sin llegar a comprometer los umbrales definidos en los requisitos.

Estas pruebas permiten afirmar que el sistema no solo es funcionalmente correcto, sino también apto para ser desplegado en producción con garantías de rendimiento, incluso en escenarios de alta demanda o condiciones operativas exigentes.

10.6 Validación con usuarios finales

Además de las pruebas técnicas desarrolladas en entornos locales y controlados, el sistema ha sido validado en un entorno de preproducción real dentro de GMV. Durante esta fase, se integró el servicio InfoVersionService con instancias operativas de ArchivosOBU y Transfer Manager, así como con una base de datos representativa.

Operadores técnicos del equipo de validación accedieron al nuevo módulo de trazabilidad desde el SIU y realizaron diversas comprobaciones funcionales y de usabilidad. Esta validación permitió confirmar que la información mostrada en la interfaz coincidía con los registros reales de versiones detectadas en los vehículos, y que el sistema respondía correctamente ante diferentes escenarios de consulta y sincronización.

Las pruebas en entorno preproductivo confirmaron también que la solución era compatible con los flujos operativos existentes, sin afectar negativamente a otros módulos del sistema, y aportando una mejora tangible en la visibilidad del estado de actualización de la flota.

Parte IV

Conclusiones

Capítulo 11

Conclusiones y trabajo futuro

11.1 Introducción

El desarrollo de este Trabajo Fin de Grado ha supuesto una experiencia completa y desafiante, tanto desde el punto de vista técnico como desde el punto de vista profesional. El sistema diseñado y construido cumple con el objetivo inicial de mejorar la trazabilidad de versiones de contenidos en sistemas de transporte inteligente, solucionando una carencia identificada durante el periodo de prácticas en la empresa GMV.

A lo largo del proyecto se ha abordado la problemática desde un enfoque integral, analizando el flujo completo de generación, transferencia y validación de contenidos, y proponiendo una arquitectura escalable, desacoplada y compatible con los sistemas existentes. Se han desarrollado distintos componentes software: un servicio independiente de análisis, un endpoint REST de consulta y un módulo visual de exploración de versiones integrado en el Gestor de Contenidos (SIU), todo ello respetando las restricciones operativas y tecnológicas de la empresa.

La solución ha sido validada tanto funcional como técnicamente, con un elevado grado de cobertura en pruebas y unos resultados de rendimiento que superan los requisitos establecidos. Además, se ha prestado especial atención a la calidad del código, la estructura modular y la mantenibilidad a largo plazo.

11.2 Aportaciones realizadas

Las principales aportaciones de este proyecto se pueden dividir en dos dimensiones: técnicas y organizativas.

Desde el punto de vista técnico, se ha diseñado e implementado un nuevo servicio de análisis de versiones (*InfoVersionService*) capaz de interpretar el estado real de cada vehículo a partir de los datos generados por ArchivosOBU y Transfer Manager. Este componente trabaja de forma autónoma y expone resultados reutilizables para otros sistemas, como el backend del SIU o futuras herramientas de análisis de versiones.

Asimismo, se ha implementado un nuevo endpoint REST en el backend corporativo (SoaBasicContentManager) y una interfaz visual para operadores que permite consultar el estado de sincronización de cada vehículo, visualizar versiones esperadas y reales, y extraer conclusiones a partir de KPIs agregados.

A nivel organizativo, se ha documentado todo el flujo de integración con un sistema existente y complejo, demostrando cómo se puede introducir nueva funcionalidad sin alterar el comportamiento de los sistemas productivos actuales. También se ha seguido un enfoque de trabajo profesional, con control de versiones, revisiones mediante pull requests, uso de integración continua con Jenkins, y pruebas automatizadas dentro del pipeline de desarrollo.

11.3 Valoración del resultado

El resultado final puede considerarse muy satisfactorio. El sistema propuesto resuelve de forma efectiva el problema inicial identificado (ausencia de trazabilidad automatizada de versiones), y lo hace sin introducir complejidad innecesaria ni dependencias críticas entre componentes.

Las pruebas funcionales, unitarias e integradas confirman que la solución cumple con todos los requisitos definidos. Las pruebas de rendimiento, por su parte, evidencian que el sistema es capaz de escalar a volúmenes reales de operación con tiempos de respuesta muy por debajo de los umbrales establecidos.

Además, la arquitectura propuesta sienta las bases para futuras ampliaciones, y el código desarrollado se ha estructurado de forma clara, siguiendo buenas prácticas, con una cobertura de pruebas superior al 95 %.

11.4 Mejoras a futuro

A pesar de los buenos resultados obtenidos, se han identificado varias líneas de trabajo futuro que podrían aportar valor añadido al sistema:

- **Sistema de alertas**: incorporar notificaciones automáticas (por correo, dashboard o logs activos) en caso de que se detecten inconsistencias críticas o múltiples vehículos desincronizados.
- **Histórico de versiones**: permitir la consulta de versiones pasadas o la evolución histórica de un vehículo a lo largo del tiempo, lo que podría resultar útil para diagnósticos o auditorías.
- **Exportación e integración externa**: exponer un API externo para que otras herramientas puedan consultar el estado de actualización de los vehículos desde otras plataformas o integraciones.
- **Paneles avanzados de visualización**: extender la interfaz de usuario con gráficos y métricas más visuales, facilitando el análisis global de flota.
- **Desacoplamiento completo del SIU**: evaluar, en una siguiente fase, la posibilidad de que el nuevo módulo opere de forma completamente independiente del Gestor de Contenidos actual.

Además, cabe destacar que la arquitectura del sistema, basada en procesadores desacoplados y rutas parametrizadas, permite su aplicación más allá del Gestor de Contenidos (SIU). Dado que el servicio se alimenta directamente de archivos generados por herramientas como ArchivosOBU o Transfer Manager, cualquier otro sistema de transporte que utilice estos mismos mecanismos de distribución de contenido podría beneficiarse directamente de la solución propuesta. Incluso sería posible extender el sistema mediante nuevos procesadores específicos para entornos diferentes, como por ejemplo otros sistemas de configuración que generen versiones en estructuras de carpetas propias, manteniendo la lógica común de validación, registro y consulta ya implementada.

Estas mejoras pueden abordarse de forma incremental y modular, aprovechando la arquitectura ya diseñada. Su desarrollo supondría un paso adelante en la digitalización del control de versiones en sistemas embarcados dentro del contexto de transporte inteligente.

A mayores de las mejoras mencionadas, también sería viable plantear líneas de evolución más ambiciosas. Por ejemplo, se podría integrar el sistema con módulos de mantenimiento predictivo o análisis de estado del vehículo, utilizando los datos de versiones como indicadores de consistencia técnica. Otra posible extensión sería la incorporación de inteligencia artificial para detectar patrones de desincronización y anticiparse a fallos recurrentes en determinados nodos de la flota. Estas ideas abren la puerta a una trazabilidad proactiva y a una operación más robusta y autónoma.

11.5 Objetivos personales

Además de los logros técnicos y funcionales alcanzados con el desarrollo de este proyecto, se han cumplido satisfactoriamente los objetivos personales establecidos al inicio del Trabajo Fin de Grado. El trabajo realizado ha supuesto una oportunidad para consolidar y aplicar de forma práctica los conocimientos adquiridos durante la carrera en un entorno profesional real, enfrentando problemáticas reales y aportando soluciones en el marco operativo de una empresa tecnológica.

Uno de los principales aprendizajes ha sido la profundización en conceptos de diseño de software modular, arquitecturas distribuidas y estrategias de prueba automatizada. El hecho de trabajar sobre un sistema complejo ya desplegado, con múltiples dependencias técnicas y restricciones organizativas, ha permitido comprender en primera persona las implicaciones reales del mantenimiento evolutivo, así como la importancia de mantener la cohesión y estabilidad de una solución en producción.

Asimismo, a lo largo del proyecto se han desarrollado competencias transversales esenciales en cualquier entorno de desarrollo profesional: la planificación efectiva de tareas, la organización del tiempo, la adopción de metodologías ágiles como Scrum y el uso riguroso de herramientas de control de versiones (Git, Bitbucket), gestión de proyectos (Jira) e integración continua (Jenkins). La participación en procesos de revisión de código y colaboración con otros equipos ha contribuido de forma notable a mejorar la capacidad de comunicación técnica y el sentido de responsabilidad dentro de un flujo de trabajo profesional.

Además, el proyecto ha brindado la oportunidad de adquirir experiencia en la integración de nuevas funcionalidades en un sistema grande y en producción, aprendiendo a diseñar soluciones no intrusivas, compatibles y sostenibles. Esta experiencia ha sido especialmente valiosa para desarrollar habilidades de análisis funcional, toma de decisiones técnicas y resolución de problemas en escenarios con restricciones reales.

Por último, el contacto diario con tecnologías utilizadas en GMV —como C#, servicios Windows, APIs REST, React o el servidor IIS— ha permitido profundizar y afianzar el conocimiento en un stack tecnológico moderno y demandado, cumpliendo así con uno de los objetivos formativos clave planteados al comienzo del proyecto.

En conjunto, todas estas experiencias suponen una base sólida tanto a nivel técnico como profesional, y permiten afrontar con mayor preparación los futuros retos en el ámbito académico, laboral o de especialización tecnológica.

Appendices

Apéndice A

Manual de Instalación

En este capítulo se describe todo lo necesario para la instalación y despliegue del proyecto. Cabe destacar que el presente manual únicamente será útil para el personal interno de GMV y los clientes que soliciten esta funcionalidad, ya que para poder utilizarlo es necesario el acceso a otras aplicaciones exclusivas. Algunos de los datos como rutas de los repositorios serán omitidos por cuestiones de privacidad.

A.1 Requisitos previos

Para la instalación y puesta en marcha del sistema desarrollado, se requiere el siguiente entorno y dependencias:

- Sistema operativo: Windows 10 o superior (compatible con servicios de Windows).
- .NET SDK 6.0 o superior (para compilar).
- .NET Runtime 6.0 (para ejecutar).
- SQL Server
- IIS (solo si se despliega el nuevo endpoint como aplicación web ya que SoaBasicContentManager es un IIS).
- Acceso a las rutas compartidas utilizadas por ArchivosOBU y Transfer Manager.
- Permisos de escritura/lectura en las carpetas de contenido.
- Git

A.2 Instalación del servicio InfoVersionService

A.2.1 Compilación

1. Clonar el repositorio desde Bitbucket.
2. Abrir la solución *InfoVersionService.sln* en Visual Studio 2022 o superior.
3. Seleccionar el proyecto *InfoVersionService.Svc* como proyecto de inicio.
4. Compilar en modo *Release*.

A.2.2 Instalación como servicio de Windows

1. Abrir una consola PowerShell en modo administrador.
2. Ejecutar el siguiente comando:

```
sc create InfoVersionService binPath=
   "C:\Ruta\al\ejecutable\InfoVersionService.exe"
```

3. Para iniciar el servicio:

```
net start InfoVersionService
```

4. Para detenerlo:

```
net stop InfoVersionService
```

5. El servicio quedará registrado y podrá ser gestionado desde el panel de Servicios de Windows.

A.3 Configuración del servicio

El fichero de configuración del servicio *appsettings.json* se encuentra junto al ejecutable y contiene los siguientes parámetros:

- **ConnectionStrings**: cadena de conexión a la base de datos SQL.
- **ScheduledPackagesPath**: ruta principal donde ArchivosOBU genera los archivos por procesar.
- **InBusPackagesPath**: ruta principal donde TransferManager genera los archivos por procesar.
- **MinutesForCheckingVersions**: intervalo (en minutos) entre ejecuciones del análisis.
- **HoursForCleaningOldRecords**: intervalo de limpieza de registros antiguos.
- **MaxOldRecordDays**: número de días tras los cuales un paquete se considera eliminable.

Estos parámetros pueden modificarse sin necesidad de recompilar el servicio. Es necesario reiniciarlo para que los cambios tengan efecto.

A.3 Base de datos

Se proporciona un script SQL de inicialización con las siguientes tablas, las cuales siguen el mismo modelo descrito en la sección 8.4:

- *PackageVersion*: contiene versiones esperadas y reales por vehículo y tipo de contenido.
- *File*: contiene archivos concretos sobre cada paquete, aportando mayor granularidad.
- *PackageType*: Un enum con los distintos tipos de tablas a analizar.
- Índices recomendados sobre *VehicleId*, *PackageType* y *VersionId*.

El usuario configurado en la cadena de conexión debe tener permisos de lectura y escritura sobre estas tablas.

A.5 Despliegue del nuevo endpoint REST

1. Compilar el proyecto *SoaBasicContentManager*.
2. Añadir el nuevo controlador de versiones implementado.
3. Asegurarse de que el servicio expone el nuevo endpoint.
4. Desplegar el backend en *IIS*.

El endpoint accede directamente a la base de datos generada por InfoVersionService, por lo que no requiere lógica adicional para reconstruir el estado.

A.6 Activación del módulo visual en el SIU

- El componente visual se integra como un panel adicional dentro del SIU.
- Se añade al enrutado del frontend bajo la ruta */FleetStatus*.
- Para que aparezca en el menú, el operador debe tener asignado el permiso *FleetStatusManager* a nivel central de aplicaciones de GMV.

El módulo realiza peticiones HTTP al endpoint REST y renderiza dinámicamente el estado de sincronización por vehículo.

Apéndice B

Manual de Usuario

B.1 Acceso al módulo

El panel de trazabilidad de versiones se encuentra integrado en la interfaz del *SIU*. Para acceder, el usuario debe iniciar sesión con credenciales válidas y tener asignado el permiso específico *FleetStatusManager*. Además, solo podrá ver aquella información relacionada con el resto de permisos que tenga, por ejemplo, *LineManager*, *RouteManager*, etc. Es decir, no podrá ver información acerca de los paquetes de los que no tenga permiso.

Una vez autenticado, podrá elegir la flota sobra la que realizar las acciones.

Con la flota seleccionada se accederá al módulo desde el menú principal, bajo el apartado *"FleetStatus"* o similar, dependiendo del idioma de configuración del entorno.

B.2 Vista general

El panel principal muestra una tabla con todos los vehículos de la flota y su estado de sincronización de versiones. Por cada fila (vehículo), se indica:

- Código del vehículo.
- Estado general de sincronización.
- Indicadores por tipo de contenido (Multimedia, Configuración, Sistema, etc.).
- Versión conocida por el usuario para cada contenido
- Fecha de última actualización detectada.

Los estados de sincronización están representados mediante iconos de color:

- **Verde**: completamente sincronizado.
- **Rojo**: desincronizado o con errores.

Superpuestos a estos estados de sincronización se encuentran unas badges de distintos colores encargadas de mostrar lo siguiente:

- **All**: El paquete en cuestión forma parte de una configuración global para todas las flotas.

- **Fleet**: El paquete en cuestión forma parte de una configuración realizada para una la flota seleccionada en concreto.
- **Particular**: El paquete en cuestión forma parte de una configuración particular para una cantidad seleccionada de buses en concreto.

B.3 Filtros y búsquedas

El usuario puede refinar los resultados mediante los siguientes filtros:

- Por tipo de contenido (Multimedia, Configuración...).
- Por estado de sincronización.
- Por fecha de última actualización.
- Por identificador del vehículo.

Los filtros se pueden combinar y aplicar dinámicamente. Si no hay resultados para los filtros seleccionados, se muestra un mensaje informativo y una tabla vacía.

B.4 Consulta detallada por vehículo

Pasando el ratón por cualquier estado de sincronización en cualquier fila de la tabla, se muestra un tooltip del vehículo seleccionado. En ella se muestran:

- El tipo de configuración del paquete (All, Fleet, Particular).
- Las versiones esperadas de cada tipo de contenido.
- Las versiones actualmente detectadas en el OBU.

Esta vista permite realizar diagnósticos más precisos en caso de desincronización.

B.5 Visualización de KPIs

En la parte superior del panel se muestran indicadores globales de la flota:

- Porcentaje de vehículos totalmente sincronizados.
- Número total de vehículos desincronizados.
- Número total de vehículos sincronizados.

Estos KPIs se actualizan automáticamente cada vez que se aplican filtros o cambia la consulta.

B.6 Usabilidad

Este módulo está pensado para facilitar al operador técnico la comprobación del estado de actualización de los vehículos antes de su despliegue, especialmente tras una nueva campaña de contenidos. A continuación, se describe un ejemplo práctico de uso:

1. El operador accede al SIU e ingresa al panel de versionado desde el menú.
2. En el panel principal, consulta la tabla con el listado de vehículos y su estado de sincronización.
3. Aplica los filtros que considere para centrarse en los casos a analizar, como por ejemplo los buses a desplegar al día siguiente.
4. El operador analiza de forma centralizada el estado de los vehículos pendientes de despliegue, facilitando una decisión rápida sobre su aptitud para salir a ruta. En caso de detectar vehículos desincronizados, el operador puede consultar el *tooltip* con las versiones detalladas o examinar los paquetes concretos que presentan inconsistencias. Esta información le permite valorar si las desincronizaciones afectan a contenidos críticos o si, pese a ciertas discrepancias, el vehículo puede operar con normalidad.

Este flujo permite asegurar, de forma sencilla y visual, que todos los vehículos cumplen con las condiciones necesarias antes de salir a operación, reduciendo riesgos y evitando errores humanos en las comprobaciones manuales habituales.

Bibliografía

[1] GMV, *Innovating Solutions*, mar. de 2025. dirección: https://www.gmv.com/es-es (visitado 26-03-2025).

[2] K. Schwaber y J. Sutherland, *The Scrum Guide*. dirección: https://scrumguides.org/scrum-guide.html (visitado 31-03-2025).

[3] K. e. a. Beck, *Manifesto for Agile Software Development*. dirección: https://agilemanifesto.org (visitado 31-03-2025).

[4] Atlassian, *Los tres pilares del scrum: conoce los principios fundamentales del scrum*. dirección: https://www.atlassian.com/es/agile/project-management/3-pillars-scrum (visitado 07-04-2025).

[5] jblanco, *Scrum y Artefactos: Aumenta tu productividad y logra tus objetivos*. dirección: https://www.plainconcepts.com/es/scrum-que-es (visitado 08-04-2025).

[6] Talent.com, *Salario medio para Programador Junior*. dirección: https://es.talent.com/salary?job=programador+junior (visitado 09-04-2025).

[7] TECFYS, *Vida media de un ordenador*. dirección: https://tecfys.com/blog/post/23-conoces-la-vida-media-de-tu-ordenador (visitado 09-04-2025).

[8] Lenovo, *ThinkPad P14s Gen 4 (14Ïntel)*. dirección: https://www.lenovo.com/es/es/p/laptops/thinkpad/thinkpadp/thinkpad-p14s-gen-4-14-inch-intel/len101t0063?orgRef=https%253A%252F%252Fwww.google.com%252F&srsltid=AfmBOoqNGEnfxbq2UocC4wDy4a3cCd1qDAlS9PTyUVZF_DdvMdx2iEp2 (visitado 09-04-2025).

[9] Microsoft, *Microsoft 365 para empresas | Pequeas empresas | Microsoft 365*. dirección: https://www.microsoft.com/es-es/microsoft-365/business#layout-container-uid4d2d (visitado 09-04-2025).

[10] Microsoft, *Opciones de precios y compra | Visual Studio*. dirección: https://visualstudio.microsoft.com/es/vs/pricing/?tab=business (visitado 09-04-2025).

[11] Astah, *Pricing for Individual Licenses of Astah Software - Astah*. dirección: https://astah.net/pricing/individual (visitado 09-04-2025).

[12] Kev Zettler, *¿Qu es un sistema distribuido? | Atlassian*. dirección: https://www.atlassian.com/es/microservices/microservices-architecture/distributed-architecture (visitado 09-04-2025).

[13] José Manuel Ortega, *La arquitectura de los Sistemas de Transporte Inteligente ITS*. dirección: http://www.congresodevialidad.org.ar/congreso2014/conferencias/7-ITS-Ortega-Arquitectura-ITS.pdf (visitado 21-04-2025).

[14] Sara López Mora, *¿Qu son las Single-Page Application (SPA)? El desarrollo elegido por Gmail y LinkedIn*. dirección: https://digital55.com/blog/que-son-single-page-application-spa-desarrollo-elegido-por-gmail-linkedin (visitado 24-04-2025).

[15] *INIT*. dirección: https://www.initse.com/ende/home (visitado 24-04-2025).

[16] *Intelligent Transport Solutions*. dirección: https://www.trapezegroup.eu/intelligent-transport-systems (visitado 24-04-2025).

[17] *TransTrack Solutions Group | State of the Art Transit Software*. dirección: https://www.transtracksystems.net (visitado 24-04-2025).

[18] *Singleton*. dirección: https://refactoring.guru/es/design-patterns/singleton (visitado 30-06-2025).

[19] Colaboradores de los proyectos Wikimedia, *Facade*. dirección: https://es.wikipedia.org/w/index.php?title=Facade_(patr%C3%B3n_de_dise%C3%B1o)&oldid=160853394 (visitado 30-06-2025).

[20] C. G. Almirn, «Patrn de Inyeccin de dependencias - Adictos al trabajo,» dirección: https://adictosaltrabajo.com/2008/01/03/dependency-injector (visitado 30-06-2025).

[21] *Strategy*. dirección: https://refactoring.guru/es/design-patterns/strategy (visitado 30-06-2025).

[22] Contributors to Wikimedia projects, *Scheduled-task pattern - Wikipedia*. dirección: https://en.wikipedia.org/w/index.php?title=Scheduled-task_pattern&oldid=1023133553 (visitado 30-06-2025).

[23] *Template Method*. dirección: https://reactiveprogramming.io/blog/en/design-patterns/template-method (visitado 01-07-2025).