



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

(31105151) Trabajo de Fin de Máster

Ampliación del paradigma MDA mediante Inteligencia Artificial

Estudiante:

Felipe Hernández González

Tutor:

Ismael Abad Cardiel

Curso: 2025-2026

Convocatoria de septiembre de 2026

Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

(31105151) Trabajo de Fin de Máster

Ampliación del paradigma MDA mediante Inteligencia Artificial

Estudiante:

Felipe Hernández González

Tutor:

Ismael Abad Cardiel

Resumen

En los últimos años, los modelos de lenguaje de gran tamaño han incrementado considerablemente las posibilidades de generación automática de código. Sin embargo, la generación de aplicaciones mediante Inteligencia Artificial (IA) puede implicar la delegación de decisiones relacionadas con la arquitectura, el diseño y la organización del software, dificultando el control, la trazabilidad y el mantenimiento posterior del código generado.

Este Trabajo de Fin de Máster propone una metodología que combina el paradigma de *Model-Driven Architecture* (MDA) con las capacidades actuales de la generación de código mediante IA. El objetivo es mantener la Ingeniería del Software y la documentación como elementos centrales del proceso de desarrollo, delegando a la IA la implementación de la lógica de los métodos.

Para validar esta propuesta se ha desarrollado una aplicación capaz de interpretar diagramas de clases UML exportados en formato XML. A partir de la información estructural se genera, mediante un proceso determinista, el esqueleto del código fuente, incluyendo clases, atributos, métodos y relaciones. El modelo se complementa con documentación asociada a sus elementos, que se utiliza posteriormente como información semántica para guiar la generación mediante IA. Una vez construida esta especificación, la IA completa los cuerpos de los métodos respetando la estructura previamente definida.

De esta forma, se propone un proceso híbrido en el que las decisiones arquitectónicas y de diseño permanecen bajo el control del desarrollador, mientras que la Inteligencia Artificial se usa como herramienta de implementación. El enfoque busca combinar la capacidad de automatización de los modelos de lenguaje con las ventajas del desarrollo dirigido por modelos, favoreciendo la trazabilidad, la documentación y la mantenibilidad del software generado.

Palabras clave

Model-Driven Architecture (MDA), Inteligencia Artificial (IA), generación automática de código, modelos de lenguaje de gran tamaño (LLM), Unified Modeling Language (UML), XML Metadata Interchange (XMI), Ingeniería del Software.

Tabla de contenido

1.- Introducción	1
2.- Estado del arte	2
2.1.- Ingeniería dirigida por modelos	2
2.2.- Generación automática de código	3
2.2.1.- Aproximaciones tradicionales	4
2.2.2.- Generación de código dirigida por modelos	4
2.2.3.- Generación mediante Inteligencia Artificial	5
2.3.- Modelos de lenguaje y agentes	5
2.4.- UML, XMI y herramientas existentes	8
2.5.- Trabajos relacionados	9
2.6.- Identificación del problema	12
3.- Propuesta	14
3.1.- Planteamiento e hipótesis de trabajo	14
3.2.- Arquitectura conceptual	14
3.3.- Modelo UML enriquecido	16
3.4.- Decisiones de diseño	16
4.- Diseño de la aplicación	18
4.1.- Arquitectura software	18
4.2.- Casos de uso	20
4.3.- Modelo de dominio	23
4.4.- Diseño de funcionalidades	25
4.4.1.- Persistencia (DAO)	25
4.4.2.- Análisis sintáctico (ParserEntrada)	26
4.4.3.- Generación de esqueletos (GenerarSalida)	26
4.4.4.- Integración con Inteligencia Artificial (IntegracionIA)	27
4.4.5.- Utilidades	28
4.5.- Controlador e interfaz de usuario	28
5.- Implementación del prototipo	31
5.1.- Tecnologías y entorno de desarrollo	31
5.2.- Transformación del XMI al modelo interno	32
5.3.- Generación determinista del esqueleto	32
5.4.- Generación mediante Inteligencia Artificial	33
6.- Casos de estudio	34
6.1.- Caso sencillo	34
6.2.- Caso medio	35

6.3.- Caso completo.....	36
7.- Evaluación	38
7.1.- Metodología de evaluación.....	38
7.2.- Caracterización de los casos evaluados	39
7.3.- Evaluación de la generación determinista	41
7.4.- Evaluación de la generación mediante IA.....	43
7.5.- Comparativa y análisis de los resultados	45
7.6.- Limitaciones de la evaluación	47
8.- Discusión	49
8.1.- Cumplimiento de objetivos.....	49
8.2.- Validación de la hipótesis.....	50
8.3.- Comparación con otras aproximaciones de generación.....	50
8.4.- Papel de la documentación en la metodología.....	52
8.5.- Ventajas de la propuesta	52
8.6.- Inconvenientes y compromisos	53
9.- Conclusiones	55
9.1.- Conclusiones	55
9.2.- Contribuciones del trabajo.....	56
9.3.- Líneas futuras de trabajo	57
10.- Bibliografía	59
11.- Listado de siglas, abreviaturas y acrónimos.....	61
12.- Anexos.....	62
a.- Manual del usuario	62
b.- Manual técnico	66
c.- Formato XMI soportado	66
d.- Prompt utilizado.....	66
e.- Configuración de la IA	82
f.- Diagramas UML.....	82

Lista de imágenes

Imagen 1 Arquitectura conceptual.....	15
Imagen 2 Inicio herramienta	62
Imagen 3 Apertura fichero XMI UI	62
Imagen 4 Panel comentar clases UI	63
Imagen 5 Panel comentar métodos UI.....	63
Imagen 6 Panel comentar enumeración UI	64
Imagen 7 Panel salida código UI.....	64
Imagen 8 Opción clave OpenAI	65
Imagen 9 Panel clave OpenAI.....	65
Imagen 10 Opciones guardar / cargar proyecto UI.....	65
Imagen 11 Diagrama de clases simplificado	82
Imagen 12 Caso de uso 1.....	83
Imagen 13 Caso de uso 2.....	84
Imagen 14 Caso de uso 3.....	85
Imagen 15 Caso de uso 4.....	86
Imagen 16 Caso de uso 5.....	87
Imagen 17 Prueba simple.....	87
Imagen 18 Prueba media	88
Imagen 19 Prueba completa	88

Lista de tablas

Tabla 1 Caso de uso 1..... 20

Tabla 2 Caso de uso 2..... 21

Tabla 3 Caso de uso 3..... 21

Tabla 4 Caso de uso 4..... 22

Tabla 5 Caso de uso 5..... 22

Tabla 6 Progresión de pruebas realizadas..... 40

Tabla 7 Comparativa de pruebas 45

Tabla 8 Delegación de responsabilidades 51

Tabla 9 Listado de siglas..... 61

1.- Introducción

La generación automática de código constituye desde hace décadas uno de los objetivos de la Ingeniería del Software. Diferentes enfoques han intentado reducir el esfuerzo necesario para transformar diseños y especificaciones en implementaciones ejecutables. Entre ellos destaca *Model-Driven Architecture* (MDA), que sitúa los modelos como elementos centrales del proceso de desarrollo y permite generar determinados artefactos mediante transformaciones automáticas.

Aunque este enfoque favorece la abstracción, la trazabilidad y el control arquitectónico, presenta limitaciones cuando se pretende generar lógica de negocio específica mediante mecanismos deterministas. En paralelo, la evolución reciente de la Inteligencia Artificial y de los modelos de lenguaje de gran tamaño ha ampliado considerablemente las posibilidades de generación de código a partir de instrucciones expresadas en lenguaje natural.

Sin embargo, una mayor capacidad de generación también puede implicar una mayor delegación de decisiones relacionadas con la estructura, organización y diseño del software. Esta situación plantea un problema especialmente relevante desde la perspectiva del mantenimiento, ya que el desarrollador continúa siendo responsable de comprender, validar y evolucionar el código generado.

A partir de este contexto, el presente Trabajo de Fin de Máster propone estudiar una aproximación híbrida que combine los principios del desarrollo dirigido por modelos con las capacidades actuales de generación mediante Inteligencia Artificial. La propuesta parte de diagramas de clases UML enriquecidos con documentación, a partir de los cuales se genera de forma determinista el esqueleto del código. Posteriormente, la Inteligencia Artificial utiliza la información estructural y semántica definida en el modelo para implementar los cuerpos de los métodos.

El objetivo es analizar si ambas aproximaciones pueden combinarse de forma que las decisiones de arquitectura, estructura y documentación permanezcan bajo el control del desarrollador, mientras que la Inteligencia Artificial se utiliza como mecanismo de apoyo para automatizar la implementación de la lógica del software.

2.- Estado del arte

2.1.- Ingeniería dirigida por modelos

La creciente complejidad de los sistemas software y la necesidad de facilitar su mantenimiento, evolución y adaptación tecnológica han impulsado diferentes enfoques orientados a elevar el nivel de abstracción dentro del proceso de desarrollo. Entre ellos destaca la *Model-Driven Architecture* (MDA), una iniciativa promovida por el *Object Management Group* (OMG) a comienzos de la década de 2000 que propone utilizar los modelos como elementos centrales durante el diseño y construcción de sistemas software (Object Management Group, 2014).

MDA surge en un contexto caracterizado por el desarrollo de sistemas cada vez más distribuidos, heterogéneos y dependientes de tecnologías concretas. Tradicionalmente, una parte importante de las aplicaciones quedaba fuertemente vinculada a determinados lenguajes, plataformas o infraestructuras, provocando que los procesos de migración o evolución tecnológica requirieran importantes esfuerzos de adaptación o reescritura. Frente a esta situación, MDA plantea separar la definición funcional y arquitectónica del sistema de los detalles específicos asociados a su implementación.

Uno de los principios fundamentales de este enfoque consiste, por tanto, en aumentar el nivel de abstracción utilizado durante el desarrollo. Los modelos dejan de considerarse únicamente elementos de documentación y pasan a constituir artefactos formales capaces de participar activamente en procesos de transformación y generación automática. De esta manera, la información definida durante las fases de análisis y diseño puede mantenerse relacionada con los artefactos obtenidos durante la implementación, mejorando la trazabilidad entre los requisitos, la arquitectura y el código generado.

El desarrollo y estandarización de MDA está estrechamente relacionado con la actividad del OMG, organización responsable de distintos estándares utilizados en ingeniería del software. Entre ellos se encuentra UML, empleado para representar diferentes perspectivas estructurales y de comportamiento de los sistemas; *Meta Object Facility* (MOF), orientado a la definición de metamodelos; y XMI, utilizado para representar e intercambiar modelos mediante estructuras basadas en XML (Object Management Group, 2015), (Object Management Group, 2017), (Object Management Group, 2019). Estos estándares facilitan que los modelos puedan ser interpretados por distintas herramientas y utilizados como entrada de procesos automatizados de análisis, transformación o generación.

La arquitectura propuesta por MDA distingue diferentes niveles de abstracción. Habitualmente se identifican tres tipos principales de modelos:

- *Computation Independent Model* (CIM): representa el sistema desde una perspectiva orientada al dominio y al negocio, centrándose en sus requisitos, procesos y funcionamiento sin incorporar detalles técnicos.
- *Platform Independent Model* (PIM): introduce una descripción más próxima a la arquitectura software, definiendo componentes, relaciones y comportamiento lógico, pero manteniendo la independencia respecto a una tecnología concreta.
- *Platform Specific Model* (PSM): incorpora los aspectos específicos necesarios para materializar el sistema sobre una determinada plataforma, lenguaje *framework* o infraestructura.

El proceso planteado por MDA permite evolucionar progresivamente desde modelos de alto nivel hasta representaciones específicas de implementación. Para ello se utilizan

transformaciones entre modelos, mediante las cuales una representación puede convertirse en otra aplicando reglas previamente definidas. Así, un modelo independiente de plataforma puede transformarse en uno o varios modelos específicos para diferentes tecnologías. De forma similar, estas transformaciones pueden utilizarse para generar directamente artefactos software como clases, interfaces, estructuras de bases de datos, configuraciones o documentación técnica.

La generación automática de código constituye una de las aplicaciones más relevantes del paradigma. Al disponer de modelos estructurados y formalizados, determinadas partes de una aplicación pueden obtenerse automáticamente mediante transformaciones deterministas. Este enfoque permite reducir tareas repetitivas de implementación y favorecer la consistencia entre el modelo de diseño y el código resultante. Además, la posibilidad de reutilizar modelos y reglas de transformación permite adaptar una misma especificación a diferentes tecnologías, favoreciendo la portabilidad y reduciendo la dependencia de una plataforma concreta.

Entre las principales ventajas asociadas a MDA se encuentran la separación entre lógica funcional y tecnología, la reutilización de modelos, la mejora de la trazabilidad, la automatización parcial del proceso de desarrollo y la posibilidad de mantener una representación arquitectónica del sistema independiente de su implementación concreta. El uso de modelos facilita la comunicación entre distintos perfiles participantes en un proyecto, al proporcionar una representación común de la estructura y comportamiento del software.

Pese a todos sus puntos positivos, la aplicación práctica de MDA también presenta importantes limitaciones. El uso de distintos niveles de abstracción, metamodelos y mecanismos de transformación puede incrementar la complejidad del proceso de desarrollo y exige conocimientos especializados. Muchas de sus capacidades dependen de herramientas específicas y de transformaciones previamente definidas, lo que puede limitar la flexibilidad cuando aparecen requisitos no contemplados inicialmente. La generación automática obtenida mediante estas herramientas se ha centrado en elementos estructurales y repetitivos, mientras que la implementación de lógica de negocio compleja continúa requiriendo intervención manual.

Estas limitaciones han provocado que existan diferencias significativas entre las capacidades teóricas de MDA y su aplicación práctica. Aun así, sus principios fundamentales (abstracción, separación entre diseño e implementación, uso de modelos como artefactos activos, trazabilidad y generación automática) continúan siendo relevantes dentro de la Ingeniería del Software. La aparición de nuevas técnicas de generación de código basadas en modelos de lenguaje permite reconsiderar algunas de estas limitaciones, especialmente aquellas relacionadas con la necesidad de implementar manualmente las partes del sistema que no pueden obtenerse mediante transformaciones deterministas.

2.2.- Generación automática de código

La generación automática de código constituye una línea de trabajo consolidada dentro de la Ingeniería del Software cuyo objetivo es reducir la intervención manual necesaria para transformar una especificación de un sistema en código fuente. Aunque en los últimos años este concepto se ha asociado especialmente con la Inteligencia Artificial, los intentos de automatizar la implementación de software son anteriores a la aparición de los actuales modelos de lenguaje. A lo largo de las últimas décadas pueden distinguirse diferentes etapas caracterizadas por el tipo de información usada como entrada y por el nivel de automatización alcanzado.

2.2.1.- Aproximaciones tradicionales

Las primeras aproximaciones a la generación automática de código se apoyaron principalmente en reglas de transformación, especificaciones formales, representaciones conceptuales y lenguajes controlados del sistema. Dentro de este contexto adquirieron especial importancia las herramientas CASE (*Computer-Aided Software Engineering*), destinadas a ayudar a analistas y desarrolladores durante diferentes etapas del ciclo de vida del software.

(Chaverra, 2010) realizan una revisión de diferentes aproximaciones existentes a la generación automática y semiautomática de código. Los autores identifican propuestas que parten de especificaciones formales, descripciones realizadas mediante lenguaje natural o controlado y de esquemas conceptuales. Estas aproximaciones perseguían transformar progresivamente una representación de alto nivel en estructuras cada vez más próximas al código fuente.

El grado de automatización alcanzado por estas herramientas era variable. Algunas soluciones permitían generar únicamente esquemas conceptuales o la estructura básica de las aplicaciones, mientras que otras incorporaban la generación de interfaces gráficas o determinados cuerpos de métodos. Sin embargo, la generación del comportamiento se encontraba limitada a procesos previamente definidos y conocidos, como determinadas operaciones de gestión de usuarios. Por tanto, una parte importante de la lógica específica del sistema continuaba requiriendo la intervención del desarrollador.

Esta etapa muestra una de las principales dificultades históricas de la generación de software, transformar una especificación abstracta en una implementación concreta requiere de reglas previas capaces de interpretar dicha especificación. Los autores señalan, además, que muchos de estos sistemas parten de representaciones de la solución en lugar de partir del dominio del problema y que continúa siendo necesaria una participación importante del analista para adoptar decisiones que no pueden automatizarse fácilmente.

2.2.2.- Generación de código dirigida por modelos

Posteriormente, los enfoques dirigidos por modelos reforzaron el uso de representaciones formales como punto de partida para la generación automática. En paradigmas como MDA, descrito en el apartado anterior, los modelos dejan de actuar únicamente como documentación y pasan a constituir artefactos capaces de ser procesados mediante transformaciones.

Esta aproximación permite definir reglas que interpreten modelos estructurados y produzcan automáticamente elementos de implementación. A partir de modelos UML, por ejemplo, pueden generarse clases, atributos, declaraciones de métodos, relaciones entre elementos, estructuras de persistencia, configuraciones u otros artefactos software. Frente a la programación completamente manual, este mecanismo permite automatizar aquellas partes de la implementación cuya transformación puede definirse de forma determinista.

La generación basada en modelos ha continuado siendo objeto de investigación durante las últimas décadas. (Gabriel Sebastián, 2020) realizaron un estudio sistemático de la literatura relacionada con generación de código mediante MDA correspondiente al periodo comprendido entre 2008 y 2018. El estudio identificó inicialmente 2.145 publicaciones relacionadas con MDA, de las que finalmente fueron seleccionados 50 trabajos primarios relevantes para el análisis específico. Los autores concluyen que continúa existiendo actividad investigadora relacionada con este paradigma y que diferentes trabajos demuestran la posibilidad de aplicar técnicas MDA para mejorar determinados procesos de desarrollo software.

Sin embargo, su uso en entornos laborales no ha alcanzado la extensión inicialmente prevista. Entre las posibles causas se encuentran la necesidad de disponer de profesionales especializados y las limitaciones de las herramientas necesarias para aplicar estas técnicas en entornos de producción (Gabriel Sebastián, 2020). Además, la capacidad de generación continúa dependiendo de transformaciones, plantillas y reglas desarrolladas previamente. Esto proporciona un comportamiento predecible y controlable, pero limita la capacidad de generar aquellos comportamientos que no hayan sido contemplados dentro de las transformaciones disponibles.

Como consecuencia, las técnicas tradicionales de generación automática han resultado especialmente adecuadas para producir estructuras repetitivas o formalizables, mientras que la implementación de lógica específica y comportamientos más complejos ha continuado dependiendo en gran medida del trabajo del desarrollador.

2.2.3.- Generación mediante Inteligencia Artificial

La aparición de modelos de lenguaje capaces de trabajar con código ha supuesto un cambio significativo respecto a las aproximaciones tradicionales de generación automática. En lugar de depender exclusivamente de reglas, plantillas o transformaciones definidas previamente, estos modelos pueden interpretar especificaciones expresadas mediante lenguaje natural y producir implementaciones adaptadas al contexto proporcionado.

Esta capacidad permite abordar comportamientos cuya traducción a código resultaría costosa de expresar mediante reglas deterministas específicas. La generación deja así de limitarse a aquellas estructuras cuya transformación ha sido formalizada previamente y puede incorporar información semántica menos estructurada, como descripciones funcionales, documentación o fragmentos de código existentes.

La evolución posterior de estos modelos ha dado lugar además a sistemas capaces de interactuar con herramientas y entornos de desarrollo, ampliando la automatización desde la generación de fragmentos de código hacia tareas más amplias de Ingeniería del Software. Debido a la relevancia de esta evolución para el presente trabajo, los modelos de lenguaje y los agentes basados en ellos se analizan específicamente en el apartado siguiente.

2.3.- Modelos de lenguaje y agentes

La evolución reciente de la Inteligencia Artificial está estrechamente relacionada con el desarrollo de los modelos de lenguaje de gran tamaño o *Large Language Models* (LLM). Estos modelos están diseñados para procesar grandes cantidades de información y generar nuevas secuencias a partir del contexto recibido. Aunque inicialmente su aplicación estuvo principalmente relacionada con el procesamiento y generación de lenguaje natural, progresivamente han demostrado capacidad para trabajar también con información estructurada y lenguajes de programación.

Uno de los principales avances que permitió el desarrollo de los modelos actuales fue la aparición de la arquitectura *Transformer*. Esta arquitectura introdujo mecanismos de atención capaces de establecer relaciones entre los distintos elementos que forman una secuencia, facilitando el procesamiento de grandes cantidades de información y permitiendo posteriormente desarrollar modelos con un número cada vez mayor de parámetros y capacidades (Vaswani, y otros, 2017).

La evolución de estas arquitecturas permitió entrenar modelos sobre grandes conjuntos de información procedentes de diferentes ámbitos. Como consecuencia, comenzaron a aparecer

sistemas capaces de utilizar las instrucciones y el contexto proporcionado durante una petición para adaptarse a tareas diferentes sin necesidad de disponer de una solución específica desarrollada previamente para cada una de ellas.

Esta característica resulta especialmente relevante para la generación automática de código. Los lenguajes de programación presentan una sintaxis formal y unas reglas concretas, pero el código fuente contiene también información semántica mediante nombres de clases, métodos, atributos, comentarios y documentación. Los modelos de lenguaje pueden utilizar conjuntamente estos elementos para interpretar una determinada intención y generar una posible implementación.

Los primeros modelos orientados específicamente al código mostraron la posibilidad de generar funciones completas a partir de una firma y una descripción de su comportamiento. A partir de estas aproximaciones, las capacidades se han ampliado progresivamente hacia tareas como completar fragmentos existentes, generar pruebas, documentar código, detectar errores, realizar correcciones o interpretar proyectos con un contexto mayor (Wang, Wang, Joty, & Hoi, 2021), (Hou, y otros, 2024), (Jiang, Wang, Shen, Kim, & Kim, 2026), (Zhang, y otros, 2026).

La principal diferencia respecto a los mecanismos tradicionales de generación se encuentra en que no resulta necesario definir previamente una transformación concreta para cada comportamiento que se desea implementar. Mientras que una transformación determinista necesita reglas capaces de convertir una entrada conocida en una salida determinada, un modelo de lenguaje puede interpretar una descripción expresada mediante lenguaje natural y producir código adaptado al contexto recibido.

Esto proporciona una mayor flexibilidad, especialmente cuando se pretende generar lógica de negocio específica. Sin embargo, también modifica las características del proceso. La respuesta generada no es necesariamente reproducible de forma exacta entre distintas ejecuciones y su corrección depende tanto de las capacidades del modelo como de la calidad y cantidad de información proporcionada durante la solicitud.

El contexto adquiere, por tanto, una importancia considerable. Una descripción aislada de un método puede no resultar suficiente cuando su comportamiento depende de clases, atributos o relaciones existentes en otras partes del proyecto. Para producir una implementación coherente, el modelo necesita disponer de información suficiente para identificar los elementos que puede utilizar y las restricciones que debe respetar, siendo la construcción adecuada de las instrucciones y del contexto uno de los aspectos relevantes en la interacción con estos modelos (Sahoo, y otros, 2024).

Esta necesidad de proporcionar contexto ha impulsado progresivamente la integración de los modelos de lenguaje con herramientas de desarrollo y con mecanismos capaces de proporcionarles información adicional sobre los proyectos. De esta evolución surgen los denominados agentes basados en modelos de lenguaje, que amplían el funcionamiento de una petición convencional incorporando un mayor grado de autonomía.

En una utilización directa de un LLM, el usuario proporciona una entrada y recibe una respuesta que posteriormente debe analizar e integrar. En cambio, un agente puede participar en un proceso formado por múltiples pasos. El sistema puede analizar el estado actual de una tarea, decidir una acción, utilizar una herramienta para ejecutarla, observar el resultado obtenido y utilizar esta nueva información para determinar las siguientes operaciones.

Aplicado a la Ingeniería del Software, este planteamiento permite que la Inteligencia Artificial intervenga sobre tareas que requieren trabajar con diferentes elementos de un proyecto. Un agente puede, por ejemplo, explorar la estructura de un repositorio, consultar archivos, modificar código, ejecutar una aplicación o sus pruebas y utilizar los errores obtenidos para realizar posteriormente nuevas modificaciones, capacidades que han sido estudiadas tanto mediante entornos de evaluación sobre repositorios reales como mediante propuestas específicamente orientadas a agentes de desarrollo software (Jimenez, y otros, 2024), (John Yang, 2024) (He, Treude, & Lo, 2025), (Wang, y otros, 2025).

De esta forma, la automatización deja de centrarse únicamente en la generación de un fragmento de código y puede abarcar una parte más amplia del proceso necesario para resolver un problema. El desarrollador puede proporcionar un objetivo de mayor nivel y delegar en el agente parte de las decisiones necesarias para alcanzarlo.

Esta capacidad representa una evolución importante dentro de la generación automática de software. Mientras que las primeras técnicas dependían fundamentalmente de reglas y transformaciones previamente definidas, los modelos de lenguaje introducen la posibilidad de interpretar información semántica y generar implementaciones más variables. Los agentes amplían posteriormente esta capacidad permitiendo que el propio sistema interactúe con el entorno y tome decisiones durante el proceso.

Sin embargo, el aumento de autonomía introduce también diferentes problemas. El primero se encuentra relacionado con el carácter no determinista de los modelos. Una misma petición puede producir resultados diferentes y no existe una garantía absoluta de que el código obtenido implemente correctamente el comportamiento esperado, aunque resulte sintácticamente correcto.

También pueden producirse errores derivados de una interpretación incorrecta de la información proporcionada. Cuando el contexto resulta incompleto, ambiguo o contiene información contradictoria, el modelo puede generar elementos que no existen, utilizar incorrectamente determinadas dependencias o asumir comportamientos que no han sido definidos por el desarrollador. Estas situaciones hacen necesario revisar y validar el resultado antes de incorporarlo definitivamente a una aplicación.

La calidad del código generado constituye igualmente un aspecto relevante. La generación automática no garantiza que la implementación obtenida sea necesariamente la más adecuada desde el punto de vista de rendimiento, seguridad, mantenibilidad o cumplimiento de las reglas de negocio, aspectos relacionados con la calidad del producto software contemplados en estándares como ISO/IEC 25010:2023 y analizados también en estudios recientes sobre los riesgos y retos asociados al código generado mediante IA (International Organization for Standardization; International Electrotechnical Commission, 2023), (Perry, Srivastava, Kumar, & Boneh, 2023), (Gao, Hu, Gao, Xia, & Jin, 2025). Por ello, la utilización de estos sistemas no elimina la responsabilidad del desarrollador sobre el código resultante.

En los agentes, estos problemas pueden adquirir una mayor importancia debido a su capacidad para ejecutar diferentes acciones de manera encadenada. Una decisión incorrecta puede modificar el estado del proyecto y afectar posteriormente a las siguientes decisiones tomadas por el sistema. Cuanto mayor sea la autonomía concedida al agente, mayor importancia adquieren los mecanismos destinados a limitar las operaciones permitidas y comprobar los resultados obtenidos.

Existe, por tanto, un equilibrio entre el grado de automatización y el control mantenido sobre el proceso. Los agentes permiten delegar un mayor número de tareas y pueden reducir la intervención necesaria para resolver determinados problemas, pero también pueden asumir decisiones relacionadas con la estructura, organización o modificación del software.

Esta situación resulta especialmente importante desde la perspectiva de la Ingeniería del Software. El código generado debe continuar siendo comprendido, mantenido y evolucionado posteriormente, independientemente de que su implementación haya sido realizada de forma manual o automática. Delegar decisiones estructurales puede aumentar la velocidad inicial de generación, pero también puede dificultar posteriormente la identificación del motivo por el que se ha adoptado una determinada organización del sistema.

El planteamiento utilizado en este trabajo limita deliberadamente el ámbito de actuación de la Inteligencia Artificial. El modelo de lenguaje no decide qué clases deben existir, cómo se organizan los espacios de nombres, qué relaciones se establecen entre los elementos o cuáles deben ser las firmas de los métodos. Estas decisiones se encuentran definidas previamente mediante el modelo UML y se trasladan al código mediante un proceso determinista.

La Inteligencia Artificial se utiliza principalmente para interpretar la información semántica proporcionada mediante la documentación e implementar los cuerpos de los métodos sobre una estructura ya establecida. De esta manera se pretende aprovechar la flexibilidad proporcionada por los modelos de lenguaje sin alcanzar el grado de autonomía existente en aproximaciones basadas en agentes capaces de modificar directamente el proyecto.

La propuesta se sitúa así entre dos aproximaciones diferentes. Por una parte, mantiene mecanismos propios de la generación dirigida por modelos para aquellos elementos que pueden definirse de forma estructurada y transformarse mediante reglas conocidas. Por otra, incorpora la capacidad de los modelos de lenguaje para interpretar descripciones funcionales y generar lógica específica. La combinación de ambas técnicas constituye la base sobre la que se desarrolla la metodología propuesta en los capítulos posteriores.

2.4.- UML, XMI y herramientas existentes

Unified Modeling Language (UML) es un lenguaje de modelado estandarizado por *Object Management Group* (OMG) utilizado para representar diferentes perspectivas de los sistemas software. UML incluye distintos tipos de diagramas destinados a describir aspectos estructurales y de comportamiento. Dentro del contexto de este trabajo adquieren especial relevancia los diagramas de clases, ya que permiten representar elementos directamente relacionados con el código orientado a objetos, como clases, interfaces, atributos, métodos y relaciones entre clases.

Esta correspondencia entre los elementos del modelo y las construcciones habituales de los lenguajes de programación facilita el uso de los diagramas como punto de partida para procesos de generación automática. Sin embargo, para que esta información pueda ser interpretada por una aplicación es necesario disponer de una representación estructurada del modelo, más allá de su representación gráfica.

Para facilitar el intercambio de esta información, OMG define *XML Metadata Interchange* (XMI), un estándar basado en XML destinado a la representación e intercambio de modelos. Mediante XMI es posible almacenar información sobre los distintos elementos definidos dentro de un modelo UML, incluyendo identificadores, clases, atributos, operaciones y relaciones. De esta forma, el modelo puede ser procesado automáticamente por herramientas diferentes.

Aunque XMI pretende favorecer la interoperabilidad, en la práctica pueden existir diferencias entre las versiones soportadas y las extensiones incorporadas por cada herramienta de modelado. Esto provoca que la compatibilidad entre aplicaciones no sea siempre completa y hace necesario conocer la estructura concreta de los documentos XMI que se utilizan como entrada.

Existen numerosas herramientas destinadas a la creación y gestión de modelos UML. Modelio es una herramienta de modelado que proporciona soporte para UML y BPMN y permite importar y exportar modelos mediante XMI. Su orientación principal se encuentra en el diseño y documentación de sistemas, aunque puede ampliarse mediante módulos y mecanismos adicionales de generación. Estas características permiten utilizar los diagramas creados por el desarrollador como origen de información estructurada para otros procesos automatizados.

En conjunto, UML proporciona el mecanismo de representación del sistema, XMI permite intercambiar y procesar automáticamente la información contenida en los modelos y las herramientas de modelado proporcionan el entorno necesario para su creación y gestión. Estos elementos constituyen la infraestructura sobre la que pueden construirse posteriormente procesos de transformación y generación automática de software.

2.5.- Trabajos relacionados

La combinación de técnicas de automatización e Inteligencia Artificial con enfoques dirigidos por modelos ha sido objeto de investigación desde antes de la aparición de los actuales modelos de lenguaje y continúa siendo estudiada específicamente con la incorporación de LLM dentro de la Ingeniería Dirigida por Modelos (Di Rocco, Di Ruscio, Di Sipio, Nguyen, & Rubei, 2025). Aunque las tecnologías utilizadas y el grado de automatización han evolucionado considerablemente, pueden encontrarse diferentes trabajos que persiguen objetivos relacionados con los planteados en este TFM: aumentar la cantidad de software generado automáticamente, facilitar la implementación del comportamiento y mantener la información procedente de las fases de modelado e Ingeniería del Software.

Uno de los antecedentes encontrados es el trabajo de (Cáceres, 2011), *Generación automática de aplicaciones software a partir del estándar MDA basándose en la metodología de sistemas expertos e Inteligencia Artificial*. El autor parte de las limitaciones de los mecanismos tradicionales de generación asociados a MDA y propone complementar el paradigma mediante técnicas de Inteligencia Artificial y sistemas expertos. Una de las principales dificultades identificadas se encuentra en la automatización de las reglas de negocio, ya que los generadores disponibles permitían obtener determinados elementos estructurales o repetitivos, mientras que la lógica específica de las aplicaciones continuaba presentando mayores dificultades.

Para abordar este problema, Cáceres plantea incorporar las reglas de negocio al proceso de transformación y utilizar conceptos procedentes de los sistemas expertos y de la Inteligencia Artificial. La propuesta contempla además la utilización de agentes como apoyo al proceso, aunque se trata de aproximaciones propias de la tecnología disponible en ese momento y no de los actuales agentes basados en modelos de lenguaje. El interés de este trabajo para el presente TFM reside principalmente en mostrar que la posibilidad de complementar MDA mediante Inteligencia Artificial para ampliar la generación del comportamiento ya había sido planteada antes de la aparición de la IA generativa.

Posteriormente, (Mercedes Picón, 2017), en *Hacia un Enfoque para la Generación de Código con MDA*, analizan diferentes mecanismos de generación automática a partir de modelos UML. Su

trabajo estudia aspectos como los diagramas soportados, las transformaciones entre modelos, la generación para distintos lenguajes, la ingeniería directa e inversa y la integración de los generadores con otras herramientas de desarrollo.

Los autores muestran que, aunque las herramientas basadas en modelos pueden automatizar una parte importante de la construcción de una aplicación, la generación del comportamiento continúa siendo uno de los aspectos más difíciles de abordar. En determinados escenarios, el resultado obtenido a partir del modelo se limita a estructuras formadas por clases, atributos y declaraciones de métodos que posteriormente deben ser completadas manualmente por el desarrollador.

Picón y Torrealba destacan además que el nivel de detalle del modelo condiciona directamente las posibilidades de generación. Una representación estructural puede ser suficiente para generar determinadas construcciones de código, pero no necesariamente contiene toda la información requerida para determinar el comportamiento de las operaciones. Incrementar el grado de automatización exige, por tanto, aumentar la información proporcionada mediante modelos adicionales, lenguajes específicos o mecanismos complementarios.

La evolución reciente de la Inteligencia Artificial ha permitido abordar este problema desde una perspectiva diferente. (Zina Zammel, 2025), en *Towards AI-Enabled Model-Driven Architecture: Systematic Literature Review*, realizan una revisión sistemática de trabajos publicados entre 2018 y 2024 relacionados con la utilización de técnicas de Inteligencia Artificial dentro de MDA. Los autores analizan la aplicación de estas técnicas sobre distintas fases del paradigma, incluyendo la interpretación de requisitos, la generación de modelos, las transformaciones entre diferentes niveles de abstracción y la generación de código.

La revisión muestra que la relación entre ambas áreas puede producirse en diferentes sentidos. La Inteligencia Artificial puede utilizarse para automatizar o asistir actividades tradicionalmente complejas dentro del desarrollo dirigido por modelos, mientras que los propios modelos proporcionan información estructurada que puede servir como contexto para los sistemas de IA. La aparición de los modelos de lenguaje amplía especialmente esta posibilidad al permitir interpretar información expresada mediante lenguaje natural y generar implementaciones adaptadas a dicha información.

Sin embargo, Zammel et al. señalan también diferentes limitaciones de las aproximaciones actuales, como la necesidad de realizar evaluaciones más amplias, las dificultades de generalización entre herramientas y formatos de modelado y los posibles errores producidos por los sistemas generativos. Estos resultados muestran que la incorporación de Inteligencia Artificial a procesos dirigidos por modelos continúa siendo un ámbito de investigación abierto.

Dentro de las propuestas recientes, (Buchmann, Peinl, & Schwägerl, White-box LLM-supported Low-code Engineering: A Vision and First Insights, 2024). El trabajo parte del problema de utilizar modelos de lenguaje para generar aplicaciones completas a partir de instrucciones de alto nivel, donde parte de las decisiones adoptadas durante la generación pueden quedar ocultas dentro de un proceso no determinista.

Como alternativa, los autores proponen un enfoque en el que el usuario define modelos de dominio semiestructurados y complementa determinados elementos mediante anotaciones y restricciones expresadas en lenguaje natural. A partir de esta información se obtiene un esqueleto de aplicación que se proporciona al modelo de lenguaje junto con las descripciones

correspondientes, permitiendo que la IA complete aquellas partes cuyo comportamiento se encuentra expresado mediante información semántica.

Esta propuesta presenta una relación especialmente próxima con el planteamiento desarrollado en este TFM, ya que combina información estructurada, un esqueleto generado previamente y descripciones en lenguaje natural utilizadas por un modelo de lenguaje. No obstante, su orientación se encuentra principalmente relacionada con entornos *low-code* y con modelos de dominio semiformales, mientras que el presente trabajo utiliza diagramas de clases UML intercambiados mediante XMI y establece de forma explícita qué elementos deben obtenerse de manera determinista y cuáles pueden ser implementados posteriormente mediante Inteligencia Artificial.

Los mismos autores desarrollan posteriormente esta línea mediante MoProCo, presentado por (Buchmann, Schwägerl, & Peinl, To Model, to Prompt, or to Code? The Choice Is Yours: A Multi-Paradigmatic Approach to Software Development, 2025). La propuesta combina dentro de un mismo proceso tres mecanismos diferentes para desarrollar software: modelado, instrucciones expresadas mediante lenguaje natural y programación convencional.

El desarrollador puede dividir una funcionalidad y decidir qué partes se representan mediante modelos, cuáles se describen mediante *prompts* y cuáles se implementan directamente mediante código. A partir de esta información, la herramienta genera posteriormente una aplicación Java mediante un proceso dividido en diferentes fases. El planteamiento busca aprovechar las ventajas del carácter determinista de los modelos y la mayor flexibilidad proporcionada por los modelos de lenguaje.

MoProCo resulta especialmente relevante para este trabajo porque representa una aproximación híbrida en la que las técnicas tradicionales de modelado y la generación mediante Inteligencia Artificial no se consideran mecanismos excluyentes. Sin embargo, la distribución de responsabilidades es diferente. En MoProCo el desarrollador puede combinar modelado, prompting y programación según las necesidades de cada parte del sistema, mientras que la metodología desarrollada en este TFM establece previamente una frontera más concreta entre ambos mecanismos.

Los trabajos analizados muestran, por tanto, una evolución progresiva en la relación entre Ingeniería del Software, generación automática e Inteligencia Artificial. Las primeras aproximaciones ya identificaban la lógica de negocio como una de las principales dificultades de la generación basada en modelos y planteaban utilizar técnicas de Inteligencia Artificial para complementar las transformaciones tradicionales. Posteriormente, los modelos de lenguaje han permitido incorporar información semántica expresada mediante lenguaje natural y generar directamente implementaciones adaptadas al contexto proporcionado.

Las propuestas más recientes muestran además un interés creciente por combinar ambas técnicas en lugar de sustituir completamente los mecanismos dirigidos por modelos. El uso conjunto de información estructurada y modelos de lenguaje permite mantener determinadas decisiones dentro de procesos controlados y utilizar la generación no determinista únicamente en aquellas partes donde puede aportar una mayor flexibilidad.

El presente trabajo se sitúa dentro de esta línea de investigación y adopta una materialización concreta de dicha combinación. El desarrollador define inicialmente un modelo UML enriquecido con documentación, que establece la estructura del sistema y las principales decisiones de diseño. El fichero XMI se transforma en una representación interna y, a partir de

ella, se genera de forma determinista el esqueleto C#, incluyendo clases, atributos, métodos, relaciones, tipos y documentación.

La Inteligencia Artificial interviene posteriormente sobre esta estructura. Para cada generación recibe el esqueleto correspondiente a una clase concreta y una representación estructurada del proyecto completo que proporciona contexto sobre los restantes elementos del sistema. Los cuerpos generados se relacionan con los métodos mediante identificadores previamente incorporados durante la fase determinista, permitiendo integrar posteriormente la respuesta sin delegar en el modelo la reconstrucción de la estructura de la clase.

Por tanto, la aportación del presente trabajo no se encuentra simplemente en combinar MDA e Inteligencia Artificial, posibilidad que aparece ya en diferentes investigaciones, sino en estudiar una metodología en la que ambos mecanismos mantienen responsabilidades claramente diferenciadas. La generación determinista conserva la autoridad sobre la estructura definida por el desarrollador, mientras que la Inteligencia Artificial utiliza la documentación y el contexto global del proyecto para implementar la lógica dentro de unos puntos previamente delimitados.

2.6.- Identificación del problema

El análisis realizado en los apartados anteriores permite identificar dos aproximaciones diferentes a la automatización del desarrollo software que presentan ventajas y limitaciones complementarias. Por una parte, los enfoques dirigidos por modelos proporcionan mecanismos formales para representar la arquitectura y estructura de una aplicación y permiten generar determinados artefactos mediante transformaciones controladas. Por otra, los actuales modelos de Inteligencia Artificial presentan una capacidad mayor para generar código a partir de información semántica expresada mediante lenguaje natural.

En el caso de MDA, una de sus principales ventajas reside en situar los modelos, la arquitectura y la documentación como elementos centrales del proceso de desarrollo. Las decisiones estructurales se establecen antes de la implementación y pueden mantenerse vinculadas con los artefactos generados, favoreciendo la trazabilidad y facilitando la comprensión global del sistema. Además, cuando la transformación entre modelo y código puede definirse mediante reglas deterministas, resulta posible automatizar la creación de clases, atributos, relaciones, interfaces u otras estructuras con un comportamiento predecible.

Sin embargo, esta misma naturaleza determinista supone una limitación cuando se pretende generar comportamiento específico. Para producir automáticamente la lógica de negocio mediante mecanismos tradicionales es necesario definir previamente las reglas capaces de transformar cada especificación en una implementación. Aunque este enfoque resulta adecuado para estructuras repetitivas o comportamientos formalizados, su aplicación se vuelve más compleja a medida que aumenta la complejidad de la lógica del negocio. Como se ha observado en los trabajos relacionados, la generación del comportamiento ha constituido históricamente una de las principales dificultades de los enfoques tradicionales basados en modelos.

Los modelos de lenguaje actuales modifican esta situación. La Inteligencia Artificial puede interpretar descripciones funcionales expresadas en lenguaje natural y generar código sin necesidad de disponer de una transformación específica previa para cada comportamiento. Esto permite automatizar partes de la lógica que difícilmente podrían obtenerse mediante reglas deterministas. Además, las aproximaciones basadas en agentes amplían esta capacidad al

permitir que el sistema inspeccione proyectos, modifique artefactos y compruebe los resultados obtenidos.

No obstante, el aumento de la capacidad de generación puede implicar una reducción del control explícito ejercido por el desarrollador sobre el proceso. Cuando el modelo participa no solo en la implementación, sino también en decisiones relacionadas con la creación de clases, modificación de estructuras, selección de patrones o reorganización del proyecto, una parte de las decisiones de Ingeniería del Software puede quedar implícita dentro del propio proceso de generación. Como consecuencia, puede resultar más difícil establecer la correspondencia entre las decisiones de diseño originales y el código producido.

Esta situación adquiere especial relevancia desde la perspectiva del mantenimiento. La generación de código no constituye únicamente un problema de productividad durante la implementación inicial, el software generado debe ser comprendido, validado, corregido y evolucionado posteriormente. El desarrollador continúa siendo responsable del comportamiento del sistema independientemente de que parte de su código haya sido escrita mediante una herramienta automática. Por ello, aumentar la cantidad de código generado sin disponer de mecanismos suficientes para conocer su origen, estructura y relación con el diseño puede trasladar parte del esfuerzo ahorrado durante la implementación hacia las tareas posteriores.

A partir de estas observaciones puede identificarse una tensión entre automatización y control. Los mecanismos tradicionales de MDA proporcionan un elevado control sobre la estructura del sistema, pero presentan mayores dificultades para automatizar lógica específica. Los modelos de Inteligencia Artificial, en cambio, permiten generar lógica más flexible, aunque una utilización con un alto grado de autonomía puede reducir la trazabilidad respecto a las decisiones de ingeniería.

Las características de ambas aproximaciones plantean la posibilidad de estudiar un enfoque complementario. La generación determinista puede reservarse para aquellos elementos definidos formalmente mediante el modelo, manteniendo bajo control del desarrollador las decisiones relacionadas con arquitectura, estructura y organización del sistema. La Inteligencia Artificial puede utilizarse en aquellas tareas donde las transformaciones tradicionales presentan mayores limitaciones, principalmente en la implementación de comportamiento a partir de especificaciones semánticas.

Por tanto, el problema abordado en este trabajo no se limita a maximizar la cantidad de código que puede generarse automáticamente, sino a estudiar cómo aumentar dicha automatización manteniendo al mismo tiempo la trazabilidad, documentación y control sobre la estructura del software resultante. El objetivo consiste en analizar si es posible aprovechar la capacidad generativa de los modelos actuales sin relegar las decisiones fundamentales de Ingeniería del Software a la propia herramienta encargada de implementar el código.

3.- Propuesta

3.1.- Planteamiento e hipótesis de trabajo

A partir del problema identificado en el apartado anterior, se propone una metodología de generación de software que combine mecanismos deterministas basados en modelos con las capacidades actuales de generación mediante Inteligencia Artificial. La intención es aprovechar las ventajas de ambos enfoques y establecer una separación clara entre las decisiones relacionadas con la Ingeniería del Software y aquellas tareas asociadas principalmente a la implementación.

La propuesta parte de un diagrama de clases UML definido previamente por el desarrollador. Este modelo establece la estructura del sistema mediante clases, interfaces, atributos, métodos y relaciones, constituyendo la base sobre la que se realizará posteriormente la generación. La información estructural se complementa con documentación asociada a los diferentes elementos del modelo, permitiendo describir sus responsabilidades y, especialmente, el comportamiento esperado de los métodos.

A partir de esta información se genera de forma determinista el esqueleto del código fuente. De este modo, la estructura de las clases y las decisiones arquitectónicas no dependen de la Inteligencia Artificial, sino directamente del modelo definido por el desarrollador. La documentación incorporada durante el modelado se traslada también al código generado mediante comentarios estructurados, de manera que pueda utilizarse tanto para facilitar su comprensión como para producir posteriormente documentación técnica mediante herramientas compatibles.

La Inteligencia Artificial interviene una vez establecida esta estructura. El esqueleto generado y la información semántica proporcionada mediante la documentación se utilizan como contexto para solicitar la implementación de los cuerpos de los métodos. Su ámbito de actuación queda, por tanto, limitado principalmente a la generación de la lógica, mientras que los elementos estructurales del software permanecen definidos. Por último, las implementaciones obtenidas se integran en el esqueleto para construir el código fuente resultante.

La propuesta establece así una separación entre generación determinista de la estructura y generación no determinista de la lógica. Esta división pretende mantener bajo control del desarrollador aquellos elementos fundamentales para comprender y mantener la aplicación, al mismo tiempo que se automatizan tareas de implementación que mediante transformaciones tradicionales resultarían difíciles de abordar. La documentación adquiere además una doble función: facilita la comprensión posterior del software y proporciona a la Inteligencia Artificial la información necesaria para interpretar el comportamiento esperado.

La hipótesis de trabajo plantea que esta combinación permite aumentar el grado de automatización de la implementación sin delegar en la IA las principales decisiones estructurales y arquitectónicas del sistema. También se considera que la calidad de la generación estará condicionada por la precisión del modelo y por la calidad de la documentación proporcionada, de forma que un modelo suficientemente completo y documentado debería permitir obtener implementaciones coherentes con el diseño previamente establecido.

3.2.- Arquitectura conceptual

La arquitectura conceptual de la propuesta se organiza en varios bloques funcionales que representan las principales responsabilidades del proceso de generación. El objetivo de esta

división es separar las tareas relacionadas con la interpretación del modelo, la gestión de la información, la generación determinista y la generación mediante Inteligencia Artificial.

El proceso comienza con la lectura de un fichero XMI exportado desde una herramienta de modelado UML. Este fichero contiene la información estructural del diagrama de clases y constituye la entrada inicial de la solución. Un analizador sintáctico se encarga de procesar dicha información y transformarla en una representación interna del proyecto, permitiendo trabajar posteriormente con clases, atributos, métodos, parámetros y relaciones de forma comprensible para la solución e independiente del formato original.

Sobre esta representación actúa un bloque de gestión y enriquecimiento de metadatos. Su función consiste en permitir que el desarrollador complete la información procedente del modelo mediante documentación asociada a los distintos elementos software. Esta información semántica resulta crucial para describir el comportamiento esperado de los métodos y constituye parte del contexto que será utilizado en fases posteriores.

Una vez definido y documentado el modelo, interviene el componente de generación determinista. Este bloque utiliza únicamente la información estructural disponible para generar el código fuente, respetando las clases, atributos, métodos, interfaces y relaciones previamente establecidas. De esta manera, la estructura general de la aplicación se obtiene mediante un proceso controlado y reproducible.

El siguiente bloque corresponde a la generación mediante Inteligencia Artificial. A partir del esqueleto generado y de la información semántica incorporada al modelo, se construye el contexto necesario para solicitar la implementación de los cuerpos de los métodos. La IA actúa, por tanto, sobre una estructura previamente definida, sin asumir las principales decisiones arquitectónicas del sistema.

Por último, las implementaciones generadas se integran con el esqueleto determinista para construir el código fuente completo. La interacción entre estos bloques se coordina desde la propia aplicación, permitiendo al usuario revisar el modelo, completar su documentación y ejecutar progresivamente las distintas fases del proceso.

De forma conceptual, la arquitectura puede representarse mediante el siguiente flujo:

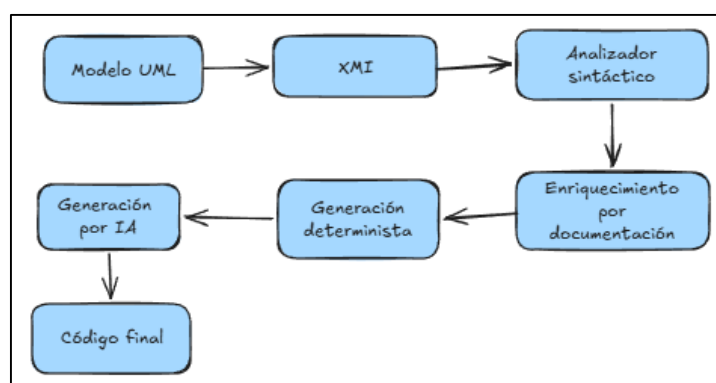


Imagen 1 Arquitectura conceptual

Esta arquitectura conceptual define las responsabilidades generales de la propuesta sin entrar en detalles técnicos de implementación. La organización interna de la aplicación, sus casos de uso y la distribución de componentes se desarrollan posteriormente en el capítulo dedicado al diseño software.

3.3.- Modelo UML enriquecido

La metodología propuesta usa como punto de partida un modelo UML construido previamente por el desarrollador. Para el desarrollo de este trabajo se ha utilizado Modelio como solución de modelado, mediante diagramas de clases en los que se definen las clases, interfaces, enumeraciones, atributos, métodos y relaciones que forman la estructura de la aplicación. Una vez finalizado el modelo, este se exporta en formato XMI utilizando el formato soportado por Modelio, obteniendo un fichero estructurado que puede ser interpretado posteriormente por la aplicación desarrollada.

La estructura de los ficheros XMI generados por la herramienta ha sido estudiada para identificar la representación de los diferentes elementos UML y permitir su transformación hacia el modelo nativo de la aplicación.

Sin embargo, la información estructural proporcionada por un diagrama de clases no resulta suficiente para describir completamente el comportamiento esperado del software. Un método puede definir su nombre, parámetros y tipo de retorno, pero estos elementos no determinan necesariamente cuál debe ser su implementación. Por este motivo, la propuesta introduce el concepto de modelo UML enriquecido, incorporando información semántica mediante documentación asociada a los diferentes elementos del modelo.

Modelio permite incorporar comentarios dentro del propio diagrama. Esta información puede ser recuperada durante el procesamiento del XMI. No obstante, la metodología propuesta no depende únicamente de que la documentación haya sido introducida durante la fase inicial de modelado. Una vez interpretado el fichero, la aplicación presenta al usuario los diferentes elementos recuperados (clases, interfaces, enumeraciones, atributos, métodos, parámetros...) y permite revisar y modificar la documentación asociada a ellos.

Antes de continuar con la generación se comprueba que los elementos relevantes dispongan de una descripción suficiente. Esta información adquiere especial importancia en los métodos, ya que sus comentarios representan el comportamiento que posteriormente deberá interpretar la Inteligencia Artificial para generar su implementación. Por tanto, la documentación deja de tener una finalidad descriptiva y pasa a constituir parte de la especificación utilizada durante el proceso de generación.

El enriquecimiento del modelo pretende así mantener la estructura del sistema definida mediante mecanismos propios de la Ingeniería del Software, complementándola con la información semántica necesaria para describir su comportamiento. Cuanto mayor sea la precisión del modelo y de la documentación asociada, mayor será también el contexto disponible para guiar la generación mediante Inteligencia Artificial.

Sobre el alcance actual del prototipo, la aplicación permite revisar y modificar la documentación de los elementos importados, pero no alterar la estructura del modelo mediante la creación o modificación de clases, atributos, métodos, etc. La incorporación de estas capacidades permitiría convertir la aplicación en un entorno de edición más completo, aunque no resulta necesaria para evaluar la metodología planteada.

3.4.- Decisiones de diseño

Las principales decisiones adoptadas durante el planteamiento de la propuesta han estado orientadas a mantener un equilibrio entre automatización y control. El objetivo no consiste en delegar el proceso completo de desarrollo en una herramienta automática, sino en determinar

qué partes pueden generarse de forma predecible y cuáles pueden beneficiarse de las capacidades de la Inteligencia Artificial.

Una de las decisiones principales consiste en separar la generación estructural de la generación de comportamiento. Todos aquellos elementos que pueden derivarse directamente del modelo UML se obtienen mediante mecanismos deterministas. De esta manera, la definición de clases, interfaces, enumeraciones, atributos, métodos... permanece vinculada al diseño realizado previamente por el desarrollador. La Inteligencia Artificial se reserva principalmente para la implementación de los cuerpos de los métodos, donde las transformaciones tradicionales presentan mayores dificultades debido a la variabilidad de la lógica de negocio.

Relacionado con lo anterior, se ha optado por restringir explícitamente el ámbito de actuación de la Inteligencia Artificial. El modelo no debe decidir la estructura de la solución ni modificar elementos previamente definidos por el desarrollador. Durante la generación se le proporciona una especificación en la que debe respetar nombres, parámetros, tipos de retorno y demás elementos estructurales existentes. Esta decisión diferencia la propuesta de aproximaciones basadas en agentes con acceso amplio al proyecto, donde el sistema puede decidir de manera autónoma qué componentes crear, modificar o reorganizar.

También se ha elegido realizar la generación de la lógica de forma incremental, trabajando sobre clases concretas en lugar de solicitar la implementación completa del proyecto en una única operación. Este planteamiento permite delimitar el contexto proporcionado al modelo, reducir el número de elementos sobre los que puede actuar en cada solicitud y facilitar la posterior validación e integración de las respuestas obtenidas. Al mismo tiempo, el contexto general del proyecto puede mantenerse disponible para que la implementación de una clase sea coherente con el resto de los elementos definidos en el modelo.

Otra decisión importante es considerar la documentación como parte de la especificación y no únicamente como un resultado del proceso. La estructura UML proporciona información suficiente para construir el esqueleto, pero no necesariamente para conocer la lógica que se espera en cada operación. Por esta razón, se requiere que el modelo se encuentre documentado antes de realizar la generación mediante IA. Esta documentación se utiliza como información semántica durante la solicitud y permanece asociada al código generado, favoreciendo tanto su comprensión como la trazabilidad entre la intención funcional y la implementación obtenida.

Por último, se ha optado por mantener una separación entre la herramienta utilizada para realizar el modelado y la aplicación encargada de generar el código. El modelo se intercambia mediante XMI en lugar de establecer una integración directa con una herramienta concreta. Aunque el prototipo desarrollado se ha probado usando Modelio y depende de la estructura XMI soportada por esta solución, conceptualmente esta separación permite distinguir el proceso de modelado del proceso posterior de interpretación y generación.

Como última decisión relacionada con la integración a la Inteligencia Artificial, se ha optado por no incorporar credenciales propias dentro de la aplicación. El acceso al proveedor de IA debe ser configurado por el usuario final mediante sus propias credenciales, de forma que el consumo del servicio y las condiciones asociadas a su uso dependan directamente de su cuenta.

4.- Diseño de la aplicación

4.1.- Arquitectura software

La herramienta desarrollada se ha implementado como una aplicación de escritorio utilizando Windows Forms sobre .NET 4.7.2. La elección de estas tecnologías responde principalmente a la experiencia previa con el entorno y a que proporcionan los componentes necesarios para desarrollar una interfaz orientada a la visualización, edición y procesamiento de la información obtenida de los modelos. Al tratarse de un prototipo cuyo objetivo principal es validar la metodología planteada, se ha priorizado el desarrollo de las funcionalidades necesarias para realizar dicha validación frente a otros aspectos relacionados con el despliegue o la distribución de la aplicación.

El código fuente se organiza bajo un directorio principal `src`, siguiendo una arquitectura modular orientada a funcionalidades, aplicando principios de separación de responsabilidades, Clean Code y SOLID. La solución se divide principalmente en cuatro áreas: Controladores, Dominio, Features y Vistas. Esta separación pretende evitar que la lógica de negocio quede acoplada a la interfaz gráfica y facilitar la evolución independiente de los diferentes componentes.

El directorio Controladores contiene el componente encargado de coordinar el funcionamiento general de la aplicación. Este controlador actúa como orquestador entre la interfaz y las diferentes funcionalidades disponibles, determinando las operaciones que deben ejecutarse y gestionando el intercambio de información entre ellas. Las tareas específicas de análisis, generación, persistencia o comunicación con servicios externos se delegan en los módulos correspondientes, evitando concentrar la lógica funcional dentro del propio controlador.

El Dominio contiene las clases utilizadas para representar internamente la información procesada por la herramienta. Se incluyen elementos como proyectos, directorios, clases, atributos, métodos, parámetros y asociaciones. Estos elementos se organizan mediante diferentes jerarquías de herencia, permitiendo compartir características comunes y disponer de una representación propia independiente de la estructura original del fichero XML. Esta transformación resulta importante, ya que el resto de funcionalidades pueden trabajar sobre un modelo homogéneo sin necesitar conocer los detalles específicos del formato utilizado por la herramienta de modelado.

Dentro del dominio se incluye también una jerarquía de objetos de retorno empleada para comunicar el resultado de las operaciones entre los distintos componentes. A partir de una clase común, `RetornoBase`, se definen especializaciones capaces de incorporar información adicional cuando un proceso necesita devolver objetos o datos concretos. Estos retornos permiten indicar si la ejecución ha finalizado correctamente y proporcionar información diferenciada para el usuario y para el desarrollador. De esta forma se dispone de un mecanismo uniforme para gestionar resultados y errores esperados, evitando utilizar excepciones como mecanismo habitual de control del flujo y permitiendo mostrar mensajes comprensibles al usuario junto con información técnica de mayor detalle cuando resulta necesaria.

La mayor parte de la lógica de la aplicación se encuentra agrupada dentro del directorio Features, que organiza las funcionalidades en módulos independientes. Entre ellos se encuentran el análisis sintáctico de entrada, la generación de código, la integración con Inteligencia Artificial, la persistencia y diferentes utilidades compartidas.

El módulo de análisis sintáctico de entrada contiene los componentes responsables de transformar la información procedente de los modelos externos en el dominio interno de la aplicación. Para evitar que el resto de la solución dependa directamente de un formato concreto, se define una abstracción común para los analizadores y una factoría encargada de seleccionar la implementación correspondiente. Actualmente se encuentra implementado el análisis de diagramas de clases UML exportados mediante XMI, pero esta estructura permite incorporar en el futuro nuevos analizadores manteniendo estable el resto de los componentes.

La funcionalidad generar salida adopta un planteamiento equivalente. Una abstracción común establece las operaciones necesarias para generar código y una factoría selecciona la implementación correspondiente al lenguaje de salida solicitado. En el alcance actual del prototipo se ha desarrollado únicamente el generador de C#. Este componente transforma el modelo interno en un esqueleto de código, incorpora la documentación asociada a sus elementos y establece los puntos en los que posteriormente se integrará la lógica producida mediante Inteligencia Artificial. La generación determinista puede ejecutarse de manera independiente para obtener únicamente el esqueleto o formar parte del proceso completo de generación.

La integración con Inteligencia Artificial constituye otro punto de extensión de la arquitectura. Se define una abstracción común para los distintos proveedores, evitando que el controlador o la interfaz gráfica dependan directamente de una implementación concreta. En la versión actual se ha desarrollado la integración con OpenAI, aunque la arquitectura permite incorporar otros proveedores mediante nuevas implementaciones de la interfaz definida. El uso de estas abstracciones y factorías busca aplicar especialmente el principio abierto/cerrado de SOLID, favoreciendo la extensión del sistema sin necesidad de modificar significativamente los componentes existentes.

La persistencia se encapsula mediante componentes DAO (*Data Access Object*), responsables de almacenar y recuperar la información manejada por la aplicación. Entre sus funciones se encuentra la serialización del proyecto en ficheros JSON, lo que permite conservar la representación interna generada a partir del modelo y recuperarla posteriormente. Esta información estructurada puede utilizarse además durante el proceso de comunicación con la Inteligencia Artificial como parte del contexto necesario para realizar la generación.

Por otra parte, el módulo Utilidades agrupa funcionalidades de carácter transversal utilizadas desde diferentes puntos de la solución, como la gestión de ficheros, la búsqueda de componentes o la generación de identificadores. Su separación evita duplicar operaciones comunes dentro de los distintos casos de uso.

Por último, el directorio Vistas contiene los componentes encargados de la interacción con el usuario. En el prototipo actual existe una interfaz principal desarrollada mediante Windows Forms desde la que se accede a las diferentes funcionalidades. La vista recoge las acciones del usuario y muestra los resultados, delegando la coordinación de las operaciones en el controlador y la lógica específica en los módulos correspondientes.

Uno de los aspectos relevantes de la arquitectura es, por tanto, la existencia de tres puntos principales de extensión: el formato de entrada, el lenguaje de generación y el proveedor de Inteligencia Artificial. Las abstracciones definidas para estos componentes pretenden que la incorporación futura de nuevos analizadores sintácticos, generadores de código o servicios de IA pueda realizarse mediante nuevas implementaciones sin modificar de forma significativa la interfaz, el controlador o las restantes funcionalidades.

En la Imagen 11 Diagrama de clases simplificado se muestra un diagrama simplificado de las principales clases de la solución. Con el objetivo de mantener su legibilidad, se representan fundamentalmente las relaciones de herencia, generalización e implementación, omitiéndose gran parte de las asociaciones, agregaciones y dependencias existentes entre los componentes. El modelo completo puede consultarse junto con el código fuente y la documentación técnica adjunta al trabajo.

4.2.- Casos de uso

Los casos de uso permiten representar las principales funcionalidades ofrecidas por la aplicación desde el punto de vista de su utilización. En este trabajo se han seleccionado aquellos casos de uso que intervienen directamente en el proceso principal de generación, desde la importación y tratamiento del modelo hasta la obtención del código final.

Dado que la herramienta desarrollada tiene como objetivo principal validar la metodología propuesta, no se pretende realizar una especificación exhaustiva de todas las operaciones auxiliares disponibles en la interfaz. Por este motivo, se describen únicamente los casos de uso principales mediante una estructura común que recoge su objetivo, actores, precondiciones, flujo principal, posibles alternativas y resultado esperado.

Tabla 1 Caso de uso 1

Campo	Descripción
Identificador	CU.01
Nombre	Importación y análisis del modelo XML.
Actor principal	Desarrollador
Objetivo	Importar un modelo UML exportado en formato XMI y transformar su información en el modelo interno utilizado por la aplicación.
Precondiciones	El usuario debe disponer de un fichero XML compatible con el formato soportado por la aplicación.
Flujo principal	1. El usuario selecciona el fichero XML. 2. La aplicación comprueba que el fichero puede ser leído. 3. El analizador sintáctico procesa su contenido. 4. Se identifican los elementos UML soportados, como clases, interfaces, enumeraciones, atributos, métodos, parámetros y relaciones. 5. La información obtenida se transforma al dominio interno de la aplicación. 6. El modelo resultante queda disponible para su consulta y posterior procesamiento.
Errores	El fichero no existe, no puede ser leído, no contiene una estructura XML compatible o presenta información que no puede ser interpretada correctamente. En estos casos la operación se interrumpe y se devuelve información controlada sobre el error producido.
Post-condiciones	El modelo UML queda cargado y representado mediante los objetos de dominio internos de la aplicación.
Diagrama	Imagen 12 Caso de uso 1

Tabla 2 Caso de uso 2

Campo	Descripción
Identificador	CU.02
Nombre	Revisión y modificación de descripciones
Actor principal	Desarrollador
Objetivo	Permitir al usuario revisar y completar la documentación asociada a los elementos importados del modelo UML antes de iniciar el proceso de generación.
Precondiciones	Debe existir un proyecto previamente importado y analizado mediante el caso de uso CU.01 o CU.05.
Flujo principal	1. La aplicación muestra los elementos obtenidos del modelo. 2. El usuario selecciona clases, interfaces, enumeraciones, atributos, métodos u otros elementos disponibles. 3. Se muestra la documentación asociada a cada elemento. 4. El usuario revisa, añade o modifica las descripciones necesarias. 5. La información actualizada queda incorporada al modelo interno del proyecto.
Errores	La aplicación detecta elementos que no disponen de la documentación considerada necesaria para continuar con la generación. En estos casos se informa al usuario para que complete la información requerida si lo considera necesario.
Post-condiciones	El modelo queda enriquecido con la información semántica necesaria para facilitar la generación del código y, especialmente, la interpretación del comportamiento esperado de los métodos por parte de la IA.
Diagrama	Imagen 13 Caso de uso 2

Tabla 3 Caso de uso 3

Campo	Descripción
Identificador	CU.03
Nombre	Generación del esqueleto de código
Actor principal	Desarrollador
Objetivo	Generar de forma determinista el esqueleto del código fuente a partir de la estructura y documentación contenidas en el modelo interno.
Precondiciones	Debe existir un proyecto previamente importado y analizado mediante el caso de uso CU.01 o CU.05.
Flujo principal	1. El usuario solicita la generación del esqueleto o del código completo. 2. La aplicación selecciona el generador correspondiente al lenguaje de salida. 3. El generador recorre los elementos del modelo interno. 4. Se genera la clase seleccionada por el usuario. 5. Se incorpora la documentación asociada a los elementos. 6. Se introducen los puntos necesarios para permitir posteriormente la integración de los cuerpos generados mediante IA. 7. El esqueleto resultante se presenta al usuario.
Errores	El modelo contiene información insuficiente o incompatible con el generador seleccionado, o se produce un error durante la construcción del código. La operación finaliza devolviendo la información correspondiente al error.

Post-condiciones	Se obtiene un esqueleto de código fuente coherente con la estructura definida previamente en el modelo UML y preparado para ser completado mediante Inteligencia Artificial.
Diagrama	Imagen 14 Caso de uso 3

Tabla 4 Caso de uso 4

Campo	Descripción
Identificador	CU.04
Nombre	Generación de código mediante Inteligencia Artificial
Actor principal	Desarrollador
Objetivo	Obtener la implementación de los cuerpos de los métodos mediante un proveedor de Inteligencia Artificial e integrarla con el esqueleto generado previamente para construir el código final.
Precondiciones	El proyecto debe disponer de un modelo correctamente importado y documentado. Debe poder generarse un esqueleto válido y el usuario debe haber configurado unas credenciales válidas para acceder al proveedor de IA seleccionado.
Flujo principal	1. El usuario solicita la generación completa del código. 2. La aplicación genera o recupera el esqueleto correspondiente. 3. Se prepara la información estructural y semántica necesaria como contexto para la solicitud. 4. Se selecciona el proveedor de IA configurado. 5. La aplicación realiza las solicitudes necesarias para obtener la implementación de los métodos. 6. Se procesa la respuesta del modelo y se comprueba que puede asociarse con los elementos correspondientes. 7. Las implementaciones generadas se integran en el esqueleto. 8. El código completo se muestra al usuario.
Errores	No existen credenciales configuradas, las credenciales son inválidas, el servicio externo no está disponible, se produce un error durante la comunicación, la respuesta generada no presenta el formato esperado o no puede integrarse correctamente con el esqueleto. En estos casos se devuelve información controlada al usuario.
Post-condiciones	Se obtiene el código fuente completo, manteniendo la estructura generada de forma determinista e incorporando los cuerpos de los métodos generados mediante IA.
Diagrama	Imagen 15 Caso de uso 4

Tabla 5 Caso de uso 5

Campo	Descripción
Identificador	CU.05
Nombre	Persistencia y recuperación del proyecto
Actor principal	Desarrollador
Objetivo	Almacenar la información del proyecto para permitir su recuperación en ejecuciones posteriores y reutilizar sus datos dentro de los distintos procesos de la aplicación.

Precondiciones	Para guardar un proyecto debe existir información cargada en el dominio de la aplicación. Para recuperarlo debe existir previamente un fichero persistido en un formato compatible.
Flujo principal	1. El usuario solicita guardar o recuperar un proyecto. 2. La aplicación delega la operación en el componente DAO correspondiente. 3. En una operación de guardado, la información del dominio se serializa y almacena en los ficheros correspondientes. 4. En una operación de recuperación, la información persistida se lee y se transforma nuevamente en los objetos internos de la aplicación. 5. El estado recuperado queda disponible para continuar trabajando con el proyecto.
Errores	No es posible acceder al fichero, la ruta no es válida, el contenido almacenado no presenta el formato esperado o se produce un error durante la serialización o deserialización. La aplicación devuelve información controlada sobre el problema detectado.
Post-condiciones	La información del proyecto queda almacenada de forma persistente o, en el caso de recuperación, vuelve a estar disponible dentro del dominio de la aplicación.
Diagrama	Imagen 16 Caso de uso 5

Los casos de uso descritos representan las funcionalidades principales de la aplicación y sirven como base para organizar sus distintos módulos. Además, existen operaciones auxiliares relacionadas con formularios, configuración y persistencia de credenciales, como la clave de acceso a la API de OpenAI, que debe ser proporcionada por el usuario final. Al no formar parte directa del flujo principal de generación, estas funcionalidades no se tienen en cuenta para las tablas de casos de uso de manera individual.

4.3.- Modelo de dominio

En este apartado se describe la información que almacena la solución y la forma en que se representa internamente. La mayoría de los objetos con los que trabaja la herramienta se encuentran dentro del directorio Dominio/, donde se distinguen dos subdirectorios y dos clases de carácter general.

El primer directorio, ElementosUML/, contiene los elementos UML utilizados por la aplicación durante su ejecución y representa los artefactos extraídos de los diagramas de clases. Todos ellos parten de una clase principal denominada ElementoBase, que almacena un identificador, el nombre del elemento, su tipo y el comentario asociado. De esta forma, todos los elementos que heredan de esta clase disponen de dicha información común.

Dependiendo de su naturaleza, los elementos pueden descender directamente de ElementoBase o de una segunda clase denominada ElementoVisible, que extiende la anterior incorporando la información relativa a la visibilidad. Los principales elementos gestionados por la aplicación son los siguientes:

- **Asociacion.** Almacena la información general de una asociación, incluyendo su navegabilidad, el tipo de asociación, los identificadores de las clases involucradas y la multiplicidad de la relación.
- **Atributo.** Almacena de forma específica la información correspondiente al tipo del atributo.

- Clase. Contiene información sobre el tipo de clase y mantiene diferentes listados para almacenar sus atributos, métodos y asociaciones. En el caso de que la clase represente una enumeración, también dispone de un listado de literales.
- Directorio. Se trata de un elemento estructural encargado de representar la organización de los artefactos y no un elemento UML propiamente dicho. Almacena mediante listados tanto las clases contenidas en el directorio, para cada una de las cuales se generará posteriormente un fichero, como sus posibles subdirectorios. Debido a que un directorio puede contener otros directorios, el recorrido completo de la estructura requiere la utilización de métodos recursivos.
- Metodo. Almacena la información correspondiente a un método, incluyendo su tipo de retorno, un listado de parámetros y dos valores booleanos destinados a indicar si se trata de un constructor, un destructor o ninguno de ellos.
- Parametro. Almacena específicamente la información relativa al tipo de un parámetro. Aunque esta información podría haberse unificado con la clase `Atributo`, se ha optado por mantener ambas clases separadas para permitir que puedan evolucionar de forma independiente si en el futuro alguna de ellas necesita incorporar información adicional. De esta forma se evita que cambios específicos en uno de estos elementos obliguen a modificar otros puntos del código.
- ProyectoBase. Al igual que Directorio, constituye un elemento estructural. Contiene un listado de directorios mediante el que se representa la estructura general del proyecto. Al heredar de ElementoBase, también almacena el nombre, comentario e identificador del proyecto, permitiendo diferenciar distintos proyectos cuando sea necesario. El identificador asignado al proyecto es un GUID generado durante el análisis sintáctico del fichero XML.

Junto a estos elementos se encuentra la clase Tipos, que agrupa diferentes enumeraciones utilizadas para representar características del dominio, como el tipo de elemento, el tipo de clase, el tipo de asociación, la visibilidad o el tipado de métodos y datos.

El segundo directorio, Retornos/, contiene los objetos utilizados para devolver información entre los distintos métodos y mantener un control uniforme de los resultados, errores y excepciones. Todos los objetos de retorno heredan de una clase principal denominada RetornoBase, que almacena un mensaje destinado al usuario, información técnica detallada para el desarrollador y un valor booleano que indica si la ejecución ha finalizado correctamente.

A partir de esta clase se definen los siguientes retornos:

- RetornoConfiguracion. Devuelve la información correspondiente a la ejecución junto con un objeto de tipo Configuracion, descrito posteriormente en este mismo apartado.
- RetornoElementoUML. Devuelve información sobre el resultado del proceso junto con un elemento UML de los descritos anteriormente.
- RetornoMultiple. Este retorno se utiliza en aquellas ejecuciones que requieren la confirmación del usuario o que no pueden continuar debido a una actuación del propio usuario o a la ausencia de determinada información. En estos casos, la ejecución puede finalizar sin éxito sin que necesariamente se haya producido un error o una excepción. Para representar estas situaciones, RetornoMultiple amplía la información de RetornoBase mediante una enumeración que permite identificar el estado concreto de la ejecución y decidir si el funcionamiento de la aplicación puede continuar.

Además de los elementos anteriores, se definen dos clases destinadas al control general de la información y a la toma de decisiones de la solución.

La primera es la clase Configuración, preparada para actuar como objeto principal de persistencia del proyecto. Almacena el tipo de diagrama leído, el tipo de fichero de entrada, el tipo de salida esperado, la información del proyecto importado, la ruta del fichero de entrada, la ruta del fichero de salida y la ruta del fichero JSON utilizado para la persistencia externa de la aplicación. El controlador principal instancia un objeto de este tipo y mantiene en él la información necesaria durante la ejecución. Al exportar un proyecto se serializa este objeto y, al recuperar una configuración previamente almacenada, se realiza el proceso inverso mediante su deserialización.

La segunda clase agrupa las enumeraciones utilizadas para definir las capacidades disponibles en la aplicación. Aunque su relevancia dentro del alcance actual del prototipo es limitada, su importancia aumentaría en caso de ampliar la herramienta, ya que determina los formatos de entrada admitidos, los tipos de diagramas soportados y los formatos de salida disponibles. Esta información constituye la base sobre la que se apoyan las diferentes factorías de la solución.

Por último, la funcionalidad de integración con Inteligencia Artificial dispone de un dominio secundario propio, ubicado también en un directorio Dominio/, destinado a gestionar la información de los ficheros utilizados durante este proceso. La funcionalidad trabaja con un fichero JSON que representa el objeto ProyectoBase y con los esqueletos de código, controlando tanto sus ubicaciones en el equipo local como los identificadores asignados por el proveedor de IA para poder eliminarlos posteriormente.

Para representar esta información se define una clase base denominada InformaciónFichero, que almacena un identificador, la ruta del fichero y el identificador asignado por la IA. También existe un objeto encargado de mantener la información correspondiente al JSON del proyecto y un listado con la información de los esqueletos. Finalmente, se define RetornoFicheroIA, que hereda de RetornoBase y permite devolver, junto con el resultado de la operación, un objeto con la información asociada al fichero gestionado.

4.4.- Diseño de funcionalidades

La aplicación implementa sus principales funcionalidades de forma independiente, agrupando la lógica asociada a cada responsabilidad dentro de subdirectorios específicos del directorio Features/. Esta organización busca mantener una separación clara entre las distintas tareas realizadas por la herramienta y aplicar principios de diseño como la responsabilidad única, el bajo acoplamiento y la extensibilidad, en consonancia con los principios SOLID utilizados durante el desarrollo.

Cada funcionalidad concentra las clases necesarias para llevar a cabo un proceso concreto, evitando distribuir su lógica entre componentes no relacionados. Este planteamiento facilita su mantenimiento, modificación y posible reutilización. En los siguientes subapartados se describe el diseño de cada una de las funcionalidades desarrolladas y la responsabilidad que desempeñan dentro del proceso general de la herramienta.

4.4.1.- Persistencia (DAO)

La funcionalidad DAO/ agrupa los componentes destinados a transformar determinados objetos del dominio en representaciones persistentes y recuperar la información cuando sea necesario. En la versión actual se han definido dos clases con responsabilidades diferentes: ConfiguraciónDAO y ProyectoDAO.

ConfiguracionDAO gestiona la persistencia de la configuración general de la aplicación. Permite serializar un objeto Configuración en formato JSON y realizar el proceso inverso para recuperar un proyecto previamente almacenado. De esta forma, el estado necesario para continuar trabajando con un proyecto puede conservarse entre distintas ejecuciones sin que el resto de componentes de la solución tenga que conocer los detalles de serialización y acceso al fichero.

Por otro lado, ProyectoDAO trabaja directamente con el objeto ProyectoBase y tiene una finalidad diferente. Su responsabilidad es generar una representación JSON de la información estructural obtenida del modelo UML. Este fichero se utiliza principalmente dentro del proceso de generación mediante Inteligencia Artificial, proporcionando al modelo información global sobre el proyecto. Durante esta serialización las enumeraciones se representan mediante sus valores textuales en lugar de mediante valores numéricos, facilitando que la información enviada resulte directamente interpretable por el modelo.

La separación entre ambos DAO responde a que gestionan objetos y finalidades distintas. Mientras que ConfiguraciónDAO está orientado a conservar y recuperar el estado de trabajo de la aplicación, ProyectoDAO genera una representación del dominio destinada principalmente a actuar como información de contexto para otras funcionalidades. Esta separación permite mantener aisladas las responsabilidades de persistencia y evita incorporar lógica de serialización dentro del controlador o de los componentes encargados de la generación.

4.4.2.- Análisis sintáctico (ParserEntrada)

La funcionalidad ParserEntrada/ es responsable de interpretar los modelos proporcionados como entrada y transformarlos al dominio interno de la aplicación. Su diseño busca desacoplar el resto de la solución del formato concreto utilizado para representar el modelo, de manera que las funcionalidades posteriores trabajen exclusivamente con objetos propios del dominio.

Para ello se define una clase abstracta ParserUMLBase, que establece la operación común de análisis y mantiene la ruta del fichero que debe procesarse. Las implementaciones específicas heredan de esta clase y son creadas mediante FactoríaParser, que selecciona el analizador correspondiente a partir del tipo de fichero definido en la configuración. Actualmente el único formato implementado es XMI, aunque esta estructura permite incorporar nuevos analizadores mediante nuevas implementaciones de la clase base.

La implementación actual se encuentra en ParserXMI. Este componente carga el documento XMI, comprueba que su estructura sea compatible con el subconjunto soportado por la herramienta y recorre sus elementos para construir un objeto ProyectoBase. Durante este proceso se identifican y transforman los paquetes, clases, interfaces, enumeraciones, atributos, métodos, parámetros y relaciones reconocidos por la aplicación. Debido a que el modelo puede contener directorios y elementos anidados, parte del procesamiento se realiza de forma recursiva hasta reconstruir la estructura completa del proyecto.

El resultado de esta funcionalidad es, por tanto, la transformación de una representación externa dependiente del formato XMI en un modelo propio e independiente del fichero original. Esta decisión permite que componentes como el generador de código, la persistencia o la integración con Inteligencia Artificial no necesiten conocer la estructura interna utilizada por Modelio para representar los elementos UML.

4.4.3.- Generación de esqueletos (GenerarSalida)

La funcionalidad GenerarSalida/ contiene la lógica encargada de transformar el modelo interno del proyecto en código fuente mediante un proceso determinista. Su diseño sigue una

estructura similar a la utilizada para el análisis de entrada, permitiendo separar la lógica común de generación de las particularidades asociadas a cada posible lenguaje de salida.

La clase abstracta `GenerarSalidaBase` mantiene la información del proyecto y define la operación necesaria para generar el código correspondiente a una clase. La creación de la implementación concreta se delega en `FactoriaSalida`, que selecciona el generador según el tipo de salida definido en la configuración. En el alcance actual del prototipo únicamente se encuentra implementada la generación de código C#, mediante la clase `C_Sharp`.

El generador de C# recorre la información almacenada en el dominio y construye el esqueleto correspondiente a la clase seleccionada. Durante este proceso se generan elementos como el espacio de nombres, la declaración de la clase, sus atributos, métodos, parámetros, enumeraciones y las relaciones necesarias con otros elementos del proyecto. También se incorpora al código la documentación definida previamente por el desarrollador.

La generación se realiza sobre una clase concreta y no sobre la totalidad del proyecto en una única operación, en consonancia con las decisiones descritas anteriormente. En los cuerpos de los métodos se incorporan identificadores que permiten reconocer posteriormente los puntos que deben ser completados mediante Inteligencia Artificial. De forma similar, el esqueleto reserva posiciones específicas para aquellas librerías o funciones auxiliares que puedan resultar necesarias durante esta segunda fase. El resultado continúa siendo, por tanto, un código estructuralmente definido de manera determinista, preparado para ser utilizado de forma independiente o como entrada de la funcionalidad de generación mediante IA.

El uso de una abstracción y una factoría permite mantener desacoplado el proceso general del lenguaje de destino. Aunque el prototipo se encuentra limitado actualmente a C#, la incorporación de un nuevo lenguaje podría realizarse mediante un nuevo generador que implemente el contrato definido por la clase base, manteniendo estable el resto del flujo de la aplicación.

4.4.4.- Integración con Inteligencia Artificial (IntegracionIA)

La funcionalidad `IntegracionIA/` concentra los componentes responsables de la comunicación con proveedores externos de Inteligencia Artificial y de la incorporación de los resultados obtenidos al código generado previamente. Su diseño pretende evitar que el controlador y las demás funcionalidades dependan directamente de un proveedor concreto.

La interfaz `IProveedorIA` establece el contrato que deben cumplir las distintas integraciones. La operación principal recibe la información global del proyecto, la clase que se desea implementar y su esqueleto previamente generado. A partir de esta abstracción, `FactoriaProveedorIA` selecciona la implementación correspondiente al proveedor configurado. Actualmente únicamente se encuentra disponible la integración con OpenAI, aunque el uso de una interfaz común permite incorporar otros proveedores sin modificar el funcionamiento general de la solución.

La implementación OpenAI se encarga de preparar la información necesaria y realizar la comunicación con el servicio. Para cada proceso de generación se utiliza como contexto tanto la representación estructurada del proyecto como el esqueleto C# de la clase concreta. De esta forma, la IA dispone de información global sobre los elementos y relaciones existentes, pero su actuación se centra en una unidad de generación delimitada. Las instrucciones proporcionadas al modelo restringen expresamente la modificación de la estructura existente y establecen que

los identificadores incluidos en el esqueleto deben utilizarse para asociar cada implementación con el método correspondiente.

La funcionalidad, como se ha comentado con anterioridad, dispone de un pequeño dominio propio para gestionar los ficheros utilizados durante la comunicación. En él se conserva información sobre los ficheros generados localmente, la clase a la que pertenecen cuando corresponde y los identificadores asignados por el proveedor tras su subida. Esto permite mantener control sobre los recursos creados durante el proceso y gestionar posteriormente su eliminación.

El acceso al proveedor se realiza utilizando las credenciales configuradas por el usuario final. La aplicación no incorpora una clave de acceso propia, sino que recibe la necesaria para crear la implementación seleccionada. De este modo, la funcionalidad de generación permanece desacoplada tanto del proveedor concreto como de una cuenta específica perteneciente al desarrollador de la herramienta.

4.4.5.- Utilidades

El directorio Utilidades/ contiene operaciones de carácter transversal usadas desde diferentes funcionalidades de la aplicación. Estos componentes no representan un caso de uso independiente, sino que encapsulan tareas comunes con el objetivo de evitar su duplicación y mantenerlas fuera de aquellos módulos cuya responsabilidad principal es diferente.

Entre estas utilidades se encuentra Ficheros, encargada de centralizar operaciones relacionadas con la selección, lectura, escritura, creación temporal y eliminación de archivos. De forma equivalente, Folder proporciona la funcionalidad necesaria para seleccionar directorios mediante la interfaz del sistema. Estas operaciones son utilizadas desde distintos puntos de la aplicación, especialmente durante la importación, persistencia y preparación de los ficheros necesarios para la generación.

La clase CrearIdentificador centraliza la creación de identificadores propios mediante GUID. Estos identificadores permiten disponer de referencias únicas para aquellos elementos que no proceden directamente del fichero de entrada o que necesitan ser identificados internamente por la aplicación. Por el momento solo se generan identificadores de tipo GUID, pero esta clase está concebida para la generación de distintos tipos de identificadores.

Por último, BuscarComponentes agrupa las operaciones necesarias para localizar elementos UML dentro de la estructura del proyecto a partir de sus identificadores. Debido a que el dominio puede contener diferentes niveles de directorios y subdirectorios, las búsquedas recorren de forma recursiva la estructura hasta localizar clases, atributos, métodos, parámetros, asociaciones o literales. Esta funcionalidad resulta especialmente útil para resolver las referencias entre elementos sin duplicar la lógica de búsqueda en los distintos componentes que trabajan con el modelo.

La agrupación de estas operaciones en un módulo común permite reutilizar comportamientos transversales desde las diferentes funcionalidades, manteniendo sus responsabilidades específicas centradas en el análisis, persistencia y generación.

4.5.- Controlador e interfaz de usuario

Los componentes descritos en los apartados anteriores necesitan un elemento encargado de coordinar su utilización y una interfaz mediante la que el usuario pueda interactuar con las

distintas funcionalidades. Para cubrir estas responsabilidades, la aplicación dispone de un controlador principal y de una interfaz gráfica desarrollada mediante Windows Forms.

El controlador principal actúa como elemento de coordinación entre la interfaz, el dominio y las funcionalidades implementadas en Features/. Mantiene durante la ejecución el objeto Configuración, que contiene el proyecto actualmente cargado y los parámetros necesarios para determinar los formatos de entrada y salida y las rutas utilizadas por la aplicación. También mantiene de forma independiente la clave de acceso a OpenAI, evitando que esta información forme parte del objeto de configuración que puede ser exportado junto con el proyecto.

Su principal responsabilidad es organizar los distintos procesos solicitados desde la interfaz. De esta forma, para la importación de un modelo selecciona el parser adecuado y almacena en la configuración el dominio obtenido, para la generación del esqueleto selecciona el generador correspondiente y para la generación completa coordina sucesivamente la validación de la información, la creación del contexto necesario, la generación determinista del esqueleto, la comunicación con el proveedor de Inteligencia Artificial y la posterior integración de la respuesta obtenida. El controlador también centraliza determinadas operaciones relacionadas con la actualización de la información del dominio y la importación o exportación de la configuración.

Esta organización permite que las distintas funcionalidades puedan mantenerse separadas, mientras que el controlador se encarga de establecer el orden en el que deben ejecutarse y de transmitir sus resultados a la interfaz. Los objetos de retorno definidos en el dominio se usan en este flujo para comunicar tanto resultados correctos como errores, cancelaciones o situaciones que requieren una decisión del usuario.

La interfaz de usuario se concentra actualmente en un formulario principal desde el que se accede a las funcionalidades incluidas en el prototipo. Su diseño pretende acompañar al desarrollador durante las diferentes etapas del proceso propuesto, permitiéndole importar el modelo, revisar la información obtenida, completar su documentación y generar posteriormente el código.

Una vez cargado un proyecto, su estructura se representa mediante un explorador jerárquico que reproduce los directorios y los diferentes tipos de clases contenidos en el modelo. A partir de esta estructura el usuario puede seleccionar los elementos sobre los que desea trabajar. La información de las clases, métodos, atributos, parámetros, asociaciones y enumeraciones se presenta mediante diferentes componentes de la interfaz, permitiendo consultar y modificar principalmente la documentación asociada a cada elemento antes de iniciar la generación.

La misma interfaz proporciona las acciones necesarias para generar de forma independiente el esqueleto determinista de una clase o solicitar su implementación completa mediante Inteligencia Artificial. El código resultante se muestra dentro del propio espacio de trabajo, permitiendo al desarrollador revisar el resultado obtenido. Además, se incluyen opciones para importar y exportar proyectos, configurar las credenciales necesarias para el proveedor de IA y seleccionar las rutas utilizadas durante los distintos procesos.

Durante las operaciones que pueden requerir un mayor tiempo de ejecución, especialmente la generación mediante IA, la interacción con determinadas partes de la interfaz se bloquea temporalmente y se muestra el progreso de la operación. De igual forma, los avisos, errores y solicitudes de confirmación se trasladan al usuario mediante mensajes diferenciados según el resultado proporcionado por el controlador.

De esta manera, la interfaz no se plantea únicamente como un mecanismo para introducir un modelo y obtener automáticamente código, sino como un espacio de trabajo en el que el desarrollador puede revisar y enriquecer la información antes de iniciar la generación. Este planteamiento mantiene la participación del usuario dentro del proceso y responde al objetivo de la propuesta de combinar automatización con control sobre la estructura y la documentación del software generado.

5.- Implementación del prototipo

Una vez descrita la arquitectura y el diseño de los diferentes componentes de la solución, este capítulo recoge los principales aspectos relacionados con su implementación. Debido a que la organización de clases, responsabilidades y funcionalidades se ha desarrollado previamente en el capítulo de diseño, no se pretende realizar una descripción exhaustiva del código fuente, sino presentar las tecnologías utilizadas y aquellos mecanismos de implementación especialmente relevantes para materializar la propuesta.

5.1.- Tecnologías y entorno de desarrollo

El prototipo se ha desarrollado en C# como una aplicación de escritorio basada en Windows Forms y ejecutada sobre .NET Framework 4.7.2. El proyecto se genera como una aplicación ejecutable de Windows y se ha configurado para la plataforma AnyCPU, permitiendo que el mismo ensamblado pueda ejecutarse en diferentes arquitecturas compatibles con el framework utilizado.

La elección de Windows Forms responde principalmente al alcance del trabajo y a la experiencia previa con esta tecnología. El objetivo de la aplicación no consiste en evaluar un determinado framework de interfaz ni en desarrollar una solución destinada a su distribución comercial, sino en disponer de un entorno de escritorio que permita validar de forma práctica la metodología propuesta. Windows Forms proporciona los componentes necesarios para representar jerárquicamente el proyecto, editar la información recuperada del modelo, presentar el código generado y controlar las diferentes operaciones desde una misma interfaz.

El tratamiento de los documentos XML se realiza mediante las capacidades proporcionadas por LINQ to XML, principalmente a través de System.Xml.Linq. Los ficheros se cargan como documentos XML y se recorren sus elementos y atributos para identificar aquellas construcciones UML que forman parte del subconjunto soportado por la herramienta. El uso de las librerías incluidas en el ecosistema .NET evita introducir dependencias adicionales para el procesamiento del formato de entrada.

Para la representación persistente de la información se utiliza JSON mediante System.Text.Json. Esta tecnología se emplea con dos finalidades diferenciadas. Por una parte, permite serializar la configuración de la aplicación para conservar y recuperar el estado de un proyecto. Por otra, se utiliza para construir una representación estructurada de ProyectoBase destinada a proporcionar a la IA información global sobre el modelo. En este segundo caso las enumeraciones se serializan mediante sus nombres textuales, de forma que los valores puedan ser interpretados directamente durante la generación sin necesidad de conocer su representación numérica interna.

La integración con Inteligencia Artificial se ha implementado mediante el paquete oficial OpenAI para .NET, versión 2.12.0. Esta dependencia proporciona los clientes necesarios para gestionar la comunicación con el servicio, incluyendo las operaciones relacionadas con los ficheros utilizados como contexto y las solicitudes de generación. Debido a la naturaleza remota de estas operaciones, parte de esta funcionalidad se implementa mediante programación asíncrona utilizando Task, evitando bloquear innecesariamente el flujo de ejecución mientras se espera la respuesta del proveedor.

La utilización del SDK de OpenAI incorpora además diferentes dependencias auxiliares relacionadas con comunicación, serialización y ejecución asíncrona. Estas dependencias se gestionan mediante paquetes NuGet y permanecen aisladas principalmente dentro de la

funcionalidad encargada de la integración con IA. De esta manera, el resto de la solución no necesita conocer los detalles técnicos del cliente utilizado para comunicarse con el proveedor.

El código que la herramienta genera tiene como versión objetivo C# 7.3. Esta restricción se incorpora también a las instrucciones enviadas al modelo de Inteligencia Artificial, evitando que la implementación generada utilice construcciones añadidas en versiones posteriores del lenguaje. El objetivo es que tanto el esqueleto generado de forma determinista como los cuerpos producidos mediante IA utilicen una sintaxis compatible y puedan integrarse sin introducir diferencias derivadas de distintas versiones del lenguaje.

En conjunto, las tecnologías usadas responden a un criterio de simplicidad y adecuación al prototipo. C# y .NET proporcionan tanto la interfaz de escritorio como las capacidades necesarias para tratar XML, JSON y operaciones asíncronas, mientras que la única dependencia directamente vinculada con la generación mediante Inteligencia Artificial queda encapsulada tras la abstracción definida para los proveedores. Esta combinación permite centrar la implementación en la validación del proceso propuesto sin incorporar infraestructura adicional que no resulte necesaria para los objetivos del trabajo.

5.2.- Transformación del XMI al modelo interno

La implementación del análisis del fichero XMI parte de su carga en memoria mediante un objeto XDocument. Antes de procesar el contenido se realiza una comprobación básica de su estructura, verificando la presencia de los elementos necesarios para que pueda ser interpretado por el subconjunto XMI soportado por la aplicación.

A partir de este documento, el parser recorre los elementos relevantes y determina el tipo de artefacto que representa cada uno. Elementos como paquetes, clases, atributos, operaciones, parámetros, enumeraciones o relaciones se transforman progresivamente en los objetos correspondientes del dominio interno. El recorrido se realiza de forma recursiva cuando existen paquetes o estructuras anidadas, reproduciendo así dentro de ProyectoBase la organización original del proyecto.

Una peculiaridad de este proceso aparece en aquellas relaciones que hacen referencia a elementos que todavía no han sido procesados. Para evitar depender del orden en el que aparecen dentro del XMI, determinadas asociaciones se almacenan temporalmente y se enlazan una vez creados los elementos necesarios. El resultado final del análisis no mantiene referencias directas al documento XML, sino que produce una representación propia sobre la que trabajan posteriormente el resto de las funcionalidades.

Esta transformación constituye la frontera entre el formato de entrada y la aplicación. Una vez finalizado el análisis, la generación de código y la integración con IA se realizan exclusivamente utilizando el dominio interno, evitando que estas funcionalidades tengan que interpretar nuevamente la estructura del XMI.

5.3.- Generación determinista del esqueleto

La generación del esqueleto se implementa recorriendo los objetos creados durante el análisis y traduciendo cada elemento a su representación equivalente en C#. Para la clase seleccionada se determina su espacio de nombres a partir de su posición dentro de la estructura de directorios y se generan posteriormente su declaración, relaciones, atributos, métodos, parámetros y documentación.

La implementación diferencia los distintos tipos de elementos definidos en el modelo. Las enumeraciones generan directamente sus literales, mientras que clases e interfaces reciben un tratamiento adaptado a las características que pueden contener. Del mismo modo, la generación de métodos tiene en cuenta su visibilidad, parámetros, tipo de retorno y su posible condición de constructor o destructor.

Un aspecto específico de la propuesta es la incorporación de marcadores dentro del código generado. Los métodos cuya lógica debe ser implementada posteriormente contienen un identificador relacionado con el elemento correspondiente del dominio. De esta forma, el esqueleto no solo define la estructura del código, sino que establece los puntos exactos en los que posteriormente puede incorporarse la respuesta proporcionada por la IA.

También se reservan posiciones destinadas a incorporar librerías adicionales o funciones auxiliares cuando resulten necesarias. Estos elementos no modifican la estructura definida por el desarrollador, sino que permiten completar técnicamente la implementación obtenida durante la fase generativa.

5.4.- Generación mediante Inteligencia Artificial

La generación mediante IA se implementa como una segunda etapa posterior a la construcción determinista del esqueleto. Para cada clase se prepara una solicitud formada principalmente por dos fuentes de información: un fichero JSON que contiene la representación completa de ProyectoBase y un fichero con el código esqueleto correspondiente a la clase que debe implementarse.

Esta separación permite proporcionar simultáneamente contexto global y contexto local. El JSON permite conocer los elementos existentes en el proyecto, sus identificadores, relaciones y documentación, mientras que el esqueleto delimita exactamente la clase y los métodos que deben ser implementados en la solicitud actual.

La comunicación con el modelo se complementa mediante un conjunto de instrucciones que restringen expresamente su ámbito de actuación. Entre otras condiciones, se exige respetar la estructura recibida, no crear o modificar los elementos definidos previamente, utilizar los identificadores para resolver las referencias, generar código compatible con C# 7.3 y devolver el resultado con un formato estructurado para facilitar su procesamiento posterior.

Una vez recibida la respuesta, esta se interpreta como información JSON y las implementaciones obtenidas se relacionan con los marcadores presentes en el esqueleto. El controlador sustituye cada identificador por el cuerpo correspondiente y procesa también, cuando existen, las librerías o funciones auxiliares devueltas por el modelo. De esta forma se obtiene finalmente una única representación de código en la que la estructura procede del generador determinista y la lógica de los métodos de la generación mediante Inteligencia Artificial.

El uso de identificadores es muy importante en esta fase, ya que evita depender del nombre de los métodos para determinar dónde debe incorporarse cada fragmento generado. El mismo identificador que vincula los elementos dentro del modelo, sirve como mecanismo de trazabilidad.

El proceso completo materializa la separación planteada en la propuesta: el modelo UML determina la estructura, el generador la transforma de manera reproducible en código y la Inteligencia Artificial interviene después, sobre unos puntos delimitados. La implementación técnica busca mantener esta separación durante todo el flujo de generación.

6.- Casos de estudio

En este capítulo se realiza la validación funcional de la herramienta y de la metodología propuesta mediante tres casos de estudio de complejidad creciente: un caso sencillo con una única clase, un caso medio con interacción entre varias clases y un caso complejo que incorpora interfaces, herencia, tipos definidos por el desarrollador y una mayor interconexión entre los elementos del modelo.

Para cada caso se ha creado previamente un modelo UML en Modelio y se ha generado tanto el esqueleto determinista como el código completo mediante Inteligencia Artificial. En cada subapartado se presenta el modelo utilizado, los resultados obtenidos y una breve valoración de su funcionamiento.

Los artefactos asociados a las pruebas se incluyen en la carpeta Pruebas de la entrega, organizada en un directorio por caso. Cada uno contiene la configuración del proyecto, el fichero XMI, una imagen del modelo, los esqueletos generados y el código completo.

La generación del esqueleto es reproducible al tratarse de un proceso determinista, mientras que el código completo puede variar entre ejecuciones debido a la intervención de la Inteligencia Artificial.

6.1.- Caso sencillo

El primer caso de estudio tiene como objetivo comprobar el funcionamiento básico del proceso completo de generación mediante un modelo reducido y fácilmente verificable. Para ello se ha diseñado una única clase denominada Calculadora, encargada de calcular el importe correspondiente al estacionamiento de un vehículo en función del tiempo transcurrido.

El modelo UML define cuatro atributos privados que representan las reglas de tarifas del sistema: el precio de la primera hora, el precio de las horas adicionales, el importe máximo diario y los minutos de cortesía. Además, se incluyen cuatro métodos públicos destinados a calcular el importe total, calcular el coste correspondiente a varios días, obtener el número de horas facturables y validar la duración introducida. Finalmente, se incorpora un constructor encargado de inicializar las tarifas definidas en el modelo.

Este caso permite comprobar de forma aislada las principales etapas de la metodología: interpretación del modelo XMI, recuperación de la estructura y documentación, generación determinista del esqueleto y posterior implementación de los cuerpos de los métodos mediante Inteligencia Artificial. Al tratarse de una única clase y utilizar tipos de datos básicos, se evita introducir complejidad relacionada con asociaciones o dependencias entre diferentes elementos.

El esqueleto generado reproduce correctamente la estructura definida en el modelo, manteniendo los atributos, métodos, parámetros, tipos de retorno, visibilidades y comentarios asociados. Posteriormente, la Inteligencia Artificial completa exclusivamente los cuerpos previamente delimitados. El código resultante conserva la estructura original y utiliza los valores descritos en la documentación para inicializar las tarifas y desarrollar la lógica necesaria para calcular el precio del estacionamiento.

Por ejemplo, el método CalcularImporte aplica inicialmente la validación de la duración y el periodo de cortesía. Para estancias inferiores a un día calcula las horas facturables y aplica las tarifas correspondientes, mientras que para periodos superiores a 24 horas combina el cálculo de los días completos con el correspondiente al tiempo restante. De forma complementaria,

CalcularImporteVariosDias, CalcularHorasFacturables y ValidarDuracion implementan las operaciones auxiliares especificadas en sus respectivas descripciones.

El resultado obtenido demuestra que, para un caso sencillo, la herramienta es capaz de transformar el modelo UML en una estructura C# coherente y utilizar posteriormente la documentación asociada para completar su comportamiento sin modificar los elementos definidos previamente por el desarrollador. El código generado puede incorporarse a un proyecto C# y ejecutarse para comprobar funcionalmente las reglas indicadas en el modelo.

Este primer caso sirve, por tanto, como validación básica del flujo completo de la propuesta. Los siguientes casos aumentan progresivamente la complejidad incorporando interacción entre distintas clases y un mayor número de construcciones UML.

Se puede observar el modelo en el Anexo f.- Diagramas UML Imagen 17 Prueba simple.

6.2.- Caso medio

El segundo caso de estudio incrementa la complejidad respecto al anterior mediante la utilización de varias clases relacionadas entre sí. Para ello se ha modelado un sistema simplificado de gestión de préstamos de una biblioteca, organizado en dos áreas principales: Dominio, que contiene las clases Libro y Usuario, y Servicios, donde se encuentra la clase Biblioteca.

La clase Libro representa el estado básico de un ejemplar mediante su identificador, título y disponibilidad. Sus métodos permiten inicializar sus datos, comprobar su identificador y modificar o consultar su estado de préstamo. La clase Usuario mantiene la información del usuario y el número de préstamos activos, estableciendo un límite máximo de tres préstamos simultáneos y proporcionando operaciones para registrar préstamos y devoluciones.

Por su parte, Biblioteca concentra determinadas reglas generales del servicio, como la duración estándar de los préstamos y la penalización diaria por retraso. También incluye operaciones para validar si un préstamo puede realizarse, calcular penalizaciones, determinar los días restantes y validar identificadores. El modelo incorpora además relaciones entre Biblioteca, Libro y Usuario, permitiendo representar que el servicio gestiona los elementos definidos dentro del dominio.

Este caso permite comprobar que la herramienta es capaz de procesar un modelo compuesto por varias clases ubicadas en distintos espacios de nombres, mantener sus relaciones y generar de forma independiente el código correspondiente a cada una de ellas. A diferencia del caso sencillo, la Inteligencia Artificial recibe aquí información sobre el conjunto completo del proyecto, aunque la implementación se realiza individualmente para cada clase.

El resultado obtenido mantiene correctamente la estructura definida en el modelo y genera implementaciones coherentes con la documentación proporcionada. En Libro, por ejemplo, se respetan las reglas de inicialización y los cambios de disponibilidad; en Usuario se controla el límite de préstamos y la actualización del contador; y en Biblioteca se implementan las reglas de duración y penalización establecidas en la especificación.

Este caso muestra, por tanto, que el proceso puede aplicarse a un modelo con varias clases relacionadas y con responsabilidades diferenciadas, manteniendo la generación estructural bajo un proceso determinista y utilizando la Inteligencia Artificial únicamente para completar la lógica de los métodos. Frente al caso anterior, se valida además la capacidad de trabajar con una estructura de proyecto más próxima a una aplicación real, aunque todavía limitada a relaciones y comportamientos relativamente sencillos.

Se puede observar el modelo en el Anexo f.- Diagramas UML Imagen 18 Prueba media.

6.3.- Caso completo

El tercer caso de estudio tiene como objetivo someter la propuesta a una estructura más próxima a una aplicación real, incorporando un mayor número de elementos UML, relaciones entre clases y tipos definidos dentro del propio modelo. Para ello se ha diseñado un sistema simplificado de gestión de pedidos organizado en diferentes espacios de nombres: clases base, entidades, contratos, enumeraciones y servicios.

El modelo incluye una clase abstracta *Persona*, de la que hereda *Cliente*, una interfaz *ICalculableTotal*, implementada por *Pedido*, una enumeración *EstadoPedido*, y las entidades *Producto*, *LineaPedido* y *Pedido*, coordinadas desde el servicio *GestorPedidos*. De esta forma, el caso incorpora herencia, implementación de interfaces, composición y dependencia entre clases, además de atributos, parámetros y retornos cuyo tipo corresponde a elementos definidos por el propio desarrollador.

La clase *Persona* concentra la información común de identificación y nombre, mientras que *Cliente* amplía este comportamiento incorporando el porcentaje de descuento y el estado activo del cliente. La implementación generada respeta la herencia y reutiliza correctamente el método protegido *InicializarBase* definido en la clase superior.

La gestión de los productos y de las líneas de pedido introduce una interacción más directa entre clases. *LineaPedido* mantiene una referencia a *Producto* y utiliza sus operaciones para comprobar disponibilidad, calcular subtotales y reservar o liberar stock. El código generado hace uso de estas relaciones para implementar el comportamiento descrito en la documentación sin introducir nuevas dependencias estructurales.

Por su parte, *Pedido* mantiene referencias a un *Cliente* y a una *LineaPedido*, implementa la interfaz *ICalculableTotal* y utiliza la enumeración *EstadoPedido* para controlar su ciclo de vida. El método *CalcularTotal* combina el subtotal de la línea con el descuento definido por el cliente, mientras que *CambiarEstado* aplica las transiciones permitidas entre los estados creado, confirmado, completado y cancelado.

Finalmente, *GestorPedidos* actúa como componente de coordinación. Sus métodos crean pedidos y líneas, validan el estado de los elementos relacionados, reservan o liberan stock y realizan los cambios de estado correspondientes. Este comportamiento exige usar información procedente de distintas clases del proyecto y constituye una prueba más exigente del contexto global proporcionado a la IA durante la generación.

Durante este caso se observó una situación relevante relacionada con los tipos definidos por el desarrollador. La enumeración *EstadoPedido* no dispone de una asociación explícita con *Pedido* dentro del modelo UML. Como consecuencia, la generación determinista del esqueleto no puede deducir por sí sola la dependencia necesaria para incorporar automáticamente el espacio de nombres correspondiente. Sin embargo, la documentación de los métodos y atributos identifica expresamente el uso de *EstadoPedido*, permitiendo que durante la fase de generación mediante IA se incorpore la librería necesaria y se obtenga un código final coherente.

Este comportamiento resulta muy útil para la evaluación de la metodología, ya que muestra la diferencia entre la información estrictamente estructural disponible para la generación determinista y la información semántica aportada mediante la documentación. Aunque esta situación pone de manifiesto una limitación de la información estructural disponible para el

generador determinista, el resultado permite comprobar cómo la información semántica proporcionada mediante la documentación puede complementar dicha carencia durante la fase generativa.

En conjunto, el caso completo demuestra que la herramienta puede trabajar con una estructura formada por múltiples clases, herencia, interfaces, relaciones entre entidades y tipos definidos dentro del propio proyecto. El código obtenido conserva la arquitectura establecida en el modelo y utiliza la información semántica disponible para completar la lógica de interacción entre sus componentes, constituyendo el escenario más representativo de la metodología propuesta.

Se puede observar el modelo en el Anexo f.- Diagramas UML Imagen 19 Prueba completa.

7.- Evaluación

En este capítulo se evalúan los resultados obtenidos durante los casos de estudio con el objetivo de analizar el comportamiento de la herramienta y comprobar en qué medida la metodología propuesta cumple los objetivos planteados. Para ello se definen los criterios usados en la evaluación, se presentan de forma comparativa los resultados de los tres casos realizados y se analizan posteriormente los principales aspectos observados, incluyendo las limitaciones asociadas al alcance de las pruebas.

7.1.- Metodología de evaluación

La evaluación se plantea a partir de las pruebas realizadas en el capítulo 6.- Casos de estudio, casos seleccionados con un nivel de complejidad creciente para comprobar el comportamiento de la herramienta y de la metodología propuesta. El objetivo consiste en verificar tanto la correcta generación del código como la persistencia de la estructura definida previamente por el desarrollador a lo largo de las distintas fases del proceso.

Para hacer esta evaluación se establecen varios criterios comunes a los tres casos. Primero, se analiza la estructura generada de manera determinista, comprobando que los elementos definidos en el modelo UML se llevan de manera correcta al esqueleto del código. Para comprobar el correcto funcionamiento de esta funcionalidad se revisan las clases, atributos, métodos, etc.

Después, se evalúa la conservación del esqueleto definido por el usuario durante la generación mediante Inteligencia Artificial. El código obtenido debe mantener las decisiones estructurales establecidas con anterioridad. Se comprueba que no se producen modificaciones sobre las relaciones de los métodos, las herencias, interfaces o se crea una trazabilidad paralela con los métodos propios definidos por la Inteligencia Artificial.

Otro criterio que se ha tenido en cuenta es la coherencia funcional del código generado. Se analiza si las implementaciones producidas mediante IA responden al comportamiento descrito en la documentación asociada a los métodos y si usan correctamente los elementos disponibles en el proyecto. Cuando la lógica requiere la intervención de otras clases, se comprueba que las llamadas y los datos usados sean coherentes con la estructura y las responsabilidades definidas en el diagrama de clases.

Para verificar este comportamiento, los resultados obtenidos se integran en proyectos C# de prueba y se ejecutan diferentes escenarios representativos de las reglas definidas para cada caso. Esta comprobación permite valorar no solo que el código generado pueda integrarse dentro de la estructura prevista, sino también que las operaciones implementadas produzcan los resultados esperados en las situaciones evaluadas.

También se analiza el uso del contexto global del proyecto durante la generación mediante IA. Aunque la implementación se realiza individualmente para cada clase, el modelo recibe una representación estructurada del proyecto completo. En los casos que incluyen varias clases se comprueba si esta información permite utilizar correctamente atributos, métodos, tipos y relaciones definidos fuera del esqueleto concreto que está siendo procesado.

Por último, se registran las incidencias y limitaciones observadas durante las pruebas. Este análisis permite identificar tanto posibles restricciones del prototipo como situaciones en las que la información disponible no resulta suficiente para resolver un determinado aspecto de la

generación. Las incidencias detectadas se analizan posteriormente junto con los resultados obtenidos para valorar su influencia sobre el funcionamiento general de la metodología.

La evaluación realizada tiene un carácter principalmente funcional y cualitativo. Su finalidad es comprobar la viabilidad de la metodología propuesta, el cumplimiento de las restricciones establecidas para la generación y el comportamiento del proceso ante modelos con distintos niveles de complejidad. Los resultados obtenidos se presentan y analizan en los apartados siguientes.

7.2.- Caracterización de los casos evaluados

Los tres casos de estudio utilizados durante la evaluación se han definido con un incremento progresivo de complejidad. Esta selección permite comprobar inicialmente el funcionamiento del proceso sobre una estructura aislada y añadir posteriormente elementos que requieren una mayor interacción entre los componentes del modelo. El objetivo no consiste únicamente en utilizar proyectos de diferente tamaño, sino en incorporar de forma progresiva características relevantes para comprobar el comportamiento de las dos fases de generación.

El caso sencillo representa el escenario de menor complejidad. Está formado por una única clase, Calculadora, que utiliza exclusivamente tipos de datos básicos y no mantiene relaciones con otros elementos del proyecto. El modelo contiene los atributos necesarios para representar las tarifas del sistema y los métodos encargados de realizar los diferentes cálculos asociados al estacionamiento.

Este caso permite evaluar el funcionamiento básico del proceso completo sin introducir dependencias externas a la propia clase. De esta forma, se comprueba la interpretación de los elementos definidos en el modelo, su transformación al dominio interno, la generación determinista del esqueleto y la implementación posterior de la lógica a partir de la documentación. Al no existir relaciones entre clases, cualquier problema detectado puede asociarse directamente al tratamiento de los elementos básicos o a la interpretación del comportamiento descrito.

El caso medio introduce una estructura formada por varias clases y diferentes espacios de nombres. El sistema está compuesto por Libro, Usuario y Biblioteca, distribuidos entre las áreas de dominio y servicios. Además de aumentar el número de elementos procesados, el modelo incorpora relaciones entre las clases y responsabilidades diferenciadas para cada una de ellas.

La incorporación de varias clases permite evaluar aspectos que no aparecen en el caso anterior. En primer lugar, se comprueba que el análisis y la generación mantienen correctamente la organización del proyecto y las relaciones representadas en el modelo. En segundo lugar, la generación mediante Inteligencia Artificial deja de trabajar únicamente con información perteneciente a una clase aislada y dispone de un contexto global que incluye los restantes elementos del proyecto.

Este escenario permite, por tanto, comprobar que la generación individual de cada clase puede mantenerse dentro de un proyecto formado por varios componentes y que la información proporcionada como contexto resulta suficiente para interpretar elementos definidos fuera del esqueleto concreto que se está procesando.

El caso completo constituye el escenario de mayor complejidad utilizado durante la evaluación. El modelo incorpora una clase abstracta, varias clases de dominio y servicio, una interfaz y una enumeración, además de diferentes relaciones entre sus elementos. Se incluyen también tipos

definidos por el propio desarrollador utilizados como atributos, parámetros o retornos, aumentando la dependencia existente entre las distintas partes del proyecto.

Dentro de este caso se encuentran elementos como la clase abstracta Persona y su especialización Cliente, la interfaz ICalculableTotal implementada por Pedido, la enumeración EstadoPedido y las clases Producto, LineaPedido, Pedido y GestorPedidos. La estructura permite evaluar conjuntamente herencia, implementación de interfaces, relaciones entre entidades, utilización de tipos propios e interacción entre clases pertenecientes a diferentes espacios de nombres.

El caso completo permite comprobar especialmente el comportamiento de la metodología cuando la implementación de una clase necesita utilizar información definida en otros elementos del proyecto. La generación mediante Inteligencia Artificial continúa realizándose de forma individual, pero el contexto proporcionado contiene la representación global del modelo, por lo que la lógica generada puede necesitar localizar clases, métodos, tipos y relaciones que no aparecen definidos directamente dentro del esqueleto procesado.

La progresión utilizada puede resumirse de la siguiente forma:

Tabla 6 Progresión de pruebas realizadas

Característica evaluada	Caso sencillo	Caso medio	Caso completo
Varias clases	No	Sí	Sí
Diferentes espacios de nombres	No	Sí	Sí
Relaciones entre elementos	No	Sí	Sí
Interacción entre clases	No	Sí	Sí
Herencia	No	No	Sí
Interfaces	No	No	Sí
Enumeraciones	No	No	Sí
Tipos definidos por el desarrollador	No	No	Sí
Necesidad de contexto global	Baja	Media	Alta

La selección de los tres casos permite aumentar progresivamente la cantidad y variedad de información que debe procesarse sin modificar el flujo general de la metodología. El primer

escenario se centra en comprobar las capacidades fundamentales, el segundo introduce organización e interacción entre diferentes clases y el tercero reúne las principales construcciones soportadas por el prototipo dentro de un único proyecto.

Esta caracterización establece la base para analizar por separado el comportamiento de la generación determinista y de la generación mediante Inteligencia Artificial en los apartados siguientes.

7.3.- Evaluación de la generación determinista

La primera fase analizada corresponde a la generación determinista del esqueleto de código. Su objetivo es transformar la información recuperada del modelo UML en una representación C# manteniendo las decisiones estructurales definidas previamente por el desarrollador. Al tratarse de un proceso basado en reglas, una misma entrada produce siempre la misma estructura de salida, permitiendo comprobar directamente la correspondencia entre el modelo y el código generado.

En el caso sencillo, la generación reproduce correctamente la clase Calculadora y los elementos definidos en el modelo. Se mantienen los cuatro atributos privados, los métodos públicos, sus parámetros, tipos de retorno, visibilidades y la documentación asociada. Al no existir relaciones con otras clases ni tipos definidos dentro del propio proyecto, este escenario permite comprobar de forma aislada el tratamiento de los elementos estructurales básicos.

El resultado del primer caso permite verificar que la información recuperada desde el fichero XMI puede transformarse hasta obtener una clase C# coherente sin necesidad de realizar modificaciones manuales sobre su estructura. Los cuerpos de aquellos métodos destinados a ser completados posteriormente quedan preparados mediante los identificadores correspondientes, manteniendo separada la generación estructural de la futura implementación de la lógica.

En el caso medio, la generación debe trabajar con varias clases y con una organización distribuida en distintos espacios de nombres. Las clases Libro y Usuario se encuentran dentro del área de dominio, mientras que Biblioteca pertenece al espacio correspondiente a servicios. La estructura generada mantiene esta distribución y permite obtener individualmente el esqueleto correspondiente a cada una de las clases.

Este caso incorpora además relaciones entre Biblioteca, Libro y Usuario. Estas relaciones son recuperadas y representadas dentro del dominio interno, permitiendo mantener la información estructural existente entre los elementos. En este escenario concreto las asociaciones no implican la creación automática de atributos de tipo Libro o Usuario, ya que dichos miembros no forman parte de la estructura definida para el caso de estudio. La generación se limita, por tanto, a reproducir los elementos establecidos previamente sin introducir miembros adicionales.

La ausencia de modificaciones automáticas sobre la estructura resulta relevante para el planteamiento de la propuesta. Una relación presente en el modelo no provoca por sí sola la creación de atributos o métodos que no hayan sido definidos por el desarrollador. De esta forma, el generador interpreta la información estructural existente, pero no completa automáticamente decisiones de diseño que no formen parte del modelo utilizado como entrada.

El caso completo permite evaluar un conjunto mayor de construcciones soportadas por el prototipo. La generación debe considerar una clase abstracta, herencia, implementación de

interfaces, enumeraciones, diferentes espacios de nombres y atributos o parámetros cuyos tipos corresponden a elementos definidos dentro del propio proyecto.

La clase Cliente, por ejemplo, mantiene correctamente la herencia respecto a Persona, mientras que Pedido conserva la implementación de la interfaz ICalculableTotal. La enumeración EstadoPedido se genera con sus literales correspondientes y los tipos propios utilizados por las distintas clases se mantienen en atributos, parámetros y retornos cuando han sido definidos de esta forma en el modelo.

También se mantiene la organización del proyecto en los diferentes espacios de nombres establecidos, permitiendo conservar la separación entre clases base, entidades, contratos, enumeraciones y servicios. Este resultado es especialmente relevante porque incrementa el número de elementos que deben resolverse durante la transformación sin modificar el funcionamiento general del generador.

La principal incidencia observada durante esta fase aparece precisamente en el tratamiento de EstadoPedido. La enumeración se encuentra definida dentro del proyecto y se utiliza como tipo dentro de Pedido, pero el modelo no contiene una relación explícita que permita al generador determinar de forma automática la dependencia necesaria entre ambos espacios de nombres. Como consecuencia, el esqueleto determinista no incorpora por sí solo la cláusula using correspondiente.

Esta situación muestra una limitación concreta del mecanismo actual de resolución de dependencias. El generador puede transformar la información estructural disponible, pero no puede deducir una relación que no se encuentre suficientemente representada dentro del modelo usado como entrada. La incidencia no afecta a la generación del tipo ni a su utilización dentro de la firma o de los atributos correspondientes, sino a la incorporación automática del espacio de nombres necesario para resolverlo posteriormente en el código.

En los tres casos evaluados se conserva la estructura definida previamente por el desarrollador. No se introducen clases, atributos, métodos, parámetros o relaciones adicionales como resultado de la generación determinista, y tampoco se modifican los nombres, tipos o visibilidades existentes. El comportamiento del generador se mantiene, por tanto, limitado a transformar la información estructural disponible en el modelo interno hacia su representación correspondiente en C#.

La evaluación permite comprobar además que el incremento de complejidad entre los casos no requiere modificar el flujo de generación. El mismo proceso utilizado para construir el esqueleto de una clase aislada puede aplicarse posteriormente a proyectos formados por varias clases, espacios de nombres, relaciones, herencia, interfaces y tipos propios. Las diferencias se encuentran en la cantidad y variedad de información procesada, pero no en el mecanismo general usado para producir el resultado.

En conjunto, los resultados obtenidos permiten considerar que la fase determinista cumple la función asignada dentro de la metodología para los elementos evaluados. La estructura del código permanece vinculada al modelo UML y se genera mediante un proceso reproducible, mientras que las limitaciones detectadas aparecen cuando una determinada dependencia no se encuentra suficientemente expresada mediante la información estructural disponible. Esta separación resulta necesaria para analizar posteriormente hasta qué punto la fase generativa puede completar la lógica sin modificar las decisiones previamente establecidas.

7.4.- Evaluación de la generación mediante IA

La segunda fase evaluada corresponde a la generación de la lógica mediante Inteligencia Artificial. A diferencia de la generación determinista, esta etapa no pretende construir nuevamente la estructura de las clases, sino completar exclusivamente los cuerpos de los métodos definidos previamente. Para ello, la IA recibe como contexto el esqueleto de la clase que debe implementar y una representación estructurada del proyecto completo, incluyendo la documentación asociada a sus distintos elementos.

La evaluación de esta fase se centra principalmente en comprobar tres aspectos: que las implementaciones obtenidas sean coherentes con la documentación proporcionada, que la IA respete la estructura definida previamente y que sea capaz de usar correctamente la información global del proyecto cuando el comportamiento de una clase depende de otros elementos.

En el caso sencillo, la generación se realiza sobre una única clase, Calculadora, por lo que la mayor parte de la información necesaria se encuentra directamente disponible dentro del propio esqueleto y de su documentación. La IA completa los métodos destinados al cálculo del importe del estacionamiento, utilizando las reglas funcionales descritas para aplicar el periodo de cortesía, las tarifas correspondientes y el tratamiento de estancias superiores a un día.

El resultado mantiene los atributos, métodos, parámetros y tipos definidos previamente, concentrándose la intervención de la IA en los puntos marcados para la implementación. En este escenario no resulta necesario resolver interacciones con otras clases, por lo que la evaluación se centra en comprobar la capacidad del modelo para interpretar correctamente la documentación y transformarla en lógica de programación.

El caso medio introduce una situación diferente al estar formado por las clases Libro, Usuario y Biblioteca. Aunque cada una se genera de manera independiente, la IA dispone de la representación completa del proyecto como contexto. Esto permite comprobar si la generación puede mantenerse delimitada a una clase concreta sin perder la información necesaria sobre los restantes elementos del sistema.

Las implementaciones obtenidas respetan las responsabilidades descritas para cada clase. Libro mantiene las operaciones relacionadas con su identificación y disponibilidad, Usuario incorpora el control del número de préstamos activos y Biblioteca desarrolla las reglas correspondientes a la duración de los préstamos, penalizaciones y validaciones definidas en la documentación.

En este caso se comprueba además que la existencia de información global no provoca que la IA modifique la estructura para introducir elementos no definidos originalmente. Las relaciones presentes en el proyecto forman parte del contexto disponible, pero la generación permanece condicionada por el esqueleto de la clase correspondiente y por los miembros realmente accesibles desde ella. De esta forma, la información global se utiliza como apoyo para interpretar el proyecto sin convertir la fase generativa en un mecanismo encargado de rediseñar su estructura.

El caso completo supone la prueba más exigente de esta fase debido al mayor número de elementos y relaciones existentes. La implementación de determinadas operaciones necesita interpretar información procedente de diferentes clases, como Cliente, Producto, LineaPedido, Pedido o GestorPedidos, además de considerar la herencia, la interfaz ICalculableTotal y la enumeración EstadoPedido.

Los resultados muestran que la información global proporcionada permite utilizar elementos definidos fuera de la clase concreta que está siendo procesada. Por ejemplo, las implementaciones relacionadas con pedidos utilizan las operaciones disponibles en otras entidades cuando la documentación describe una interacción entre ellas. De igual forma, la lógica generada mantiene las relaciones ya establecidas y utiliza los tipos definidos dentro del propio proyecto sin sustituirlos por estructuras diferentes.

Este comportamiento resulta relevante porque la generación sigue realizándose por clases individuales. El modelo no recibe únicamente el fichero correspondiente a la clase que debe completar, sino también información estructurada sobre el resto del proyecto. La evaluación de los casos medio y completo permite comprobar que esta estrategia aporta el contexto necesario para las interacciones presentes en los escenarios utilizados sin necesidad de solicitar la generación completa de la aplicación en una única operación.

Durante el caso completo se observa también el comportamiento ya identificado con EstadoPedido. El esqueleto determinista no incorpora automáticamente el espacio de nombres correspondiente debido a que la dependencia no puede deducirse a partir de una relación explícita del modelo. Sin embargo, la documentación disponible identifica la utilización de esta enumeración y la fase generativa incorpora la librería necesaria para que el código final pueda resolver correctamente el tipo.

Esta incidencia permite observar una función adicional de la información semántica dentro del proceso. La documentación no sustituye a la estructura UML ni modifica las decisiones establecidas por el desarrollador, pero puede aportar información que no se encuentra representada explícitamente mediante las relaciones estructurales disponibles. En este caso concreto, la fase generativa utiliza dicha información para completar un detalle técnico necesario para la integración del código.

Otro aspecto evaluado es el respeto de las restricciones establecidas para la IA. En las generaciones analizadas no se han observado modificaciones sobre los nombres de las clases, firmas de los métodos, parámetros, tipos de retorno, visibilidades, relaciones, herencias o interfaces definidas previamente. Los cuerpos generados se integran en los puntos delimitados mediante identificadores, manteniendo diferenciada la procedencia de la estructura y de la lógica.

El uso de identificadores resulta relevante en esta fase, ya que permite asociar cada fragmento generado con el método correspondiente sin depender exclusivamente de su nombre. La respuesta de la IA se procesa de forma estructurada y cada implementación se integra en el punto previamente establecido durante la generación determinista. De esta forma, la incorporación de la lógica no requiere que el modelo devuelva nuevamente la clase completa ni que la aplicación tenga que interpretar qué partes de la estructura deben conservarse.

La evaluación funcional realizada sobre los casos permite comprobar además que las implementaciones obtenidas responden al comportamiento descrito en la documentación utilizada como entrada. Esto no implica que cualquier generación futura produzca exactamente el mismo código, ya que esta fase mantiene un carácter no determinista, pero sí permite verificar que los resultados obtenidos durante las pruebas son compatibles con las reglas definidas para los escenarios evaluados.

En conjunto, los resultados muestran que la generación mediante Inteligencia Artificial puede actuar sobre una estructura previamente delimitada y completar la lógica de los métodos sin

asumir las principales decisiones estructurales del proyecto. Los casos de estudio permiten observar además que el contexto global resulta útil cuando aumenta la interacción entre clases y que la documentación desempeña un papel fundamental para describir el comportamiento que no puede deducirse únicamente a partir de la estructura del modelo.

7.5.- Comparativa y análisis de los resultados

Una vez analizadas de forma independiente la generación determinista y la generación mediante Inteligencia Artificial, resulta posible comparar conjuntamente el comportamiento de la metodología en los tres casos de estudio. Esta comparación permite observar cómo responde el proceso a medida que aumenta la complejidad del modelo y se incorporan nuevas construcciones UML e interacciones entre clases.

Los resultados principales obtenidos pueden resumirse mediante la siguiente tabla:

Tabla 7 Comparativa de pruebas

Criterio	Caso sencillo	Caso medio	Caso completo
Generación correcta de clases y miembros	Sí	Sí	Sí
Conservación de firmas y tipos	Sí	Sí	Sí
Varias clases y espacios de nombres	No aplica	Sí	Sí
Relaciones entre elementos	No aplica	Sí	Sí
Herencia	No aplica	No aplica	Sí
Interfaces	No aplica	No aplica	Sí
Tipos definidos por el desarrollador	No aplica	No aplica	Sí
Implementación coherente con la documentación	Sí	Sí	Sí
Conservación de la estructura durante la generación con IA	Sí	Sí	Sí
Incidencias relevantes	Ninguna	Ninguna	Dependencia de EstadoPedido no resuelta de manera determinista

La comparación muestra que el mismo flujo general de generación puede mantenerse en los tres escenarios evaluados. El incremento progresivo de complejidad no requiere modificar la

secuencia formada por interpretación del modelo, construcción del dominio interno, generación determinista del esqueleto y generación posterior de la lógica mediante Inteligencia Artificial.

En el caso sencillo, la evaluación se limita a los elementos fundamentales de una clase aislada. La ausencia de relaciones, herencia, interfaces o tipos propios permite comprobar de forma directa que los elementos básicos definidos en el modelo se trasladan correctamente al código y que la IA puede completar posteriormente el comportamiento descrito sin modificar dicha estructura.

El caso medio amplía esta situación mediante la incorporación de varias clases, diferentes espacios de nombres y relaciones entre elementos. Los resultados se mantienen respecto al caso anterior, pero se añade la necesidad de conservar la organización global del proyecto y proporcionar a la IA información suficiente sobre elementos distintos de la clase que se está procesando. En las pruebas realizadas, esta ampliación no introduce incidencias relevantes.

El caso completo incorpora el mayor número de construcciones evaluadas. La presencia de herencia, interfaces, enumeraciones, tipos definidos por el desarrollador y relaciones entre diferentes entidades permite comprobar el comportamiento del proceso en un escenario con un grado superior de interdependencia. A pesar de este incremento de complejidad, se conserva la estructura definida en el modelo y las implementaciones generadas utilizan los elementos disponibles dentro del proyecto de acuerdo con la documentación proporcionada.

Uno de los resultados más importantes de la comparación es la conservación de la estructura durante la generación mediante IA. Este criterio se cumple en los tres casos evaluados. Las clases, métodos, parámetros, tipos de retorno, visibilidades y relaciones permanecen determinados por el modelo y por el esqueleto generado previamente, mientras que la Inteligencia Artificial se limita a completar los puntos destinados a la implementación de la lógica.

Este comportamiento permite mantener una separación clara entre las dos fases de generación. La estructura puede reproducirse mediante las reglas definidas en el generador, mientras que los cuerpos de los métodos proceden de una fase no determinista. De esta forma, una variación en la implementación generada por la IA no implica necesariamente una modificación de la organización estructural establecida previamente por el desarrollador.

La comparación también permite observar la importancia creciente del contexto global del proyecto. En el caso sencillo prácticamente toda la información necesaria se encuentra dentro de la propia clase. En el caso medio comienza a ser necesario interpretar la existencia y responsabilidad de otros elementos, mientras que en el caso completo determinadas implementaciones dependen directamente de clases, métodos, relaciones y tipos definidos en otras partes del proyecto.

Los resultados obtenidos en los casos medio y completo indican que la estrategia de proporcionar el proyecto completo como contexto, manteniendo al mismo tiempo una generación individual por clase, permite resolver las interacciones presentes en los escenarios evaluados. De esta manera se evita solicitar la generación completa de la aplicación en una única operación, pero la IA continúa disponiendo de información sobre el resto del sistema cuando necesita interpretar una dependencia o utilizar un elemento externo a la clase actual.

Otro aspecto común a los tres casos es la coherencia entre la implementación generada y la documentación proporcionada. En las pruebas realizadas, los cuerpos obtenidos responden a

las responsabilidades y reglas descritas para los diferentes métodos. Este resultado refuerza el papel de la documentación dentro de la metodología, ya que constituye la principal fuente de información para expresar el comportamiento que no puede obtenerse directamente a partir de la estructura UML.

La principal diferencia entre los resultados aparece en el caso completo mediante la dependencia asociada a EstadoPedido. Mientras que el resto de los elementos analizados pueden resolverse directamente a partir de la información estructural disponible, en este caso el generador determinista no dispone de una relación explícita que permita incorporar automáticamente el espacio de nombres correspondiente.

Esta incidencia no impide obtener el código final, ya que la documentación proporciona información suficiente para que la fase generativa incorpore la librería necesaria. Sin embargo, sí pone de manifiesto que la capacidad del generador determinista está directamente condicionada por la información representada en el modelo. Cuando una dependencia necesaria no puede deducirse estructuralmente, el mecanismo actual no dispone de información suficiente para resolverla de manera automática.

Al mismo tiempo, esta situación muestra que la información estructural y la información semántica desempeñan funciones complementarias. La primera permite establecer de forma controlada la organización del software, mientras que la segunda puede describir aspectos funcionales y determinados detalles de implementación que no se encuentran representados mediante las relaciones del modelo.

En conjunto, la comparación de los tres casos muestra un comportamiento estable del proceso dentro del alcance evaluado. El aumento de complejidad amplía la cantidad de información que debe procesarse y aumenta la importancia del contexto global, pero no modifica la separación entre la generación determinista de la estructura y la generación mediante Inteligencia Artificial de la lógica. La única incidencia relevante detectada se encuentra relacionada con una dependencia no representada de forma explícita en el modelo, aspecto que permite además identificar una de las limitaciones que deben tenerse en cuenta al interpretar los resultados.

7.6.- Limitaciones de la evaluación

La evaluación realizada permite comprobar el funcionamiento de la metodología propuesta dentro de los escenarios definidos. Los resultados obtenidos deben interpretarse teniendo en cuenta una serie de limitaciones asociadas al alcance del prototipo y al diseño de las pruebas realizadas.

1. Número limitado de casos de estudio. La validación se ha realizado sobre tres casos diseñados específicamente para este trabajo. Esta selección permite comprobar distintos aspectos de la metodología, pero no resulta suficiente para generalizar los resultados a cualquier tipo de proyecto software.
2. Dependencia del formato de entrada. El prototipo se ha evaluado únicamente con diagramas de clases exportados desde Modelio mediante XMI. La herramienta interpreta un subconjunto concreto de los elementos presentes en estos ficheros, por lo que no se ha evaluado su comportamiento con otras herramientas de modelado, otras versiones de XMI o construcciones UML no contempladas.
3. Único lenguaje de salida. La generación se ha validado únicamente para C#. Aunque la arquitectura de la aplicación permite incorporar otros generadores, no se han implementado ni probado otros lenguajes.

4. Único proveedor de IA. La integración desarrollada usa OpenAI como proveedor de IA. La solución contempla la posibilidad de incorporar otras implementaciones, pero no se ha evaluado si otros proveedores o modelos presentarían un comportamiento equivalente dentro de la metodología propuesta.
5. Carácter no determinista de la generación mediante IA. A diferencia del esqueleto, que puede reproducirse de manera estable, la implementación generada mediante IA puede variar entre ejecuciones. La evaluación analiza los resultados obtenidos en las pruebas realizadas, pero no incluye un estudio repetido o estadístico sobre múltiples generaciones del mismo caso.
6. Dependencia de la calidad de la documentación. Los casos de estudio usados contienen descripciones suficientemente detalladas para expresar el comportamiento esperado. Por tanto, no se ha evaluado de forma específica cómo afecta al resultado una documentación incompleta o contradictoria.
7. Evaluación principalmente funcional y cualitativa. La validación se centra en comprobar la conservación de la estructura, la coherencia con la documentación y el comportamiento esperado en los casos seleccionados. No se han incorporado métricas cuantitativas sobre productividad, coste, tiempo de desarrollo, mantenibilidad a largo plazo o calidad estática del código generado.

En consecuencia, la evaluación realizada proporciona evidencia sobre la viabilidad de la metodología dentro de los escenarios analizados, pero no pretende demostrar su validez universal. Una evaluación más amplia requeriría incorporar un mayor número de proyectos, distintas herramientas y formatos de modelado, otros lenguajes y proveedores de Inteligencia Artificial, así como pruebas repetidas sobre los mismos casos, diferentes niveles de calidad en la documentación proporcionada y métricas cuantitativas adicionales.

8.- Discusión

La evaluación realizada permite analizar la propuesta más allá de la comprobación funcional del prototipo. Los resultados obtenidos muestran cómo se comporta la separación entre generación determinista y generación mediante Inteligencia Artificial en los escenarios estudiados, pero también permiten valorar las implicaciones que esta división tiene desde el punto de vista de la Ingeniería del Software. En este capítulo se interpretan los resultados en relación con los objetivos y la hipótesis planteados, se compara la aproximación con otros niveles de automatización y se analizan sus principales ventajas, inconvenientes y compromisos.

8.1.- Cumplimiento de objetivos

El objetivo principal del trabajo era estudiar si las técnicas dirigidas por modelos podían combinarse con la generación mediante Inteligencia Artificial manteniendo bajo control del desarrollador las principales decisiones relacionadas con la estructura del software. Para ello no resultaba suficiente demostrar que un modelo de lenguaje podía generar código, sino que era necesario definir un proceso en el que la intervención de la IA quedase delimitada y pudiera integrarse con artefactos obtenidos de forma determinista.

El prototipo desarrollado permite materializar este planteamiento mediante un flujo que comienza con la interpretación de un modelo UML exportado en formato XMI. La información obtenida se transforma en un dominio interno sobre el que pueden revisarse y completar las descripciones de los distintos elementos. A partir de esta representación se genera de forma determinista el esqueleto de código C#, manteniendo las clases, interfaces, enumeraciones, atributos, métodos, parámetros, tipos, visibilidades y relaciones definidas previamente.

Una vez construida esta estructura, la Inteligencia Artificial interviene únicamente sobre los puntos preparados para completar la implementación. La solicitud incorpora tanto el esqueleto de la clase concreta como una representación estructurada del proyecto completo, de forma que el modelo dispone de contexto global sin recibir libertad para reorganizar la solución. Los identificadores internos permiten relacionar las implementaciones devueltas con los métodos correspondientes y facilitan la integración posterior de la respuesta en el código generado.

Los tres casos de estudio muestran que este flujo puede aplicarse sin modificar su funcionamiento general cuando aumenta progresivamente la complejidad del modelo. El caso sencillo permite comprobar la generación sobre una clase aislada, el caso medio incorpora varias clases y espacios de nombres, y el caso completo añade herencia, interfaces, enumeraciones, tipos definidos dentro del propio proyecto y una mayor interacción entre los distintos componentes.

Dentro de este alcance puede considerarse cumplido el objetivo de separar las responsabilidades de ambas fases de generación. La estructura permanece vinculada al modelo y al generador determinista, mientras que la IA se utiliza para interpretar la información semántica y completar la lógica de los métodos. No obstante, este cumplimiento debe entenderse dentro de las condiciones evaluadas: el prototipo trabaja con el subconjunto XMI soportado, genera C# y utiliza un único proveedor de Inteligencia Artificial.

También se cumple el objetivo de mantener la documentación como un elemento activo del proceso. Las descripciones no se incorporan únicamente al código como comentarios, sino que forman parte de la información utilizada para expresar el comportamiento esperado. Esto permite que la misma documentación desempeñe una función durante la generación y permanezca posteriormente asociada al software obtenido.

8.2.- Validación de la hipótesis

La hipótesis de trabajo planteaba que la combinación de generación determinista y generación mediante Inteligencia Artificial permitiría aumentar el grado de automatización de la implementación sin delegar en la IA las principales decisiones estructurales y arquitectónicas del sistema. Los resultados obtenidos apoyan esta hipótesis para los escenarios analizados.

En los tres casos de estudio la estructura utilizada por la fase generativa procede del modelo UML y del esqueleto construido previamente. Durante las generaciones evaluadas no se observaron modificaciones sobre los nombres de las clases, firmas de los métodos, parámetros, tipos de retorno, visibilidades, herencias o interfaces definidos por el desarrollador. La intervención del modelo se concentra en los cuerpos de los métodos y, cuando resulta necesario, en elementos técnicos delimitados como las librerías o funciones auxiliares previstas por el proceso.

Esta separación permite aumentar la automatización respecto a una generación dirigida por modelos centrada únicamente en estructuras. La lógica que normalmente requeriría una implementación manual puede producirse a partir de la documentación proporcionada, sin necesidad de definir previamente una transformación específica para cada comportamiento. Al mismo tiempo, la generación de dicha lógica se realiza sobre una estructura que no ha sido propuesta libremente por el modelo de lenguaje.

El resultado debe interpretarse con cautela. La evaluación no permite afirmar que la metodología garantice una implementación correcta para cualquier proyecto ni que una estructura previamente definida elimine los posibles errores de la IA. La fase generativa continúa siendo no determinista y depende de la calidad del contexto y de la documentación disponible. Por tanto, la validación de la hipótesis se refiere principalmente a la posibilidad de mantener una separación efectiva entre la estructura controlada y la lógica generada, no a garantizar automáticamente la corrección funcional de cualquier implementación obtenida.

La incidencia detectada con EstadoPedido refuerza además esta interpretación. El generador determinista no puede resolver una dependencia que no se encuentra suficientemente representada mediante la información estructural disponible, mientras que la documentación permite que la fase generativa complete el espacio de nombres necesario. Este comportamiento muestra que ambas fuentes de información cumplen funciones distintas y que la incorporación de Inteligencia Artificial puede complementar determinados aspectos que no se deducen directamente del modelo, sin que ello implique sustituir el papel de la estructura.

8.3.- Comparación con otras aproximaciones de generación

La propuesta puede situarse entre dos formas de automatización con características diferentes. En un extremo se encuentran las aproximaciones tradicionales dirigidas por modelos, donde el resultado depende de reglas y transformaciones definidas previamente. En el otro se encuentran soluciones basadas en modelos de lenguaje y agentes con un mayor grado de autonomía, capaces de interpretar objetivos de alto nivel y actuar sobre diferentes elementos de un proyecto.

La generación tradicional basada en modelos presenta como principal ventaja la previsibilidad de las transformaciones. Cuando una construcción está soportada, la relación entre la entrada y la salida puede definirse de manera explícita y reproducible. Esta característica resulta especialmente adecuada para elementos estructurales y repetitivos, pero aumenta el coste de

automatizar comportamientos específicos cuando cada nueva situación exige incorporar nuevas reglas.

Las aproximaciones basadas directamente en modelos de lenguaje reducen esa dependencia de transformaciones específicas. Una descripción funcional puede utilizarse para obtener código sin formalizar previamente todas las reglas necesarias para producirlo. Los agentes amplían esta capacidad al permitir que la propia IA inspeccione archivos, realice modificaciones, ejecute herramientas y utilice los resultados obtenidos para decidir nuevas acciones.

La metodología propuesta adopta una posición intermedia. No intenta sustituir la generación determinista por un modelo de lenguaje, sino asignar a cada mecanismo la tarea para la que se considera más adecuado dentro del planteamiento del trabajo. La estructura se obtiene mediante reglas conocidas y la lógica se delega a la IA una vez que los límites de actuación han sido definidos.

Tabla 8 Delegación de responsabilidades

Aspecto	MDA tradicional	Propuesta híbrida	IA con mayor autonomía
Definición de la estructura	Modelo y transformaciones	Modelo y generación determinista	Puede ser propuesta o modificada por la IA
Generación de lógica específica	Limitada a reglas previamente formalizadas	Generada mediante IA sobre métodos definidos	Generada mediante IA
Grado de autonomía de la IA	No aplica	Restringido	Elevado o variable
Control sobre clases y firmas	Alto	Alto	Depende de las restricciones aplicadas
Necesidad de modelo previo	Fundamental	Fundamental	No necesariamente
Papel de la documentación	Descriptivo o complementario	Especificación semántica para la generación	Parte del contexto disponible
Reproducibilidad	Alta en transformaciones definidas	Alta en estructura y variable en lógica	Variable
Intervención del desarrollador	Diseño y lógica no automatizada	Diseño, documentación y revisión	Definición del objetivo y supervisión

La comparación pone de manifiesto que la propuesta no busca alcanzar el máximo nivel de autonomía posible. El trabajo previo de modelado y documentación se mantiene de forma deliberada porque constituye el mecanismo mediante el que el desarrollador conserva la capacidad de decidir la estructura y proporcionar la información funcional que debe respetarse durante la generación.

Esta decisión implica un compromiso. Una solución más autónoma puede reducir el trabajo inicial necesario para obtener una implementación, mientras que la metodología propuesta exige disponer de un modelo y revisar su documentación antes de generar código. A cambio, la

procedencia de la estructura permanece explícita y no depende de decisiones adoptadas durante una generación no determinista.

Por tanto, la propuesta no debe considerarse una sustitución de las herramientas basadas en agentes ni una evolución destinada necesariamente a superarlas. Representa una alternativa orientada a escenarios en los que el desarrollador desea utilizar capacidades generativas para automatizar la implementación, pero considera necesario mantener un control previo sobre la organización del software.

8.4.- Papel de la documentación en la metodología

Uno de los aspectos que adquiere mayor relevancia dentro de la propuesta es el papel asignado a la documentación. En un proceso convencional, los comentarios pueden considerarse principalmente un mecanismo destinado a facilitar la comprensión del código una vez implementado. En la metodología desarrollada, además de conservar esta función, pasan a formar parte de la especificación utilizada durante la generación.

El modelo UML permite establecer qué elementos forman el sistema y cómo se relacionan, pero la firma de un método no determina por sí sola el comportamiento que debe realizar. Dos métodos con el mismo conjunto de parámetros y el mismo tipo de retorno pueden implementar reglas completamente diferentes. La documentación aporta la información semántica necesaria para describir esa intención sin trasladarla a una transformación determinista específica.

Esta utilización modifica también el momento en el que la documentación adquiere importancia. En lugar de incorporarse únicamente después de implementar el software, debe estar suficientemente definida antes de solicitar la generación mediante IA. Como consecuencia, parte del esfuerzo tradicionalmente dedicado a escribir código se desplaza hacia la definición del comportamiento esperado.

Este desplazamiento no elimina el trabajo del desarrollador, sino que cambia su naturaleza. La precisión de las descripciones condiciona el contexto disponible para el modelo y, por tanto, una documentación ambigua, incompleta o contradictoria puede producir resultados igualmente ambiguos o incorrectos. La metodología depende de que el desarrollador sea capaz de expresar con claridad las responsabilidades y reglas que desea implementar.

Al mismo tiempo, la documentación permanece asociada al código resultante. Esto evita que la información utilizada para guiar la generación desaparezca una vez completada la implementación y permite conservar una relación entre la intención funcional expresada y el código producido. En el prototipo, esta información puede utilizarse además para generar documentación técnica mediante herramientas como Doxygen.

La incidencia relacionada con EstadoPedido muestra de forma práctica el carácter complementario de esta información. La estructura disponible no permite resolver por sí sola la dependencia necesaria, pero la documentación proporciona una referencia semántica que permite completar el detalle técnico correspondiente durante la fase generativa. Este resultado no significa que la documentación deba sustituir a relaciones estructurales que deberían encontrarse correctamente modeladas, sino que puede aportar contexto adicional cuando existe información que no ha quedado expresada de forma suficiente en el modelo.

8.5.- Ventajas de la propuesta

La principal ventaja de la propuesta es la posibilidad de mantener una separación explícita entre las decisiones estructurales y la implementación generada mediante Inteligencia Artificial. Las

clases, interfaces, enumeraciones, atributos, métodos, parámetros, tipos y relaciones se definen antes de la intervención del modelo de lenguaje, por lo que el desarrollador puede determinar previamente la organización que desea obtener.

Esta separación aporta también trazabilidad al proceso de generación. La estructura del código puede relacionarse con los elementos recuperados del modelo y los identificadores permiten asociar cada cuerpo generado con el método correspondiente. De esta manera, la integración de la respuesta no depende de reconstruir o comparar de nuevo la clase completa.

Otra ventaja es la utilización conjunta de contexto global y generación local. La IA recibe información sobre el proyecto completo, pero la unidad de trabajo permanece limitada a una clase concreta. Esta decisión permite proporcionar información sobre otros tipos, métodos y relaciones sin conceder al modelo la responsabilidad de generar o reorganizar todo el proyecto en una única operación.

La documentación representa igualmente una ventaja desde el punto de vista de la comprensión. La información necesaria para guiar la generación permanece asociada a los elementos resultantes, por lo que puede seguir siendo utilizada por el desarrollador después de obtener el código. La metodología favorece así que la automatización no elimine la necesidad de explicar las responsabilidades de los componentes.

Por último, la arquitectura del prototipo mantiene separados el formato de entrada, el lenguaje de salida y el proveedor de Inteligencia Artificial mediante abstracciones específicas. Aunque la evaluación se ha realizado únicamente con Modelio, C# y OpenAI, esta organización facilita plantear futuras extensiones sin vincular conceptualmente la metodología a una única combinación tecnológica.

8.6.- Inconvenientes y compromisos

La principal contrapartida de la propuesta es el trabajo previo necesario para iniciar la generación. El desarrollador debe construir el modelo UML, comprobar que contiene la estructura deseada, revisar la documentación y completar aquellas descripciones necesarias para expresar el comportamiento. Este proceso puede resultar más costoso inicialmente que utilizar una herramienta en la que una especificación textual de alto nivel sea suficiente para comenzar a modificar el proyecto.

La metodología exige además conocimientos de modelado y diseño software. El usuario debe decidir previamente las clases, responsabilidades, firmas, relaciones y demás elementos estructurales. Por tanto, la automatización de la implementación no elimina la necesidad de realizar actividades de Ingeniería del Software y puede presentar una barrera de entrada superior a la de herramientas orientadas exclusivamente a instrucciones en lenguaje natural.

Otro inconveniente es la dependencia de la calidad de la información proporcionada. El generador determinista solo puede trabajar con los elementos que el prototipo sea capaz de interpretar, mientras que la generación mediante IA depende de que el contexto y la documentación describan suficientemente el comportamiento esperado. Una carencia en cualquiera de estas dos fuentes puede trasladarse al resultado final.

También continúa siendo necesaria la revisión del código generado. Restringir la actuación del modelo reduce el espacio sobre el que puede introducir cambios, pero no convierte la lógica generada en determinista ni garantiza su corrección. El desarrollador sigue siendo responsable

de comprobar que la implementación satisface las reglas funcionales y que no introduce errores, problemas de seguridad o comportamientos no previstos.

El prototipo actual presenta además limitaciones técnicas que reducen el alcance de la generación. Entre ellas se encuentra el tratamiento de un subconjunto concreto de XMI, la existencia de un único generador C# y un único proveedor de IA. La multiplicidad de las relaciones se conserva como información del dominio, pero actualmente no se utiliza para determinar automáticamente estructuras de colección en el código, por lo que no se generan listas o vectores a partir de relaciones con multiplicidades superiores a una referencia individual.

Finalmente, la elección de una generación individual por clase supone otro compromiso entre control y automatización. Esta estrategia facilita delimitar cada solicitud y revisar el resultado, pero requiere repetir el proceso para las diferentes clases del proyecto. Una generación masiva podría simplificar la interacción del usuario, aunque debería mantener internamente las restricciones necesarias para evitar que el aumento de automatización implique una pérdida de control sobre la estructura.

En conjunto, la metodología no elimina las dificultades asociadas a la generación automática de software, sino que redistribuye responsabilidades. Se reduce la necesidad de implementar manualmente parte de la lógica, pero se mantiene el esfuerzo de modelado, especificación, documentación y validación. El valor de la propuesta se encuentra precisamente en aceptar este compromiso y priorizar el control estructural sobre una autonomía completa del proceso generativo.

9.- Conclusiones

9.1.- Conclusiones

El presente Trabajo de Fin de Máster ha estudiado la posibilidad de combinar mecanismos propios del desarrollo dirigido por modelos con las capacidades actuales de generación de código mediante Inteligencia Artificial. El punto de partida ha sido la existencia de dos aproximaciones con características complementarias: las transformaciones deterministas permiten mantener un control sobre los elementos que pueden formalizarse, mientras que los modelos de lenguaje ofrecen una mayor flexibilidad para interpretar información semántica y producir lógica específica.

A partir de esta idea se ha definido una metodología híbrida en la que la estructura del software se determina antes de la intervención de la IA. El desarrollador construye un modelo UML, completa la documentación necesaria y usa esta información para obtener un esqueleto de código mediante un proceso determinista. Posteriormente, la IA recibe dicho esqueleto y el contexto global del proyecto para implementar los cuerpos de los métodos dentro de los límites establecidos.

El prototipo desarrollado demuestra que esta separación puede ser viable técnicamente. La herramienta interpreta diagramas de clases UML exportados mediante XMI, transforma la información hacia un dominio interno, permite revisar la documentación, genera esqueletos C# e integra posteriormente los cuerpos producidos mediante Inteligencia Artificial. El uso de identificadores únicos permite mantener la correspondencia entre los elementos del modelo y los fragmentos de código generado automáticamente.

Los tres casos de estudio muestran que el mismo flujo puede aplicarse a escenarios con diferentes niveles de complejidad. La evaluación comienza con una clase aislada y continúa con proyectos que incorporan varias clases, espacios de nombres, relaciones, herencia, interfaces, enumeraciones y tipos definidos por el desarrollador. Dentro de estos escenarios, la estructura permanece determinada por el modelo y el generador, mientras que la IA completa la lógica descrita mediante la documentación.

Los resultados apoyan, por tanto, la hipótesis planteada dentro del alcance evaluado. Es posible aumentar la automatización de la implementación usando un modelo de lenguaje sin concederle capacidad para modificar las decisiones estructurales del proyecto. Esta conclusión no implica que el código generado sea correcto ni que la metodología pueda generalizarse a cualquier aplicación, sino que la separación entre estructura determinista y lógica generativa resulta viable.

Una conclusión especialmente relevante es el papel que puede desempeñar la documentación cuando se integra dentro del propio proceso de generación. Las descripciones dejan de ser únicamente información destinada al mantenimiento posterior y pasan a expresar parte del comportamiento que debe interpretar la IA. El modelo aporta la estructura y la documentación complementa esta información con una especificación semántica, permitiendo utilizar conjuntamente dos niveles distintos de descripción del software.

La metodología propuesta no persigue maximizar la autonomía de la Inteligencia Artificial. Frente a aproximaciones en las que un agente puede explorar y modificar libremente diferentes partes de un proyecto, se ha optado por restringir deliberadamente su ámbito de actuación. Esta decisión implica mantener actividades previas de modelado y documentación, pero permite

conservar de forma más explícita la relación entre las decisiones del desarrollador y la estructura finalmente obtenida.

En consecuencia, el trabajo muestra que la incorporación de IA a procesos dirigidos por modelos no tiene por qué plantearse como una sustitución de las transformaciones tradicionales. Ambos mecanismos pueden utilizarse de forma complementaria, reservando la generación determinista para la información formalizada y utilizando la capacidad generativa en aquellas partes donde la expresión mediante reglas específicas resultaría más costosa.

9.2.- Contribuciones del trabajo

La principal contribución del trabajo es la definición y materialización de una metodología que establece una separación explícita entre la generación determinista de la estructura y la generación mediante Inteligencia Artificial de la lógica. Esta división constituye el elemento central de la propuesta y determina tanto el diseño de la herramienta como la forma en que se prepara el contexto y se integra posteriormente el código.

Una segunda contribución es la utilización de un modelo UML enriquecido mediante documentación como especificación de entrada. La estructura del modelo establece los elementos y relaciones principales del sistema, mientras que las descripciones incorporan información sobre responsabilidades y comportamiento. La combinación de ambos tipos de información permite utilizar una representación formal para aquello que puede modelarse estructuralmente y lenguaje natural para expresar aspectos funcionales.

También constituye una aportación el mecanismo utilizado para combinar contexto global y generación por clase. La representación JSON del proyecto proporciona al modelo información sobre los restantes elementos del sistema, mientras que el esqueleto delimita la unidad concreta que debe implementarse. Esta estrategia intenta evitar tanto el aislamiento de una generación basada únicamente en una clase como la delegación del proyecto completo en una única solicitud.

El uso de identificadores internos representa otra contribución práctica de la implementación. Los marcadores incorporados durante la generación determinista permiten relacionar cada cuerpo recibido con el método exacto al que pertenece, evitando depender exclusivamente de nombres o de la comparación textual del código. El mismo mecanismo contribuye a mantener la trazabilidad entre el dominio interno, el esqueleto y el resultado generado.

Desde el punto de vista de la Ingeniería del Software, el trabajo aporta además una forma concreta de limitar la autonomía de un modelo de lenguaje durante la implementación. Las restricciones incluidas en el proceso establecen qué elementos pueden modificarse y cuáles deben permanecer invariables, permitiendo utilizar la IA como mecanismo de implementación sobre decisiones adoptadas previamente.

Finalmente, se aporta un prototipo funcional que materializa la metodología y permite evaluarla mediante casos de estudio. La herramienta no constituye únicamente una demostración aislada de generación de código, sino que integra importación del modelo, representación de dominio, revisión de documentación, persistencia, generación determinista, comunicación con un proveedor de IA e integración del resultado dentro de un mismo flujo.

9.3.- Líneas futuras de trabajo

A partir del prototipo desarrollado pueden plantearse distintas líneas de trabajo destinadas tanto a ampliar las capacidades de la herramienta como a profundizar en la evaluación de la metodología propuesta.

Una primera evolución consistiría en permitir que la propia aplicación pueda crear y modificar el modelo estructural, en lugar de limitarse a importar un diagrama previamente elaborado en una herramienta externa. Esto permitiría añadir o eliminar clases, atributos, métodos, parámetros, interfaces, enumeraciones y relaciones directamente desde la aplicación, convirtiéndola en un entorno más completo para definir y enriquecer el sistema antes de iniciar la generación.

Relacionado con lo anterior, también podría incorporarse la generación asistida de modelos UML a partir de especificaciones textuales. La Inteligencia Artificial podría proponer una primera estructura de clases y relaciones que posteriormente fuese revisada y validada por el desarrollador. Una vez aceptada, dicha estructura pasaría a actuar como referencia determinista para las siguientes fases del proceso, manteniendo la intervención humana antes de generar el código.

Otra línea de trabajo consiste en incorporar una generación masiva del proyecto completo. Actualmente el proceso se realiza sobre clases concretas, pero una evolución natural permitiría seleccionar una ubicación de destino y generar automáticamente todos los elementos del modelo mediante una única acción. La herramienta podría reproducir la estructura de directorios y espacios de nombres, crear los ficheros correspondientes y mantener internamente la generación individual cuando resulte conveniente para controlar el contexto y las solicitudes realizadas a la IA.

También resulta de interés ampliar los modelos y formatos de entrada soportados. El prototipo actual se centra en diagramas de clases UML exportados mediante el formato XMI utilizado por Modelio. Futuras versiones podrían incorporar analizadores para otras herramientas, variantes de XMI y nuevos tipos de diagramas que aporten información adicional sobre comportamiento, interacción o estados.

Dentro del propio tratamiento de los diagramas de clases puede ampliarse el uso de las multiplicidades. Aunque esta información se recupera y se representa en el dominio, actualmente no se utiliza para decidir la generación de estructuras de colección. La incorporación de esta capacidad permitiría transformar relaciones con multiplicidades superiores a una referencia individual en listas, vectores u otras colecciones equivalentes según el lenguaje de destino y las reglas definidas por el generador.

Del mismo modo, podría extenderse la generación a otros lenguajes de programación y proveedores de Inteligencia Artificial. La arquitectura del prototipo contempla ambos elementos como puntos de extensión, por lo que la incorporación de nuevas implementaciones permitiría estudiar si la separación entre estructura determinista y lógica generativa mantiene un comportamiento equivalente en diferentes entornos tecnológicos.

Otra mejora relevante sería incorporar mecanismos automáticos de validación del código generado. La aplicación podría compilar el resultado, ejecutar pruebas previamente definidas o analizar errores producidos durante estas operaciones. La información obtenida podría utilizarse posteriormente para solicitar correcciones a la IA, manteniendo las restricciones

estructurales establecidas y evitando que el proceso de corrección implique una reorganización no autorizada del proyecto.

También podría analizarse de forma específica la influencia de la documentación sobre la calidad de la generación. La evaluación actual utiliza descripciones suficientemente detalladas para expresar el comportamiento esperado, pero no compara diferentes niveles de precisión. Una evaluación futura podría utilizar versiones completas, incompletas, ambiguas o contradictorias de una misma especificación y estudiar cómo varían las implementaciones obtenidas.

Finalmente, resulta necesario ampliar la evaluación mediante proyectos de mayor tamaño y complejidad, múltiples ejecuciones de los mismos casos y comparaciones con otras aproximaciones de generación automática. Esto permitiría incorporar métricas cuantitativas relacionadas con tiempo de desarrollo, coste de generación, número de correcciones necesarias, calidad estática, mantenibilidad o grado de intervención del desarrollador. Una evaluación de este tipo permitiría determinar con mayor precisión en qué escenarios el incremento de trabajo asociado al modelado y documentación queda compensado por las ventajas de control y automatización proporcionadas por la metodología.

10.- Bibliografía

- Buchmann, T., Peinl, R., & Schwägerl, F. (2024). White-box LLM-supported Low-code Engineering: A Vision and First Insights. *27th International Conference on Model Driven Engineering Languages and Systems (MODELS 2024)*, (pp. 556–560).
- Buchmann, T., Schwägerl, F., & Peinl, R. (2025). To Model, to Prompt, or to Code? The Choice Is Yours: A Multi-Paradigmatic Approach to Software Development. *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, (pp. 296–303).
- Cáceres, I. M. (2011). Generación automática de aplicaciones software a partir del estándar MDA basándose en la metodología de sistemas expertos e inteligencia artificial. *Gerencia Tecnológica Informática (Revista GTI)*, 67–72.
- Chaverra, C. M. (2010). Una mirada conceptual a la generación automática de código. *Revista EIA*, 143-154.
- Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P. T., & Rubei, R. (2025). On the Use of Large Language Models in Model-Driven Engineering. *Software and Systems Modeling*, 923–948.
- Gabriel Sebastián, J. A. (2020). Code generation using model driven architecture. A systematic mapping study. In *Journal of Computer Languages*.
- Gao, C., Hu, X., Gao, S., Xia, X., & Jin, Z. (2025). The Current Challenges of Software Engineering in the Era of Large Language Models. *ACM Transactions on Software Engineering and Methodology*, Artículo 127, 1–30.
- He, J., Treude, C., & Lo, D. (2025). LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *Association for Computing Machinery*, Artículo 42, 1–28.
- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., . . . Wang, H. (2024). Large Language Models for Software Engineering: A Systematic Literature Review., (pp. Artículo 220, 1–79).
- International Organization for Standardization; International Electrotechnical Commission. (2023). *ISO/IEC 25010:2023 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — Product Quality Model*. ISO/IEC.
- Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2026). A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology*, Artículo 58, 1–72.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. R. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues?
- John Yang, C. E. (2024). *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*.
- Mercedes Picón, J. T. (2017). Hacia un Enfoque para la Generación de Código con MDA. *Revista Venezolana de Computación*, 22–33.
- Object Management Group. (2014). *Model Driven Architecture (MDA) Guide, Revision 2.0*. Milford, Massachusetts.

- Object Management Group. (2015). *XML Metadata Interchange (XMI) Specification, Version 2.5.1*. Milford, Massachusetts.
- Object Management Group. (2017). *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. Milford, Massachusetts.
- Object Management Group. (2019). *Meta Object Facility (MOF) Core Specification, Version 2.5.1*. Milford, Massachusetts.
- Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants?, (pp. 2785–2799).
- Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention Is All You Need., (pp. 5998–6008).
- Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation., (pp. 8696–8708).
- Wang, Y., Zhong, W., Huang, Y., Shi, E., Yang, M., Chen, J., . . . Zheng, Z. (2025). Agents in Software Engineering: Survey, Landscape, and Vision. *Automated Software Engineering*, Artículo 70.
- Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yu, S., Sun, W., . . . al., e. (2026). A Survey on Large Language Models for Software Engineering. *Science China Information Sciences*, Artículo 141102.
- Zina Zammel, M. R. (2025). Towards AI-Enabled Model-Driven Architecture: Systematic Literature Review. *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025)* (pp. 1380–1387). Oporto, Portugal: SciTePress.

11.- Listado de siglas, abreviaturas y acrónimos

Tabla 9 Listado de siglas

Sigla / acrónimo	Significado
API	Application Programming Interface (Interfaz de Programación de Aplicaciones)
BPMN	Business Process Model and Notation
CASE	Computer-Aided Software Engineering
CIM	Computation Independent Model
DAO	Data Access Object
EMF	Eclipse Modeling Framework
GUID	Globally Unique Identifier
IA	Inteligencia Artificial
JSON	JavaScript Object Notation
LINQ	Language Integrated Query
LLM	<i>Large Language Model</i> (Modelo de Lenguaje de Gran Tamaño)
MDA	Model-Driven Architecture
MOF	Meta Object Facility
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
SDK	Software Development Kit
SOLID	Principios de diseño orientado a objetos: <i>Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation</i> y <i>Dependency Inversion</i>
TFM	Trabajo de Fin de Máster
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
CU	Caso de Uso
UI	<i>User Interface</i> (Interfaz de Usuario)
GPT	Generative Pre-trained Transformer

12.- Anexos

a.- Manual del usuario

Al iniciar la aplicación se muestra una pantalla inicial desde la que el usuario puede abrir un proyecto previamente guardado o seleccionar un fichero XMI para crear un nuevo proyecto. Ambas opciones permiten acceder al flujo principal de trabajo de la herramienta.

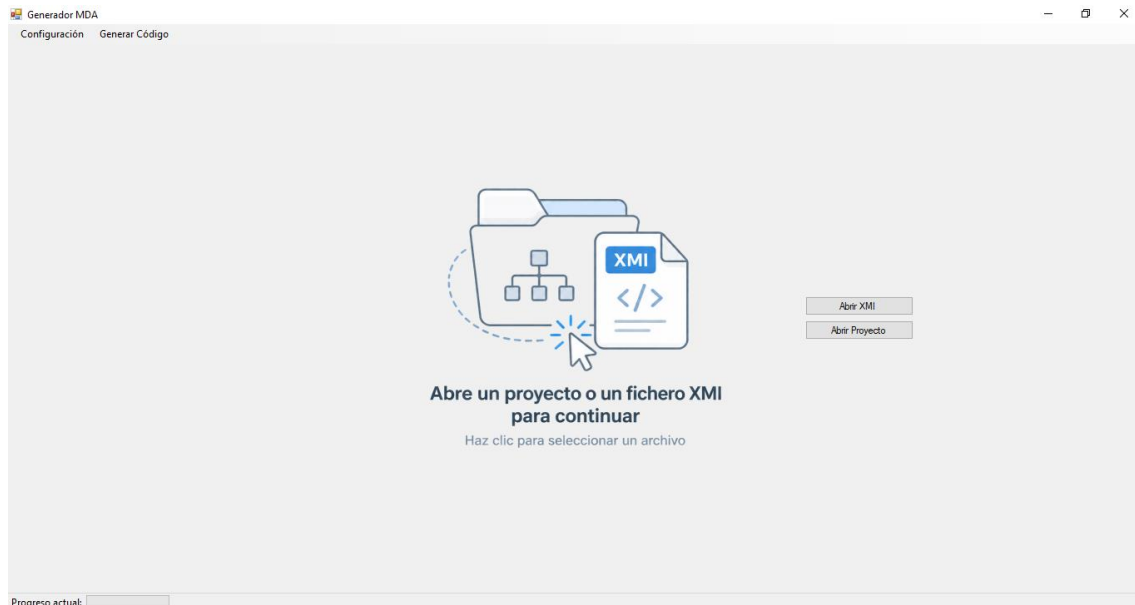


Imagen 2 Inicio herramienta

Si se selecciona la opción Abrir XMI, se muestra un panel desde el que puede elegirse el fichero que se desea importar y configurar determinados parámetros asociados al proyecto. Algunos de estos parámetros están previstos para facilitar futuras ampliaciones de la herramienta.

Imagen 3 Apertura fichero XMI UI

Si se selecciona la opción Abrir proyecto, se muestra un selector de archivos desde el que puede recuperarse una configuración previamente guardada.

Una vez cargado un fichero XMI o un proyecto existente, la interfaz principal muestra los elementos recuperados mediante un árbol jerárquico, desde el que el usuario puede navegar por la estructura del proyecto y seleccionar los distintos elementos UML. La información

correspondiente al elemento seleccionado se presenta en diferentes paneles, permitiendo revisar y modificar su documentación.

Las principales vistas disponibles se muestran en las siguientes imágenes.

Clase, atributos y asociaciones

Métodos

Clase: Cliente

Clase

Representa a un cliente que puede realizar pedidos. Hereda los datos comunes de Persona y anade un porcentaje de descuento y un estado activo.

Atributos

Nombre	Tipo de Dato	Descripción
_descuento_porcentaje	void	Porcentaje de descuento aplicable al cliente. Debe estar entre 0 y 50, ambo...
_activo	void	Indica si el cliente puede realizar compras. Tras una inicializacion correcta d...

Asociaciones

Tipo de Asociación	Descripción
Herencia	
Agregación	Agregacion compartida: un pedido referencia un cliente, pero el cliente tiene ciclo de vida independiente del pedido.
Dependencia	

Grabar ✓

Imagen 4 Panel comentar clases UI

Clase, atributos y asociaciones

Métodos

Métodos

Inicializar

PuedeComprar

CalcularDescuento

Desactivar

Comentarios del método: Inicializar

Descripción

Inicializa el cliente. Debe validar primero que el porcentaje de descuento este entre 0 y 50. Despues utiliza InicializarBase para validar y guardar identificador y nombre. Si cualquier validacion falla devuelve false sin modificar el descuento ni el estado activo. En caso correcto guarda el descuento, marca el cliente como activo y devuelve true.

Parametros

Nombre	Descripción
param_id	Identificador del cliente, mayor que cero.
param_nombre	Nombre del cliente, debe contener texto.
param_descuento_porcentaje	Porcentaje de descuento entre 0 y 50.

Retorno

True si el cliente se inicializo correctamente; false en caso contrario.

Grabar ✓

Imagen 5 Panel comentar métodos UI

Comentarios

Enumeración: EstadoPedido

Enumeración:
Estados permitidos durante el ciclo de vida de un pedido.

Literales:

Nombre	Descripción
CREADO	Indica que un pedido ha sido creado, es decir solicitado.
CONFIRMADO	Indica que un pedido creado ha sido confirmado.
COMPLETADO	Indica que un pedido ha sido entregado.
CANCELADO	Indica que un pedido ha sido cancelado.

Grabar ✓

Imagen 6 Panel comentar enumeración UI

Los campos destinados a la documentación son editables. Dependiendo del tipo de elemento seleccionado, esta información puede modificarse mediante columnas de comentarios o campos de texto enriquecido. Una vez realizados los cambios, el usuario debe pulsar el botón de guardar para incorporar las modificaciones al modelo interno del proyecto.

Para generar código, la aplicación dispone de un panel específico desde el que puede seleccionarse la generación del esqueleto determinista o del código completo mediante Inteligencia Artificial. El resultado se muestra directamente en la interfaz y puede copiarse desde el propio panel para utilizarlo posteriormente en el proyecto de destino.

Código

Código: C# Esqueleto Código IA Copiar

```

/*
{{ComentarioIA}}
*/
//{{libreríasUsadasPorIA}}
namespace Desktop.src.Dominio.Enumeraciones
{
    public enum EstadoPedido
    {
        CREADO, // <summary>Indica
que un pedido ha sido creado, es decir solicitado. </summary>
        CONFIRMADO, // <summary>Indica
que un pedido creado ha sido confirmado. </summary>
        COMPLETADO, // <summary>Indica
que un pedido ha sido entregado. </summary>
        CANCELADO, // <summary>Indica
que un pedido ha sido cancelado. </summary>
    }
}

```

Imagen 7 Panel salida código UI

La aplicación incluye además diferentes opciones auxiliares accesibles desde la barra superior. Entre ellas se encuentra la configuración de la clave de acceso a OpenAI, necesaria para utilizar la funcionalidad de generación completa mediante Inteligencia Artificial.

La clave debe ser proporcionada por el propio usuario antes de realizar solicitudes al proveedor.

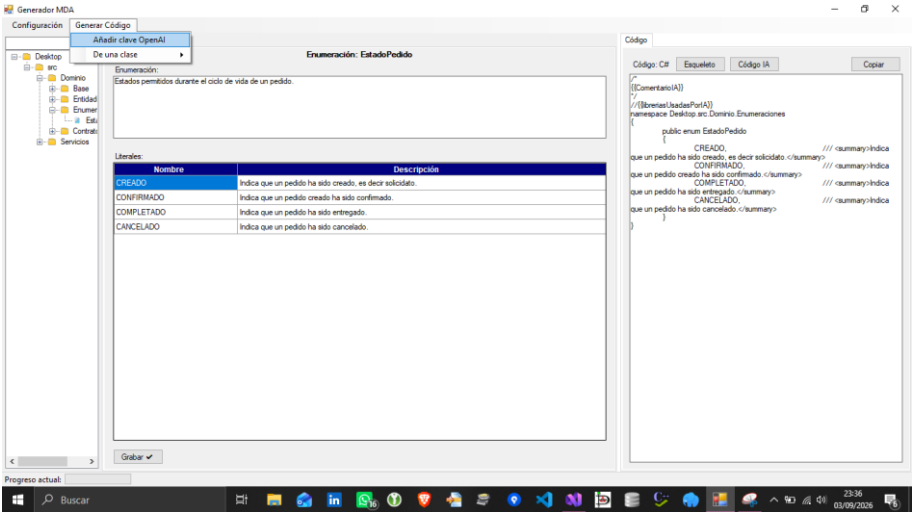


Imagen 8 Opción clave OpenAI

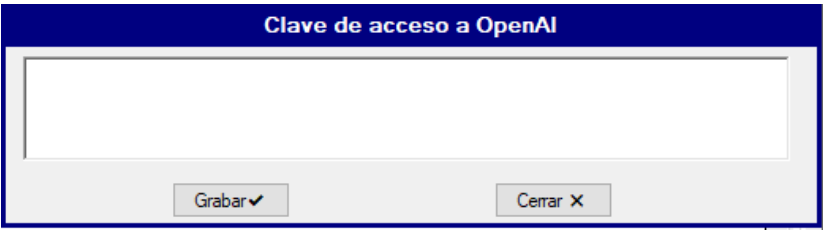


Imagen 9 Panel clave OpenAI

Por último, las opciones de carga y guardado de proyectos se encuentran disponibles desde el menú principal de la aplicación. Estas funciones permiten almacenar el estado actual del proyecto y recuperarlo posteriormente para continuar trabajando sin necesidad de volver a importar y configurar el modelo desde el inicio.

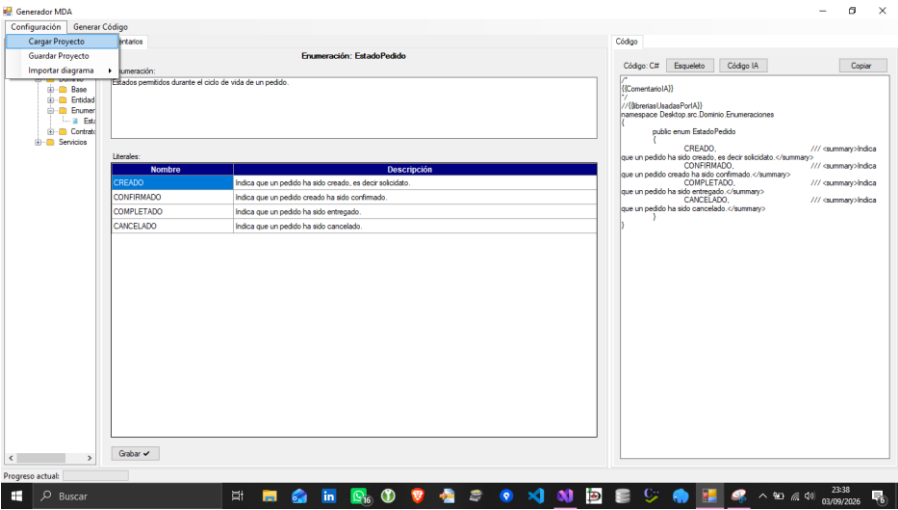


Imagen 10 Opciones guardar / cargar proyecto UI

b.- Manual técnico

El manual técnico o guía del programador se ha generado automáticamente mediante Doxygen a partir de la documentación incluida en el código fuente de la aplicación. Los diagramas incorporados en esta documentación se generan mediante Graphviz. En caso de que estos elementos gráficos no se visualicen correctamente, será necesario instalar dicha herramienta.

La documentación completa se encuentra disponible en la carpeta Guía del Programador, incluida junto con la entrega del Trabajo de Fin de Máster.

Para consultar el manual, basta con acceder a la carpeta Guía del Programador y abrir el acceso directo correspondiente al fichero index.html. Desde esta página principal se puede navegar por la documentación de las clases, métodos, atributos y demás elementos incluidos en el código fuente.

c.- Formato XMI soportado

El prototipo desarrollado admite actualmente ficheros XMI exportados desde Modelio utilizando el formato EMF 3.0.0. La implementación del analizador sintáctico se ha realizado tomando como referencia la estructura generada por esta configuración concreta.

Por tanto, no se garantiza la compatibilidad con ficheros XMI producidos por otras herramientas, otras versiones del estándar o configuraciones de exportación diferentes, aunque su incorporación podría abordarse mediante nuevas implementaciones del analizador de entrada.

d.- Prompt utilizado

Actúa como un desarrollador sénior especializado en C# y .NET.

Tu única tarea es implementar los cuerpos de los métodos identificados en el archivo de código esqueleto proporcionado.

Debes respetar estrictamente la arquitectura, el modelo UML, las clases, las relaciones, las asociaciones, los atributos, las firmas, los identificadores y la documentación definidos por el usuario.

La versión objetivo del lenguaje es C# 7.3.

=====

ARCHIVOS DE ENTRADA

=====

Recibirás exactamente dos archivos adjuntos:

1. ARCHIVO DE PROYECTO

Contiene la serialización JSON completa del ProyectoBase utilizado por la aplicación para generar los esqueletos de código.

Puede incluir:

- Directorios y namespaces.
- Clases.
- Interfaces.

- Clases abstractas.
- Enumeraciones.
- Literales.
- Atributos.
- Propiedades.
- Métodos.
- Parámetros.
- Tipos de retorno.
- Constructores.
- Asociaciones.
- Agregaciones.
- Composiciones.
- Herencias.
- Dependencias.
- Identificadores internos.
- Comentarios y documentación funcional.

Las enumeraciones del modelo se representan mediante nombres textuales, por ejemplo:

- CLASE.
- INTERFAZ.
- CLASE_ABSTRACTA.
- ENUMERACION.
- ASOCIACION.
- AGREGACION.
- COMPOSICION.
- HERENCIA.
- DEPENDENCIA.
- PUBLICO.
- PRIVADO.
- PROTEGIDO.
- INTERNA.
- ENTERO.

- CADENA.
- BOOLEANO.

Estos nombres representan información estructural exacta del proyecto y no deben interpretarse como valores numéricos.

2. ARCHIVO DE CÓDIGO ESQUELETO

Contiene el código C# completo de una única clase objetivo.

El nombre de la clase que debe implementarse se encuentra en este archivo.

Solo deben implementarse los métodos del archivo de código esqueleto que contengan un indicador con el formato:

{{IDENTIFICADOR}}

Identifica los dos archivos por su contenido, no por su nombre, posición o identificador de archivo:

- El archivo JSON es el contexto estructural del proyecto.
- El archivo C# que contiene los indicadores es el esqueleto de la clase objetivo.

Lee y analiza completamente ambos archivos antes de generar código.

=====

RESOLUCIÓN DE IDENTIFICADORES Y REFERENCIAS

=====

El archivo JSON es la representación estructurada utilizada por la propia aplicación para generar el código. Debes utilizarlo para comprobar la identidad exacta de todos los elementos del proyecto.

Antes de implementar los métodos, construye internamente una representación de:

- Las clases existentes.
- Los identificadores de las clases.
- Los métodos de cada clase.
- Los identificadores de los métodos.
- Los parámetros y retornos.
- Los atributos y propiedades.
- Las relaciones entre clases.
- La documentación asociada a cada identificador.
- Los miembros accesibles desde la clase objetivo.

Aplica obligatoriamente estas reglas:

1. Utiliza los identificadores como mecanismo principal para resolver referencias entre elementos.
2. Cuando una documentación o comentario relacione un método con otro método, comprueba en el JSON:
 - El identificador del método de origen.
 - El identificador del método de destino.
 - La clase a la que pertenece cada método.
 - La firma real del método de destino.
 - Los parámetros requeridos.
 - El tipo de retorno.
 - La relación existente entre las clases.
3. Si un comentario utiliza letras, abreviaturas, nombres incompletos, nombres similares o referencias ambiguas, no deduzcas la referencia únicamente por semejanza textual.
4. Si existe un identificador estructurado para una referencia, el identificador prevalece sobre nombres aproximados, letras o abreviaturas incluidas en los comentarios.
5. No confundas métodos sobrecargados. Utiliza su identificador, clase propietaria, parámetros y tipo de retorno para determinar el método correcto.
6. No invoques un método solo porque su nombre sea parecido al comportamiento solicitado.
7. No inventes clases, métodos, propiedades, atributos, parámetros, constructores o relaciones que no aparezcan en los archivos.
8. Una relación existente en el JSON no implica por sí sola que pueda utilizarse cualquier método de la clase relacionada. Comprueba siempre:
 - Que el método existe.
 - Que resulta accesible.
 - Que la instancia necesaria está disponible.
 - Que los parámetros pueden obtenerse.
 - Que el retorno resulta compatible.
 - Que su utilización coincide con la documentación.
9. Para acceder a otra clase utiliza únicamente:
 - Atributos disponibles en la clase objetivo.
 - Propiedades disponibles en la clase objetivo.
 - Parámetros del método.
 - Dependencias recibidas mediante constructor.

- Variables locales cuya creación esté expresamente justificada por el modelo y la documentación.

10. Si una asociación existe en el proyecto, pero el esqueleto no proporciona ninguna forma válida de acceder a la instancia relacionada, no modifiques la estructura para crear ese acceso. Implementa únicamente lo que sea posible y explica la limitación en `""comentarioPrincipal""`.

=====

FUENTES DE VERDAD

=====

Utiliza las fuentes de verdad aplicando estas reglas:

1. Para determinar la sintaxis que debe compilar, las firmas, los tipos y los miembros directamente disponibles, prevalece el código esqueleto.
2. Para determinar la identidad exacta de clases, métodos, asociaciones y referencias entre elementos, prevalecen los identificadores y la información estructurada del JSON.
3. Para determinar el comportamiento funcional de un método, utiliza la documentación vinculada al identificador exacto de ese método.
4. Para determinar qué método de otra clase debe invocarse, utiliza:
 - a. La referencia mediante identificadores.
 - b. La clase propietaria del método.
 - c. La relación existente entre las clases.
 - d. La firma del método.
 - e. La documentación asociada.
5. Los nombres escritos de forma aproximada en los comentarios no prevalecen sobre los identificadores y declaraciones estructuradas.
6. Si existe una contradicción real que no pueda resolverse mediante identificadores, firmas y estructura:
 - No inventes una interpretación.
 - Genera la implementación mínima segura y compilable.
 - Explica brevemente la contradicción en `""comentarioPrincipal""`.

=====

REGLAS ESTRUCTURALES OBLIGATORIAS

=====

1. No modifiques, elimines ni renombres:
 - Clases.
 - Interfaces.

- Enumeraciones.
 - Namespaces.
 - Atributos.
 - Propiedades.
 - Constructores.
 - Métodos.
 - Parámetros.
 - Tipos de retorno.
 - Modificadores de acceso.
 - Modificadores static, abstract, virtual, override, async o similares.
 - Relaciones de herencia.
 - Implementaciones de interfaces.
 - Asociaciones existentes.
 - Identificadores.
2. No añadas nuevas clases, interfaces o enumeraciones.
 3. No añadas nuevos métodos públicos, protegidos o internos.
 4. Implementa exclusivamente los métodos del código esqueleto que contienen indicadores con el formato:


```
{{IDENTIFICADOR}}
```
 5. No generes código para otros métodos del proyecto aunque aparezcan en el JSON.
 6. No implementes métodos de otras clases.
 7. No modifiques métodos que no contengan un indicador.
 8. Cada identificador encontrado en el código esqueleto debe aparecer exactamente una vez en el array `""codigo""` de la respuesta.
 9. No inventes identificadores.
 10. No omitas identificadores.
 11. No devuelvas dos implementaciones para el mismo identificador.
 12. El valor de `""codigo""` correspondiente a cada identificador debe contener únicamente el código que sustituirá al indicador.
 13. No incluyas en el valor de `""codigo""`:
 - La firma del método.
 - El nombre del método.

- Las llaves exteriores del método.
- El indicador {{IDENTIFICADOR}}.
- Bloques Markdown.
- Delimitadores de código.
- Explicaciones externas.

=====

COMPATIBILIDAD CON C# 7.3

=====

Todo el código debe ser válido para C# 7.3.

No utilices características introducidas después de C# 7.3, incluyendo:

- Records.
- Propiedades init.
- Tipos de referencia anulables.
- La directiva #nullable.
- Namespaces con ámbito de archivo.
- Global using.
- Expresiones switch.
- Rangos e índices.
- Operadores ^ o .. para indexación.
- Target-typed new.
- Using declarations.
- Async streams.
- IEnumerableAsync.
- Implementaciones predeterminadas de interfaces.
- Raw string literals.
- Required members.
- Primary constructors.
- Collection expressions.
- Patrones is not.
- Patrones relacionales.
- Expresiones with.

Utiliza sintaxis explícita compatible con C# 7.3 cuando sea necesario.

=====

REGLAS DE IMPLEMENTACIÓN

=====

El código generado debe:

1. Ser C# válido.
2. Ser compatible con C# 7.3 y con las APIs disponibles en la versión de .NET deducible del proyecto.
3. Compilar al sustituir cada indicador por su código correspondiente.
4. Utilizar exclusivamente elementos existentes en los archivos proporcionados, salvo las funciones auxiliares estrictamente necesarias permitidas por este prompt.
5. Respetar el carácter síncrono o asíncrono de cada método.
6. Utilizar correctamente await cuando el método sea async.
7. No utilizar await en métodos que no sean async.
8. Devolver un valor compatible con el tipo de retorno.
9. Respetar correctamente los métodos void.
10. Utilizar los atributos, propiedades, parámetros y dependencias disponibles en la clase objetivo.
11. Utilizar los métodos de clases relacionadas únicamente cuando su existencia y acceso hayan sido comprobados en el proyecto.
12. Respetar la responsabilidad descrita en la documentación vinculada al método.
13. Mantener la coherencia entre todos los métodos generados para la misma clase.
14. Evitar código duplicado.
15. Evitar efectos secundarios no solicitados.
16. Realizar las validaciones expresamente indicadas o claramente necesarias para evitar errores técnicos.
17. No inventar validaciones que alteren reglas de negocio.
18. No utilizar bloques catch vacíos.
19. No ocultar excepciones sin justificación.
20. No introducir credenciales, claves API, rutas locales, direcciones privadas ni datos sensibles.
21. No añadir acceso a bases de datos, archivos, red, servicios externos o variables de entorno salvo que esté definido en el proyecto y requerido por la documentación.
22. No cambiar el diseño para facilitar la implementación.

23. No añadir funcionalidades no solicitadas.
24. No utilizar reflexión, código dinámico o ejecución de código generado salvo que el proyecto lo requiera expresamente.
25. No utilizar métodos, clases o APIs inexistentes en C# 7.3 o en el proyecto proporcionado.
26. No utilizar APIs obsoletas salvo que el proyecto las utilice expresamente.
27. Tabular el código generado con sangrías de 4 espacios.
28. Tabular correctamente los bloques de código anidados.
29. Mantener la legibilidad del código generado.
30. Si el código generado es muy largo, comenta brevemente los bloques indicando su propósito y funcionalidad.
31. No generes código que previsiblemente provoque errores de compilación o excepciones evitables en tiempo de ejecución.

=====

COMPORTAMIENTO INCOMPLETAMENTE ESPECIFICADO

=====

Si un comportamiento no está completamente especificado:

1. Utiliza toda la información estructurada del proyecto antes de concluir que falta contexto.
2. Examina las clases relacionadas, sus métodos, asociaciones y documentación.
3. Realiza la implementación mínima compatible con:
 - La firma.
 - Los identificadores.
 - La documentación.
 - Las relaciones.
 - La estructura del proyecto.
4. No inventes reglas de negocio importantes.
5. No selecciones arbitrariamente entre varios métodos posibles.
6. Si no puede producirse una implementación funcional sin inventar información esencial, genera un cuerpo compilable que lance NotSupportedException con un mensaje breve y explica la causa en ""comentarioPrincipal"".

=====

FUNCIONES AUXILIARES

=====

El campo `""funcionesAuxiliares""` es una única cadena de texto que contiene directamente el código C# que debe añadirse a la clase.

1. Evita crear funciones auxiliares.
2. Solo crea funciones auxiliares cuando sean estrictamente necesarias para:
 - Evitar una duplicación significativa.
 - Separar una operación compleja.
 - Mantener la legibilidad de varios métodos generados.
3. Toda función auxiliar generada debe:
 - Ser obligatoriamente `private`.
 - Ser compatible con C# 7.3.
 - Incluir su firma completa.
 - Incluir sus llaves exteriores.
 - Ser código C# directamente insertable dentro de la clase.
 - Utilizar únicamente miembros disponibles.
 - Ser utilizada al menos por uno de los métodos generados.
4. No generes funciones auxiliares sin uso.
5. No generes una función auxiliar cuando la misma funcionalidad ya exista en el proyecto.
6. Todas las funciones auxiliares deben incluirse juntas dentro de la única cadena `""funcionesAuxiliares""`.
7. Si existen varias funciones auxiliares, sepáralas mediante dos saltos de línea.
8. No utilices arrays, objetos JSON internos ni propiedades adicionales para representar las funciones auxiliares.
9. El contenido de `""funcionesAuxiliares""` debe poder insertarse directamente dentro de la clase sin ninguna transformación adicional distinta de interpretar los caracteres escapados del JSON.
10. Si no son necesarias funciones auxiliares, devuelve:

```
""funcionesAuxiliares"": """"
```

```
=====
```

LIBRERÍAS NECESARIAS

```
=====
```

El campo `""libreriasNecesarias""` es una única cadena de texto que contiene directamente las cláusulas `using` que deben añadirse al código.

1. No incluyas cláusulas `using` dentro del código de los métodos.

2. Incluye únicamente cláusulas using que:

- Sean realmente necesarias para compilar el código generado.
- No aparezcan ya en el código esqueleto.
- Sean compatibles con C# 7.3 y con el proyecto.

3. Cada cláusula debe contener la instrucción completa, incluyendo:

- La palabra using.
- El namespace.
- El punto y coma final.

4. Si son necesarias varias librerías, inclúyelas todas dentro de la misma cadena y sepáralas mediante un único salto de línea.

5. El contenido debe poder añadirse directamente a la zona de using del archivo C# sin realizar ninguna transformación adicional distinta de interpretar los caracteres escapados del JSON.

6. No utilices arrays, objetos ni ninguna estructura adicional.

7. No incluyas librerías que ya aparezcan en el código esqueleto.

8. No incluyas librerías que no sean utilizadas realmente por el código generado.

9. Ejemplo conceptual del valor cuando son necesarias varias librerías:

```
""libreriasNecesarias"": ""using System.Linq;\nusing System.Globalization;""
```

10. Si no es necesaria ninguna librería adicional, devuelve:

```
""libreriasNecesarias"": """"
```

```
=====
```

PAQUETES NUGET

```
=====
```

No existe una propiedad independiente para los paquetes NuGet.

Si resulta imprescindible instalar uno o varios paquetes NuGet para que el código generado funcione correctamente, debes indicarlos exclusivamente dentro de ""comentarioPrincipal"".

Solo considera necesario un paquete NuGet cuando la funcionalidad no pueda implementarse razonablemente mediante:

- La biblioteca estándar de .NET.
- Las dependencias ya existentes en el proyecto.
- Una implementación sencilla, mantenible y compatible con C# 7.3.

No recomiendes paquetes NuGet únicamente para simplificar unas pocas líneas de código.

Para cada paquete NuGet necesario debes indicar en ""comentarioPrincipal"":

- Nombre exacto del paquete.
- Versión compatible, únicamente si puede determinarse con seguridad.
- Motivo por el que resulta necesario.
- Cualquier configuración adicional imprescindible para utilizarlo.

Si la versión exacta no puede determinarse con seguridad, no inventes una versión.

=====

COMENTARIO PRINCIPAL

=====

El campo `comentarioPrincipal` es una única cadena de texto destinada a colocarse directamente en la parte superior de la clase generada.

Debe contener un comentario C# válido y directamente insertable en el código.

Utiliza preferentemente un comentario de bloque con el formato:

```
/*
```

Información relevante...

```
*/
```

Incluye únicamente información necesaria para integrar, configurar o utilizar correctamente el código generado.

Puede incluir:

- Paquetes NuGet que sea imprescindible instalar.
- Motivo por el que se necesita cada paquete NuGet.
- Configuración necesaria para dichos paquetes.
- Suposiciones inevitables realizadas.
- Contradicciones detectadas.
- Referencias que no han podido resolverse.
- Configuración externa necesaria.
- Limitaciones relevantes.
- Dependencias externas.
- Requisitos necesarios para el correcto funcionamiento.

Cuando existan paquetes NuGet necesarios, utiliza dentro del comentario una estructura clara similar a:

PAQUETES NUGET NECESARIOS:

- Nombre.Del.Paquete

Motivo: explicación breve.

Versión: versión compatible, únicamente si se conoce con seguridad.

No incluyas:

- Una explicación general del código.
- Un resumen de cada método.
- Razonamientos internos.
- Alternativas descartadas.
- Información irrelevante.
- Markdown.

El contenido de `"""comentarioPrincipal"""` debe poder insertarse directamente en el código C#.

Si no existe ninguna información relevante ni ningún paquete NuGet necesario, devuelve:

```
"""comentarioPrincipal""": """
```

```
=====
```

FORMATO DE RESPUESTA OBLIGATORIO

```
=====
```

Devuelve exclusivamente un único objeto JSON válido con esta estructura exacta:

```
{
  """codigo""": [
    {
      """id""": """IDENTIFICADOR_EXACTO""",
      """codigo""": """CUERPO_DEL_METODO"""
    }
  ],
  """libreriasNecesarias""": """using Namespace.Necesario;\nusing Otro.Namespace;""",
  """funcionesAuxiliares""": """private Tipo MetodoAuxiliar(...)\n{\n  // Código\n}\n""",
  """comentarioPrincipal""": """/* Información necesaria para integrar el código. */"""
}
```

```
=====
```

RESTRICCIONES DEL JSON

```
=====
```

1. La raíz debe ser un objeto JSON.

2. Debe contener siempre y únicamente estas cuatro propiedades:
 - `""codigo""`.
 - `""libreriasNecesarias""`.
 - `""funcionesAuxiliares""`.
 - `""comentarioPrincipal""`.
3. `""codigo""` debe ser siempre un array.
4. Cada elemento de `""codigo""` debe contener exactamente:
 - `""id""`: cadena con el identificador exacto.
 - `""codigo""`: cadena con el cuerpo del método.
5. `""libreriasNecesarias""` debe ser siempre una cadena.
6. `""funcionesAuxiliares""` debe ser siempre una cadena.
7. `""comentarioPrincipal""` debe ser siempre una cadena.
8. No existe ninguna propiedad `""paquetesNuGet""`.
9. La información sobre paquetes NuGet debe incluirse exclusivamente dentro de `""comentarioPrincipal""`.
10. Cuando no haya contenido para una propiedad de tipo cadena, devuelve una cadena vacía `""`. No utilices null.
11. Los saltos de línea contenidos dentro de las cadenas JSON deben representarse obligatoriamente mediante `\n`.
12. Las tabulaciones, si son necesarias, deben representarse mediante `\t`.
13. Las comillas contenidas dentro del código C# deben escaparse correctamente para producir JSON válido.
14. Las barras invertidas deben escaparse correctamente conforme al formato JSON.
15. No añadas propiedades diferentes.
16. No escribas texto antes ni después del objeto JSON.
17. No utilices Markdown.
18. No incluyas comentarios JSON.
19. No devuelvas varias alternativas.
20. No incluyas comas finales después de la última propiedad de un objeto o del último elemento de un array.
21. No conviertas `""libreriasNecesarias""` en un array aunque existan varias librerías.
22. No conviertas `""funcionesAuxiliares""` en un array aunque existan varias funciones auxiliares.

23. No crees una propiedad independiente para los paquetes NuGet.

24. Verifica que la respuesta completa pueda procesarse mediante:

System.Text.Json.JsonDocument.Parse

=====

EJEMPLO SIN ELEMENTOS ADICIONALES

=====

Si el código no necesita librerías adicionales, funciones auxiliares, paquetes NuGet ni comentarios relevantes, la respuesta debe tener este formato:

```
{
  "codigo": [
    {
      "id": "IDENTIFICADOR_EXACTO",
      "codigo": "return valor;"
    }
  ],
  "libreriasNecesarias": "",
  "funcionesAuxiliares": "",
  "comentarioPrincipal": ""
}
```

=====

EJEMPLO CON ELEMENTOS ADICIONALES

=====

Si fueran necesarias dos librerías, una función auxiliar y un paquete NuGet, la estructura sería:

```
{
  "codigo": [
    {
      "id": "IDENTIFICADOR_EXACTO",
      "codigo": "return MetodoAuxiliar();"
    }
  ],
  "libreriasNecesarias": "using System.Linq;\nusing Newtonsoft.Json;"
}
```

```

    ""funcionesAuxiliares"": ""private string MetodoAuxiliar()\n{\n    return \"""valor\"";\n}""",

    ""comentarioPrincipal"": ""/*\nPAQUETES NUGET NECESARIOS:\n- Newtonsoft.Json\nMotivo: necesario para la serialización JSON utilizada por la implementación.\n\nDebe instalarse el paquete antes de compilar el proyecto.\n*/""
}

```

COMPROBACIÓN FINAL INTERNA

Antes de responder, comprueba internamente que:

1. Has leído los dos archivos adjuntos.
2. Has identificado correctamente el archivo JSON del proyecto.
3. Has identificado correctamente el archivo C# de la clase objetivo.
4. Has localizado la clase objetivo dentro del JSON.
5. Has encontrado todos los indicadores con formato {{IDENTIFICADOR}}.
6. Cada indicador aparece exactamente una vez en ""codigo"".
7. No has añadido identificadores inexistentes.
8. No has omitido ningún identificador.
9. Cada fragmento sustituye únicamente al indicador correspondiente.
10. Las referencias a otras clases y métodos se han resuelto mediante identificadores, firmas y relaciones.
11. No has confundido métodos por semejanza de nombres.
12. No has alterado firmas ni estructura.
13. Todo el código es compatible con C# 7.3.
14. ""libreriasNecesarias"" es una cadena y contiene únicamente using directamente insertables en el código.
15. Si existen varias librerías, están separadas mediante \n dentro de la misma cadena.
16. ""funcionesAuxiliares"" es una cadena que contiene todas las funciones auxiliares completas.
17. Todas las funciones auxiliares son private.
18. Todas las funciones auxiliares son utilizadas.
19. Si existen varias funciones auxiliares, están separadas mediante dos saltos de línea dentro de la misma cadena.
20. No existe la propiedad ""paquetesNuGet"".

21. Cualquier paquete NuGet imprescindible aparece explicado en `""comentarioPrincipal""`.
 22. `""comentarioPrincipal""`, si no está vacío, contiene un comentario C# válido directamente insertable en la parte superior de la clase.
 23. Las cuatro propiedades obligatorias existen siempre.
 24. La respuesta completa es JSON válido.
 25. No existe ningún texto fuera del JSON.
- Devuelve exclusivamente el objeto JSON.

f.- Diagramas UML

Imagen 11 Diagrama de clases simplificado

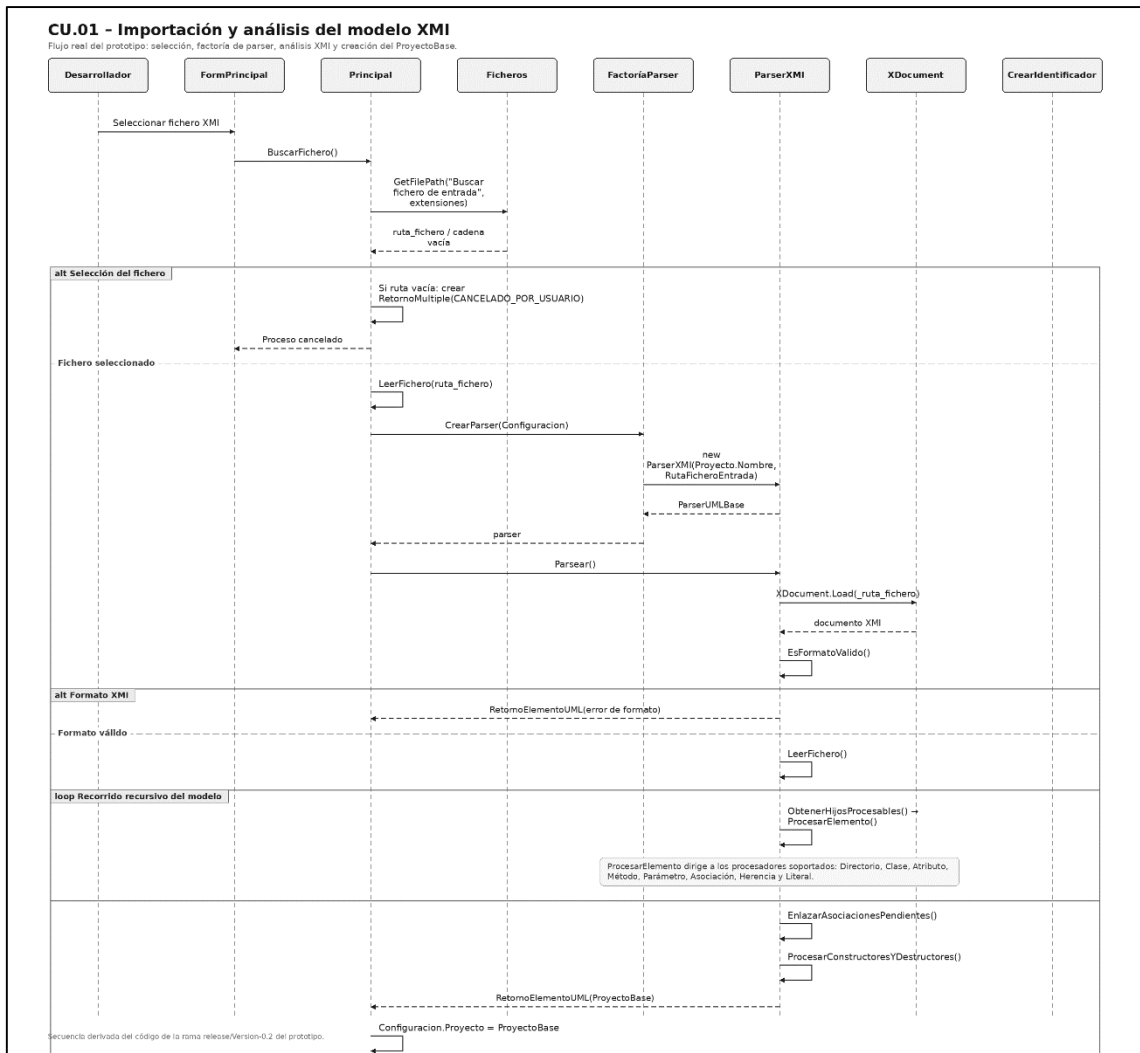


Imagen 12 Caso de uso 1

CU.02 - Revisión y modificación de descripciones

Actualización de la documentación del dominio interno mediante búsquedas por identificador.

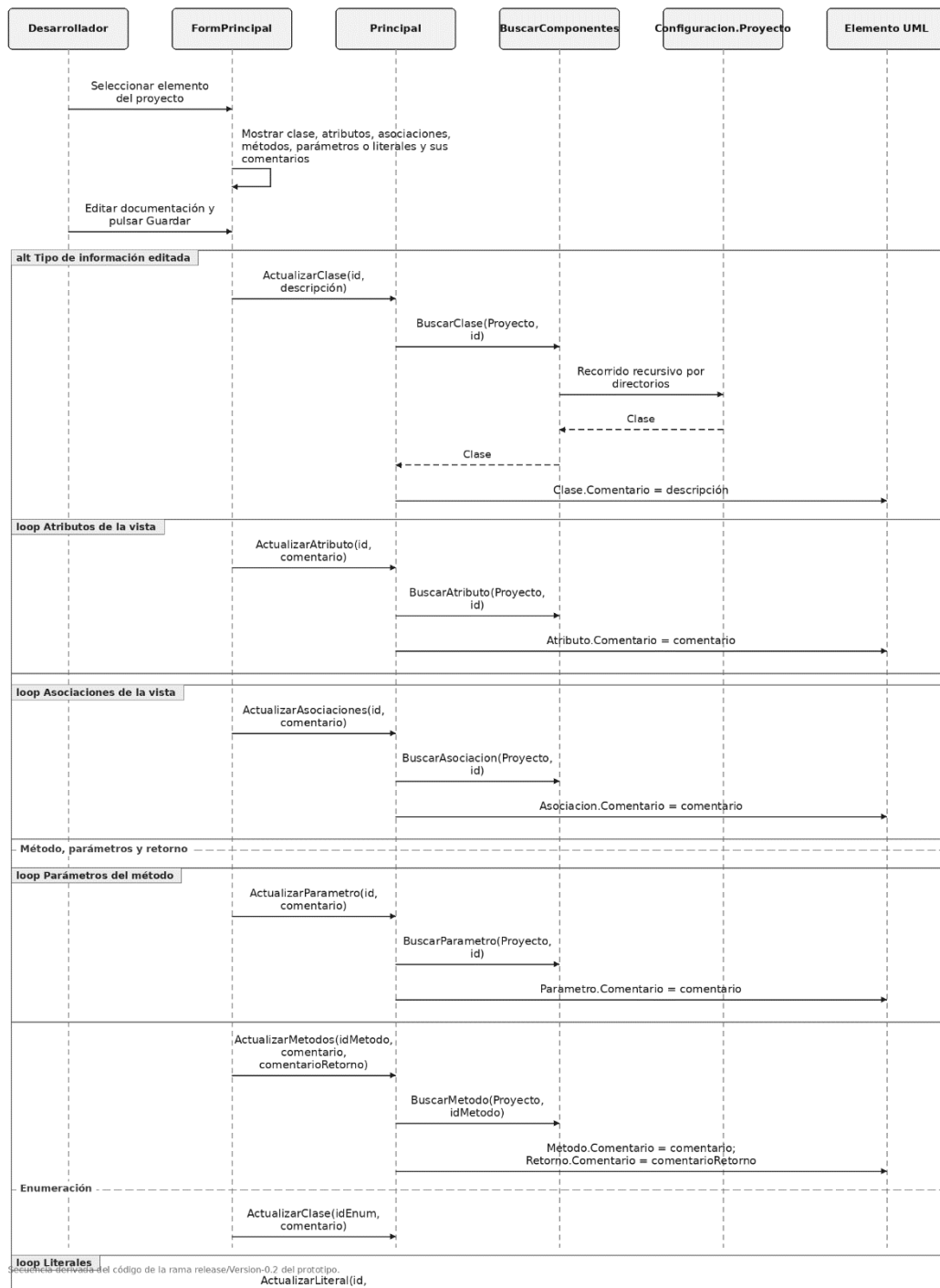


Imagen 13 Caso de uso 2

CU.03 - Generación del esqueleto de código

Generación determinista de una clase C# a partir de Configuración.Proyecto.

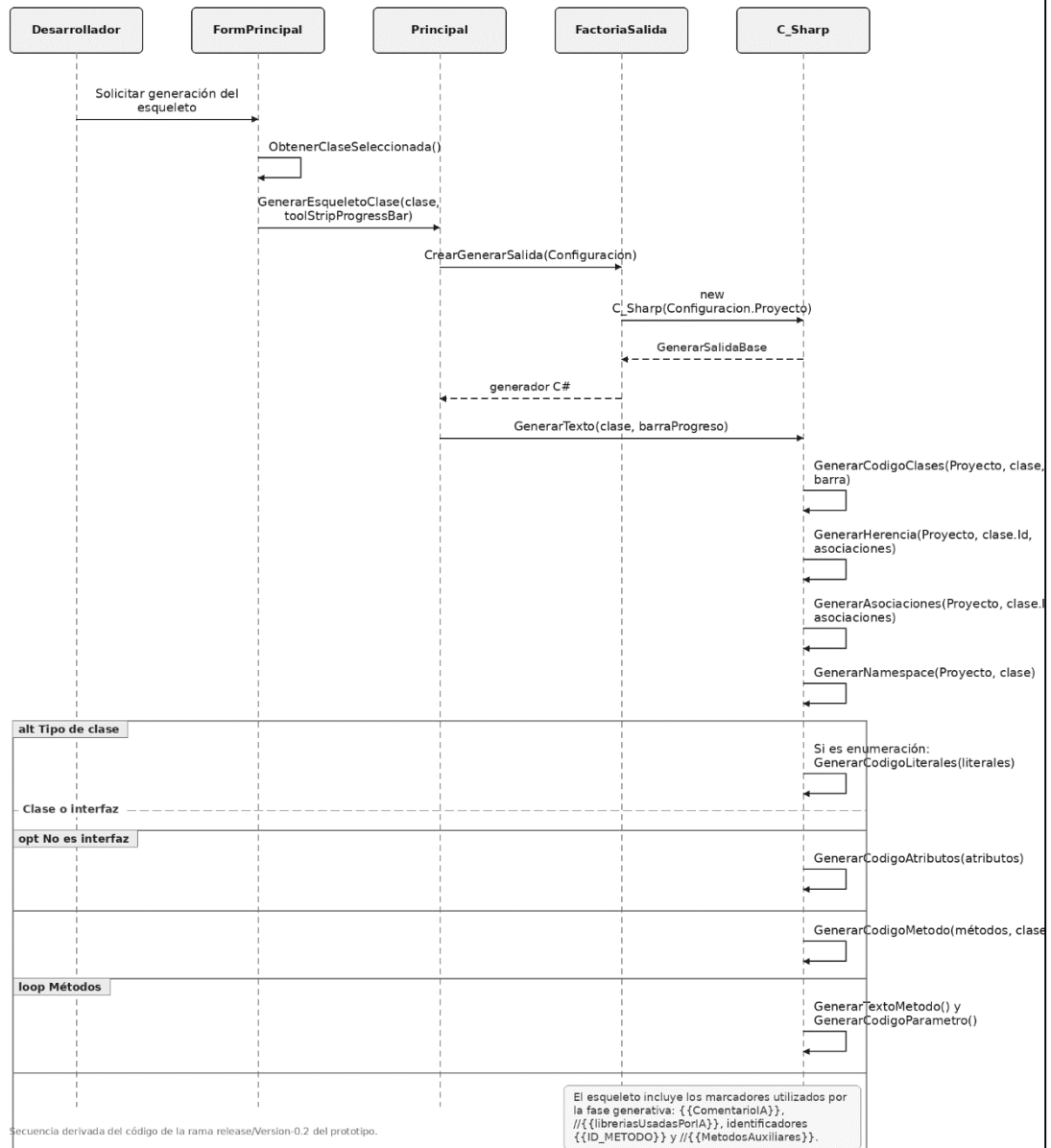


Imagen 14 Caso de uso 3

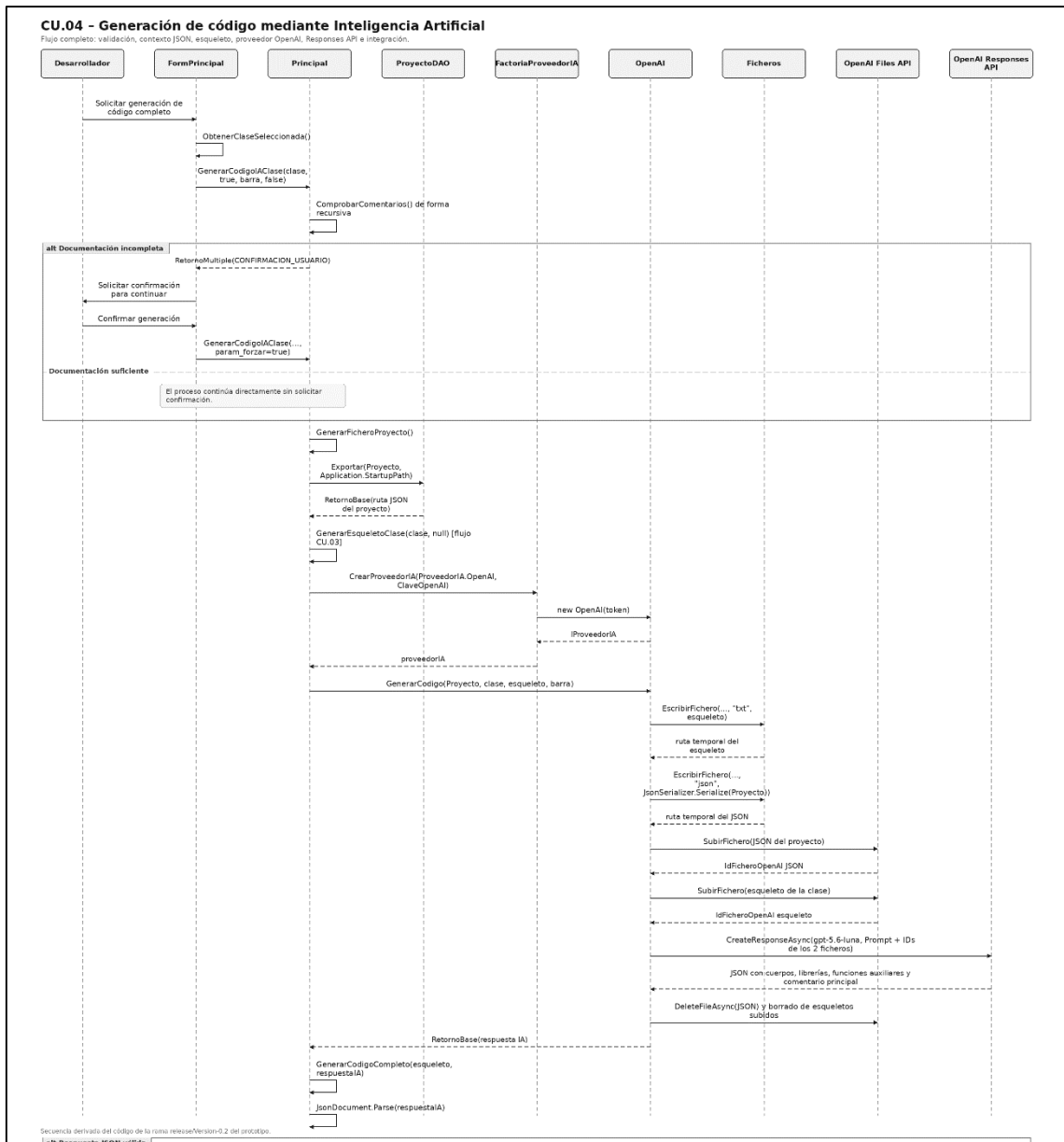


Imagen 15 Caso de uso 4

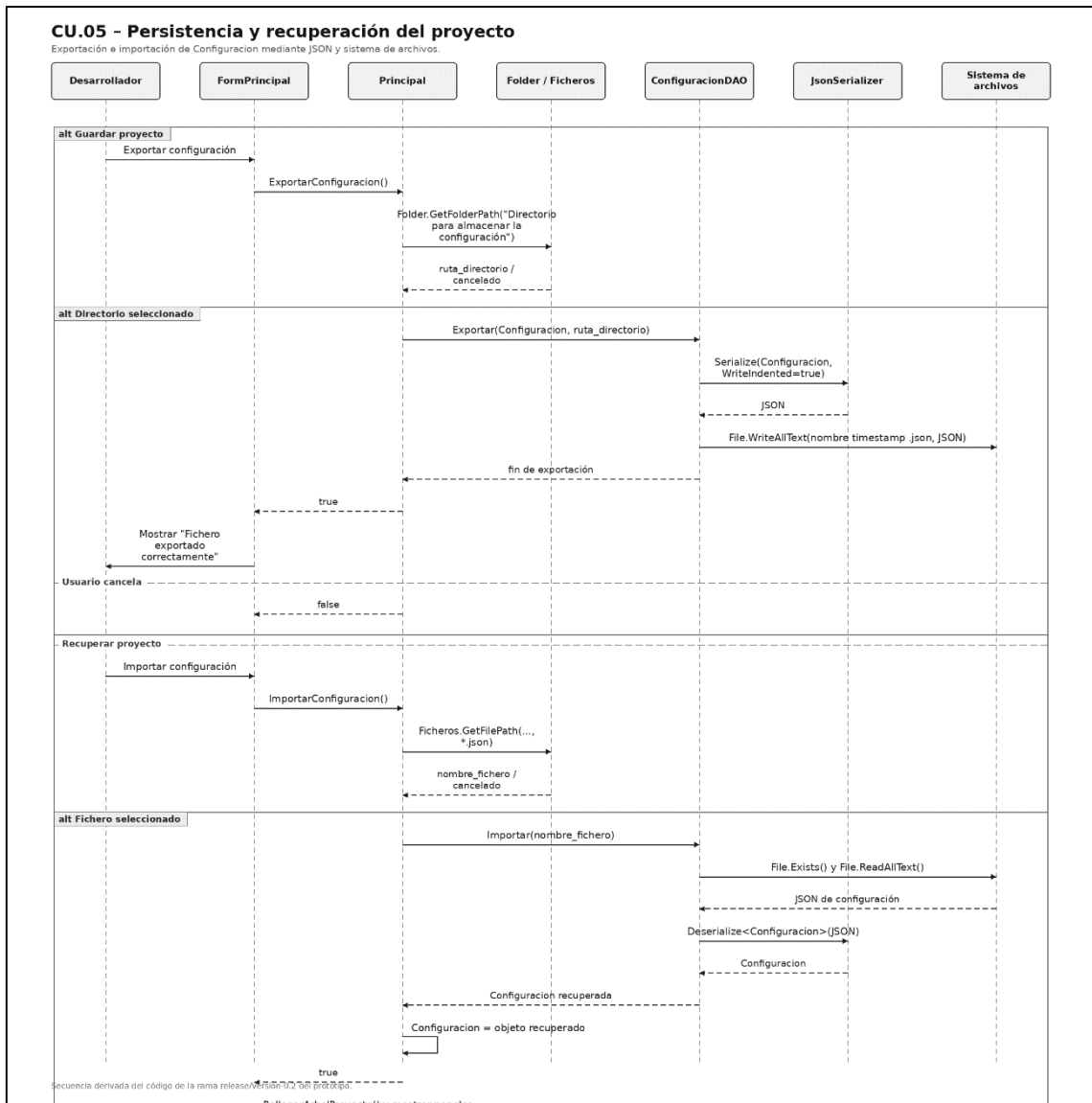


Imagen 16 Caso de uso 5

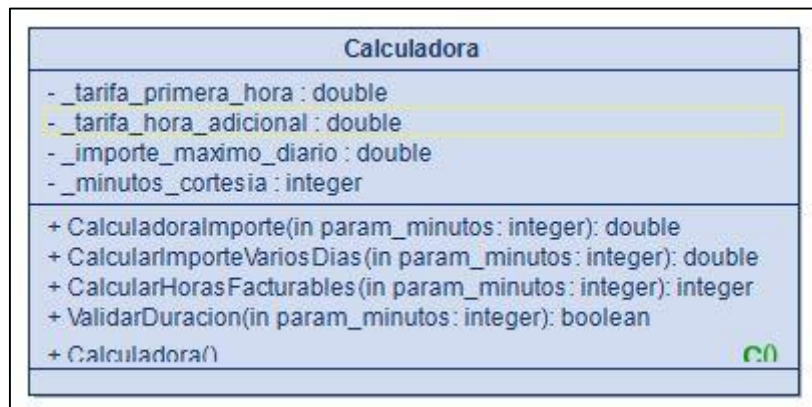


Imagen 17 Prueba simple

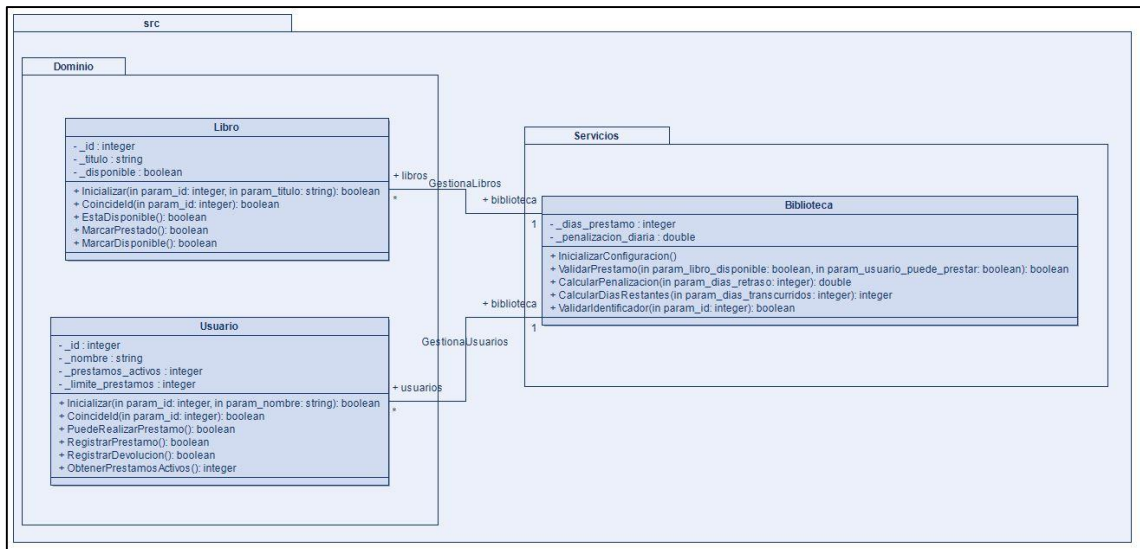


Imagen 18 Prueba media

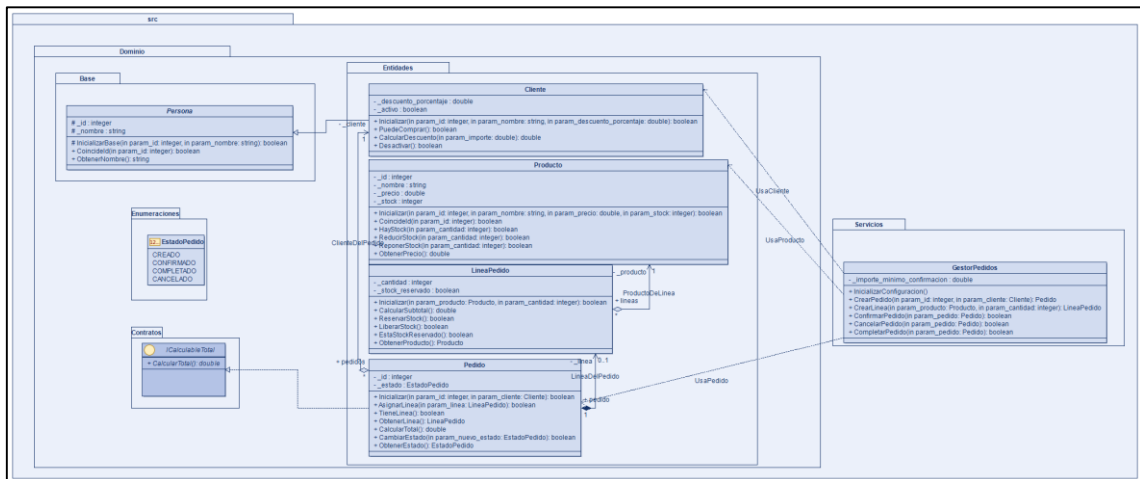


Imagen 19 Prueba completa