UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

ModelForge: Herramienta de modelado conceptual y generación de código
Generación de código y transformación de modelos

ModelForge: Tool for conceptual modelling and code generation
Code generation and model transformations

Realizado por
José Ángel Bueno Ruiz

Tutorizado por
Javier Troya Castilla

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

ModelForge: Herramienta de modelado conceptual y generación de código

Generación de código y transformación de modelos

ModelForge: Tool for conceptual modelling and code generation

Code generation and model transformations

Realizado por
José Ángel Bueno Ruiz

Tutorizado por
Javier Troya Castilla

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2025

Fecha defensa: septiembre de 2025

Resumen

La Ingeniería del Software Dirigida por Modelos (ISDM) busca elevar el nivel de abstracción en el desarrollo de software, facilitando la validación temprana de requisitos y promoviendo la independencia respecto a la plataforma de implementación. Dentro de este enfoque, los modelos conceptuales del dominio expresados con UML y complementados con OCL permiten describir la estructura y restricciones de un sistema de forma precisa. No obstante, las herramientas CASE disponibles presentan limitaciones relevantes, como la falta de integración de OCL, la escasa accesibilidad en entornos académicos o la ausencia de generación automática de código.

Con el fin de dar respuesta a estas carencias, este proyecto conjunto propone el desarrollo de ModelForge, una herramienta CASE orientada al modelado conceptual de diagramas de clase. Se plantea como una solución abierta y extensible que combina un entorno de modelado visual con soporte para OCL y compatibilidad con la herramienta USE, permitiendo que los modelos generados puedan ser validados de forma directa en dicho entorno. De esta manera, se ofrece a estudiantes, investigadores y desarrolladores un recurso intuitivo, accesible y alineado con los principios de la ISDM.

El trabajo realizado en esta parte del proyecto conjunto se ha centrado en el diseño de un Meta-modelo coherente con el subconjunto de UML empleado y OCL, que actúa como base para las transformaciones de Texto a Modelo y de Modelo a Texto. Para su desarrollo se han empleado tecnologías como C++, Qt y ANTLR4, que proporcionan tanto la base de ejecución como los mecanismos de análisis necesarios.

Palabras Clave: UML, OCL, USE, Ingeniería del Software Dirigida por Modelos , Transformaciones Modelo a Texto y Texto a Modelo.

Abstract

Model-Driven Engineering (MDE) aims to raise the level of abstraction in software development, facilitating early requirement validation and promoting independence from the implementation platform. Within this approach, domain conceptual models expressed with UML and complemented by OCL enable an accurate description of a system's structure and constraints. However, existing CASE tools present significant limitations, such as the lack of OCL integration, limited accessibility in academic contexts, and the absence of automatic code generation.

To address these shortcomings, this joint project introduces ModelForge, a CASE tool focused on the conceptual modeling of class diagrams. It is conceived as an open and extensible solution that combines a visual modeling environment with OCL support and compatibility with the USE tool, allowing the models created to be directly validated in that environment. In this way, ModelForge provides students, researchers, and developers with an intuitive and accessible resource aligned with the principles of MDE.

The work carried out in this part of the joint project has focused on the design of a Meta-model consistent with the selected subset of UML and OCL, serving as the foundation for both Text to Model and Model to Text transformations. The implementation relies on technologies such as C++, Qt, and ANTLR4, which provide both the execution framework and the necessary parsing mechanisms.

Keywords: UML, OCL, USE, Model-Driven Engineering, Model To Text and Text to Model Transformations.

ÍNDICE

1

Introducción

En este capítulo se realizará una breve introducción al problema identificado así como a la solución propuesta. Se describen tanto la motivación del proyecto como sus objetivos, además de la metodología de trabajo adoptada, basada en metodologías ágiles adaptadas a un entorno académico. Se ofrece también una visión general de la estructura del documento, indicando la organización de los distintos capítulos y el contenido que en estos se aborda.

1.1. Motivación

En la actualidad, la ISDM (Ingeniería del Software Dirigida por Modelos) es de gran importancia en el campo de la ingeniería de software. Su objetivo es elevar el nivel de abstracción durante el desarrollo de software, acercándose al dominio del problema en lugar de al dominio de la solución. De esta manera, se consigue un diseño independiente de la plataforma donde se implemente, y en el que diferentes stakeholders pueden tomar parte en el proceso de desarrollo.

Dentro de esta metodología destacan particularmente los modelos conceptuales del dominio. En ellos se abstrae la estructura del dominio del problema, describiéndolo en términos de tipos de entidad, tipos de relación y atributos. Con el fin de desarrollar este modelado conceptual se suelen usar lenguajes formales o semi-formales, entre ellos destaca UML (del inglés, *Unified Modelling Language*) [1], lenguaje de modelado referente en el desarrollo de sistemas basados en objetos y respaldado por la OMG (del inglés, *Object Management Group*).

Los anteriormente mencionados modelos conceptuales del dominio pueden expresarse con mayor nivel de detalle mediante diagramas de clase, los cuales introducen elementos como las operaciones o los tipos de atributo, acercándose ligeramente a la implementación del sistema, pero manteniendo un alto nivel de abstracción. Estos diagramas de clase son un tipo de diagrama UML que normalmente se complementa con OCL (del inglés, *Object Constraint Language*)[2], un lenguaje formal que permite detallar restricciones o definir invariantes sobre el sistema modelado. Este lenguaje, también respaldado por la OMG, se integra con UML y abre la puerta a la validación de los modelos diseñados, comprobando que cumplan las condiciones de los dominios modelados.

Al elevar la abstracción gracias a los modelos conceptuales, la ISDM trata de acercarse al lenguaje humano y agilizar así el ciclo de desarrollo. La fase de modelado del sistema también permite la captura y validación de requisitos en fases tempranas del desarrollo, reduciendo los costes causados por errores surgidos en fases tardías. Además, el modelado fomenta buenas prácticas como la reutilización de componentes y la existencia de una documentación actualizada del sistema al completo.

El modelado conceptual también es una herramienta didáctica muy útil en la enseñanza de principios fundamentales de la ingeniería del software ya que proporciona una comprensión más profunda de la abstracción, la modularidad y ayuda a la visualización de los principios y patrones de diseño.

En conjunto, todos estos aspectos ponen en evidencia la necesidad de disponer de herramientas CASE (del inglés *Computer-Aided Software Engineering*) que faciliten la aplicación práctica de la ISDM, apoyando tanto la creación como la validación de modelos conceptuales del dominio. La ausencia de este tipo de entornos limita el potencial de la metodología, mientras que su implementación permite materializar sus beneficios en entornos reales de desarrollo. Precisamente, la importancia de la ISDM y la oportunidad de cubrir esta necesidad constituyen la principal motivación de este proyecto.

1.2. Planteamiento del problema

En la actualidad, se dispone de múltiples herramientas CASE destinadas a este tipo de tareas de modelado. Sin embargo, la mayor parte de ellas no incorpora de manera conjunta funcionalidades clave como la accesibilidad, la generación automática de código o el soporte a la definición de restricciones e invariantes mediante OCL, que aportan contexto al modelo y posibilitan la validación de estados concretos del sistema. Un ejemplo representativo es la herramienta USE (del inglés, *UML-based Specification Environment*), que en realidad se trata de una herramienta de validación de estados del sistema. Esta herramienta permite comprobar instancias del modelo frente a restricciones OCL, pero carece de opciones como la generación de código o un entorno de modelado integrado, ya que la definición de modelos debe realizarse mediante la edición manual de archivos de texto en el formato de la aplicación.

Otro aspecto a tener en cuenta, especialmente en el ámbito académico, es la restricción que supone la necesidad de licencias de pago para determinadas herramientas más completas, como Visual Paradigm o StarUML. Si bien estas soluciones ofrecen entornos de modelado con una interfaz atractiva e intuitiva y con soporte para una gran variedad de tipos de modelos, no incorporan soporte para restricciones OCL, y la necesidad de una licencia limita adicionalmente su accesibilidad en contextos educativos y de investigación.

La solución propuesta al problema expuesto y que se presenta en este documento es ModelForge. Esta solución se plantea como el desarrollo de un herramienta CASE que ofrezca un entorno de modelado integrado intuitivo y con soporte para restricciones OCL y generación de código. Al estar orientada principalmente a al ámbito académico y con el propósito de facilitar la validación de estados del sistema, ModelForge se complementa con la herramienta académica mencionada anteriormente, USE, empleando el mismo formato de archivo. De este modo, los modelos creados en ModelForge pueden ser validados directamente en USE.

Más allá de lo descrito, la herramienta se plantea, al estar desarrollada como software de código abierto, como una solución extensible y adaptable. Esto permite su ampliación y personalización en función de las necesidades específicas de los usuarios, favoreciendo

tanto su evolución continua como su integración en distintos contextos académicos y de investigación.

1.3. Objetivos

El objetivo principal de este proyecto conjunto es desarrollar una herramienta CASE centrada en el ámbito del modelado y diseño de software, en concreto en el modelado de los modelos conceptuales de dominio y diagramas de clase previamente mencionados.

Esta herramienta pretende incorporar las características claves mencionadas anteriormente, proporcionando un entorno de modelado intuitivo y accesible, facilitando la interacción de los usuarios y permitiendo una experiencia clara y eficiente, incluyendo además soporte para la definición de restricciones OCL. Para ello, es necesario algún formato de archivo que permita almacenar información de un modelo conceptual junto con restricciones. La herramienta desarrollada usará el formato de archivo utilizado por USE, ya que además de cumplir con las características necesarias, proporciona interoperabilidad con un entorno de validación ya consolidado y ampliamente usado en el ámbito académico. De esta forma se cumplen dos objetivos, facilitar la validación de instancias de los modelos desarrollados a los usuarios y contribuir a la facilidad de utilización de USE, al ofrecer una alternativa gráfica al modelado textual necesario, mejorando así la experiencia del usuario en todo el proceso de diseño.

Además, otro objetivo es la capacidad de generar de forma automática código fuente partiendo de un modelo, agilizando el flujo de desarrollo al proporcionar una estructura base acorde al diseño del sistema.

Los objetivos de esta rama concreta del proyecto son los enfocados en la transformación de Texto a Modelo y Modelo a Texto, para la carga y guardado de los modelos y la generación de código, así como el desarrollo de un Meta-modelo para almacenar la información de los diagramas de clase diseñados mientras se ejecuta la herramienta.

Por último, y con el fin de asegurar la sostenibilidad del proyecto a largo plazo, se busca diseñar una arquitectura que garantice un sistema extensible y abierto a futuras expansiones. Mediante este diseño basado en los principios fundamentales del desarrollo de software, se facilita que la herramienta pueda evolucionar y adaptarse a

nuevas necesidades a lo largo del tiempo.

1.4. Metodología de trabajo

Para desarrollar este proyecto se ha seguido una metodología ágil, incremental e iterativa, adaptada al ámbito académico y colaborativo del proyecto. Este enfoque ha permitido gestionar el desarrollo de forma flexible, permitiendo adaptarse a a cualquier cambio de requisitos o problema surgido durante el proceso.

Debido al carácter colaborativo del proyecto, se ha optado por una versión simplificada de la metodología ágil SCRUM. De forma convencional, *SCRUM* es un marco de trabajo ágil ampliamente utilizado, que se basa en dividir el desarrollo de un proyecto en ciclos más cortos llamados sprints, que suelen abarcar entre una y cuatro semanas. El objetivo es entregar una versión funcional del producto en cada sprint, de forma que se fomenta la mejora continua y se obtiene retroalimentación lo antes posible. Tradicionalmente también existen figuras como la de Scrum Master o Product Owner, con los que se mantienen reuniones junto al equipo de desarrollo tras finalizar cada sprint.

En este proyecto, se han adaptado los sprints de forma que tuvieran una duración de una semana, donde los desarrolladores han desempeñado además el papel de coordinadores de su propio trabajo, manteniendo reuniones semanales entre ellos y de forma más esporádica, aunque continuada, con el tutor del TFG, que ha actuado de forma similar a un Product Owner. En cada una de estas reuniones se ha discutido el trabajo realizado durante la iteración anterior y lo que se planeaba hacer a continuación, además de mostrar el estado del sistema tras el sprint. Al seguir una metodología incremental, tras cada iteración se aportaba valor a lo desarrollado anteriormente, centrándose siempre en las características más prioritarias, hasta llegar al producto final que cumple con los objetivos propuestos.

Durante todo el desarrollo del proyecto se han mantenido tanto un *Product Backlog* como un *Sprint Backlog*, conceptos propios de la metodología *SCRUM*, que consisten respectivamente en la lista priorizada de todas las tareas y requisitos del proyecto y en el conjunto de elementos seleccionados para ser completados durante un *sprint*.

1.5. Estructura de la memoria

El documento se estructura en siete capítulos, organizados de la siguiente forma:

- *Capítulo 1. Introducción*

  Este capítulo presenta una breve descripción de la motivación y objetivos del proyecto, así como de la metodología escogida para el desarrollo.

- *Capítulo 2. Estado del arte*

  En esta sección se realiza un análisis sobre la situación actual en diferentes campos relacionados con el proyecto.

- *Capítulo 3. Tecnologías empleadas*

  En este capítulo se describen las diferentes tecnologías que han sido utilizadas durante el desarrollo del proyecto, justificando por qué han sido empleadas.

- *Capítulo 4. Diseño*

  Este capítulo explica de forma detallada el diseño del sistema, discutiendo la arquitectura del mismo y las decisiones tomadas al respecto.

- *Capítulo 5. Implementación*

  En esta sección se detalla cómo se ha realizado la implementación de los elementos descritos en el capítulo 4.

- *Capítulo 6. Pruebas y validación*

  En este capítulo se detallan las pruebas y validaciones realizadas sobre el proyecto.

- *Capítulo 7. Conclusión*

  En este último capítulo se exponen las conclusiones obtenidas tras el desarrollo del proyecto y se comentan posibles líneas de investigación en un futuro.

2

Estado del arte

Este capítulo recoge el análisis sobre el estado del arte en diferentes campos relacionados con el proyecto. Se discutirán las bases de cada uno de estos campos, así como los últimos avances y las alternativas de software más extendidas.

2.1. Ingeniería del Software Dirigida por Modelos

ISDM (Ingeniería del Software Dirigida por Modelos) o MDE (del inglés, *Model Driven Engineering*) es una metodología que busca elevar el nivel de abstracción durante el desarrollo de software [3], como ya se hizo en su día con la invención de los lenguajes de alto nivel, agilizando el proceso de desarrollo software.

Ante la constante evolución de las tecnologías, la MDE se basa en el uso de modelos para representar el sistema, buscando abstraer al máximo el diseño de su implementación y evitando la necesidad de conocimientos específicos sobre la plataforma o el lenguaje de desarrollo [4]. Con este mismo objetivo, propone la generación automática de código a partir de los modelos diseñados, convirtiendo al propio modelo en el producto a obtener y proporcionando gran versatilidad al ser agnóstico tanto a la plataforma como al lenguaje de programación.

Una aplicación más concreta de esta disciplina respaldada por la OMG es la MDA (del inglés, *Model Driven Architecture*) [5]. Este enfoque hace uso concretamente de modelos UML y otros estándares del consorcio como OCL para modelar las especificaciones del sistema. Estos paradigmas se basan en el uso de modelos conceptuales, abstracciones de alto nivel que describen las características de un sistema con términos como entidades,

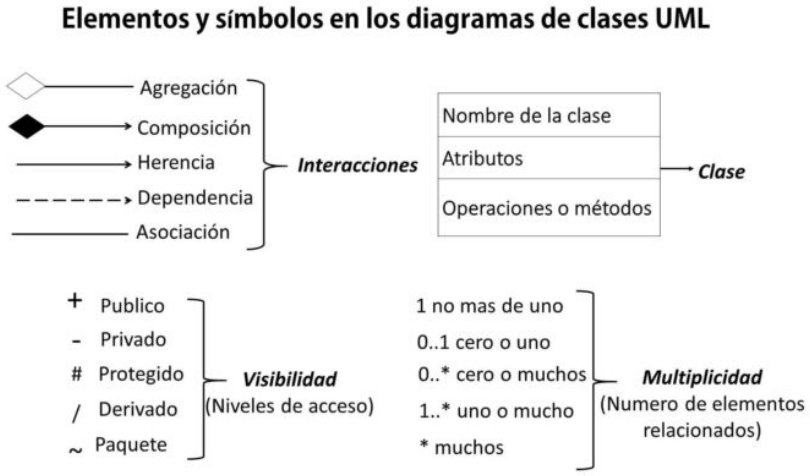


Figura 2.1 Elementos de los diagrams de clase, extraída de [8]

relaciones y restricciones [6].

En relación a este proyecto, es importante destacar los modelos conceptuales de dominio ya mencionados en la Introducción. Estos modelos permiten representar el dominio del problema de forma totalmente independiente a la implementación del sistema, definiendo solamente entidades con una serie de atributos y las relaciones entre estas. En la MDA este tipo de modelos se representan mediante UML y se denominan diagramas de clase [7].

Estos diagramas reducen ligeramente la independencia de la implementación ya que introducen una serie de conceptos que no se tienen en cuenta en el modelo puramente conceptual. La introducción de elementos como las operaciones, los tipos de atributo, las restricciones, la navegabilidad y multiplicidad de las relaciones o la visibilidad de entidades permiten a los diagramas de clase tener un mayor nivel de detalle y mantener un nivel de abstracción elevado, de forma que sigue siendo un artefacto de gran utilidad en el análisis y diseño del sistema.

En este campo destacan diferentes herramientas CASE como:

- **Visual Paradigm [9]:** Herramienta CASE orientada al modelado visual y la gestión de negocio, con multitud de tipos de modelos soportados, como diagramas de clase, diagramas de secuencia, etc. Ofrece funcionalidades avanzadas como la generación de código o la ingeniería inversa, pero no tiene soporte para declarar restricciones sobre el modelo y además requiere una licencia. Además el trabajo

colaborativo se vuelve complejo puesto que, al usar archivos binarios, los sistemas de control de versiones no son de utilidad.

- **StarUML [10]:** Similar a Visual Paradigm, pero centrada exclusivamente en el modelado. Además, ofrece soporte para generación de código y para el proceso de ingeniería inversa -generando el modelo a partir del código fuente- permitiendo que el modelo se mantenga actualizado en todo momento. Es de código abierto pero también requiere licencia.
- **PlantUML [11]:** Herramienta basada en modelado textual, soporta una gran variedad de diagramas sin limitarse a los declarados en UML. Además se trata de una herramienta de acceso gratuito.
- **Enterprise Architect [12]:** Esta herramienta está mas centrada en el uso empresarial. Aunque es ampliamente utilizada por sus extensas funcionalidades como captura de requisitos o generación de código, presenta una interfaz que puede ser confusa y requiere una licencia.

2.2. Validación de modelos

A diferencia de la verificación formal, que se basa en el uso de técnicas matemáticas y lógicas de forma automática como el model checking [13], realizado por herramientas como Alloy Analizer [14], la validación tiene un enfoque más relajado y no formal, centrándose en asegurar que el modelo cumple con las expectativas reales del dominio del problema.

Para poder realizar esta validación, es necesario representar tanto la estructura del sistema de manera adecuada como garantizar que el modelo refleje las reglas y restricciones propias del dominio. Con estos fines, resultan imprescindibles UML [1] y OCL [2], lenguajes estandarizados que permiten expresar tanto la estructura como los requerimientos del sistema.

UML es un lenguaje de modelado estandarizado por la OMG, que proporciona un extenso conjunto de notaciones gráficas para representar un amplio abanico de aspectos de un sistema software. Como se ha mencionado anteriormente, en este campo destaca especialmente el diagrama de clases, ya que permite representar modelos conceptua-

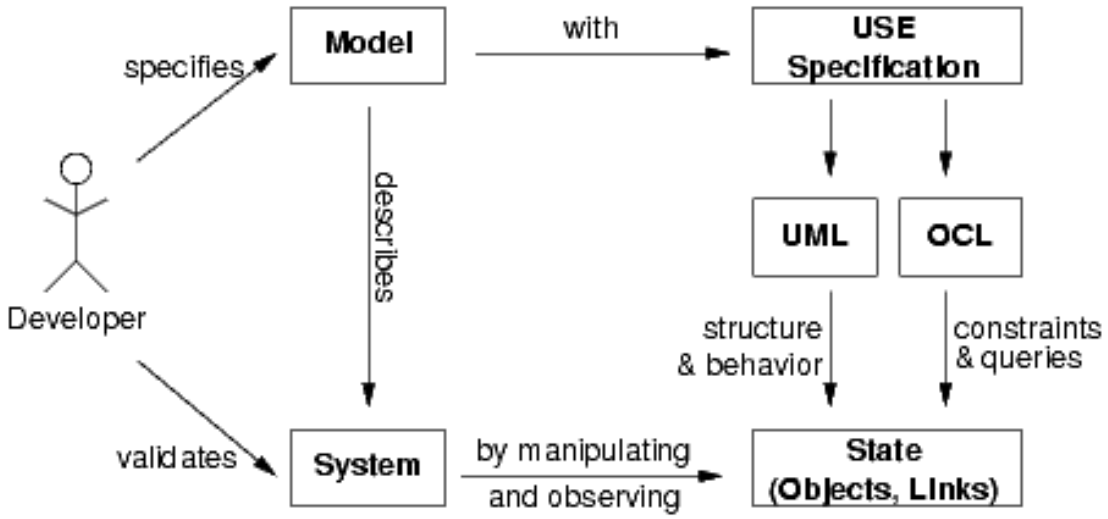


Figura 2.2 Enfoque general de la herramienta USE, extraída de [16]

les mediante la especificación de entidades, atributos, relaciones y operaciones. Sin embargo, UML por sí solo resulta insuficiente para representar todas las restricciones presentes en un sistema, ya que el diagrama de clases se trata de un modelo estructural, y por tanto no tiene la capacidad de expresar las restricciones del dominio.

OCL se estandariza con el objetivo de resolver esta limitación, al tratarse de un lenguaje formal que complementa a UML y permite definir condiciones lógicas de forma precisa y sin ambigüedades. OCL se usa para especificar invariantes de clase, precondiciones y postcondiciones de operaciones, así como restricciones adicionales sobre las asociaciones del modelo. De esta forma, la combinación de ambos lenguajes constituye el marco necesario para llevar a cabo la validación de un modelo.

En este campo destaca USE [15], una herramienta de código abierto usada en el ámbito académico que permite validar las restricciones especificadas sobre diferentes instancias del modelo. Para ello utiliza un lenguaje específico de domino, del que hablaremos más adelante, para definir sus modelos. Además se pueden incluir restricciones OCL así como funcionalidad SOIL(del inglés, *Simple OCL-based Imperative Language*), un lenguaje imperativo propio de USE que permite especificar el comportamiento del sistema usando OCL, operadores y funciones. En la Figura 2.2 se puede apreciar el enfoque seguido por esta herramienta.

USE permite simular diferentes instancias del modelo proporcionado, e incluso simular comportamientos reales mediante máquinas de estado, transiciones y funciones.

Esto permite que, aunque no se trate de una verificación formal, se pueda comprobar el comportamiento del sistema en casos que se esperen en el dominio real.

Si bien se trata de una potente herramienta de validación, la mayor desventaja que presenta USE es la necesidad de realizar el modelado de forma exclusivamente textual, puesto que, pese a estar basado en UML y OCL, no incorpora un entorno visual que facilite la especificación de modelos.

2.3. Lenguajes Específicos de Dominio

Los Lenguajes Específicos de Dominio (DSL, del inglés *Domain-specific Language*) son lenguajes especializados en un dominio concreto, a diferencia de los lenguajes de propósito general. Estos lenguajes se tratan en realidad de una forma de abstracción del dominio de un problema específico [17], y por tanto persiguen la claridad y la simplificación de conceptos para adaptarse al problema que han sido desarrollados para resolver.

Lejos de estar limitados por su especialización, estos lenguajes aprovechan propiedades particulares de un dominio específico para mejorar la calidad y flexibilidad de sistemas software y agilizar el proceso de desarrollo. Este tipo de lenguajes, por lo general, presenta menor complejidad y una curva de aprendizaje más reducida que un lenguaje de propósito general. Además, al emplear un lenguaje específico de dominio, se abstrae la lógica de la aplicación, contribuyendo a facilitar la comprensión y mejorar la productividad de los usuarios que lo utilicen así como a facilitar la comunicación entre desarrolladores y expertos de dominio [18]. Algunos ejemplos ampliamente conocidos y utilizados en la industria son HTML, SQL o LaTeX

En el caso particular de USE se emplea un DSL, en el que profundizaremos más adelante, que recoge de forma textual un subconjunto de UML, que pese a ser un lenguaje visual, se trata de un lenguaje de propósito general. En la Figura 2.3 podemos apreciar un ejemplo simple de como este lenguaje permite describir elementos como clases y asociaciones, entre otras características. OCL por su parte, también se trata de un lenguaje específico de dominio que se centra en la definición formal de restricciones sobre modelos UML.

```
model ExampleModel

class Person
attributes
name : String
age : Integer
operations
constraints
inv Positive_Age:
    self.age > 0
end

class House
end

association Lives_In between
Person [*] role person
House [0..1] role home
end
```

Figura 2.3 Ejemplo de modelo simple usando el lenguaje específico de dominio de USE

2.4. Transformación Modelo a Texto y Texto a Modelo

Como hemos mencionado previamente, la ISDM visualiza el modelo como el producto a desarrollar, siendo este independiente al completo de la implementación. Para que esto sea viable en casos reales, es crucial que una vez generados los modelos, estos puedan traducirse a una implementación real de los mismos.

Con este fin surgen las transformaciones de Modelo a Modelo (M2M, del inglés *Model to Model*), que consisten en acercar un Modelo Independiente de la Plataforma (PIM, del inglés Platform Independent Model) a su implementación real, los llamados PSM (del inglés, Platform Specific Model)[5]. Profundizando aún más en esta idea, resulta evidente la necesidad de contar también con transformaciones de Modelo a Texto (M2T, del inglés *Model to Text*), que permitan generar algún tipo de representación textual a partir de un modelo. Estas transformaciones ofrecen la posibilidad de usar estos artefactos textuales para una larga lista de finalidades, entre las que destacan la serialización de modelos —facilitando su almacenamiento en archivos o una posterior transformación de modelos por medios textuales—, la generación de documentación y, en determinados casos, la producción del propio código fuente [19], cumpliendo el

propósito de la ISDM.

Actualmente, pese a la creación del lenguaje de transformación MOFM2T (del inglés, *MOF Model-To-Text Transformation Language*) [20] por parte de la OMG en 2008, existe una gran variedad de herramientas y lenguajes de este estilo, cada uno con su conjunto de ventajas e inconvenientes y con una interoperabilidad reducida [21].

Tras mencionar la posibilidad de usar los artefactos generados por estas transformaciones como archivos de guardado, resulta igualmente necesario considerar el proceso inverso: las transformaciones de Texto a Modelo (T2M, del inglés *Text to Model*) [22]. En este contexto, destaca el estándar XMI (del inglés, *XML Metadata Interchange*) [23] propuesto por la OMG, ampliamente adoptado como formato de intercambio entre herramientas de modelado. Un ejemplo de estas transformaciones lo encontramos en la propia herramienta USE, que las implementa para cargar los modelos en la herramienta partiendo de las descripciones textuales de los mismos en el DLS propio de USE.

En este campo destaca la gran cantidad de posibilidades que ofrece el reciente auge de la Inteligencia Artificial. Una IA es capaz de recoger una serie de requisitos acerca de un sistema, o la representación de este en algún formato textual a modo de *prompt* y generar un modelo acorde, aunque es cierto que en la actualidad aún se dan casos en los que produce alguna inconsistencia o alucinación [24]. Dada la especificación de un modelo, también es capaz de generar el código correspondiente, y si bien en esta tarea la IA es más precisa por la naturaleza de los LLM (del inglés, *Large Language Model*) y los datos con los que han sido entrenados, no se elimina por completo el problema de la falta de fiabilidad, que aumenta a la par que la complejidad de los modelos [25]. Por estos motivos siguen siendo necesarias herramientas de modelado que permitan generar una especificación determinista, con objeto de usarla por sí sola o incluso como entrada para una Inteligencia Artificial.

3

Tecnologías empleadas

En este capítulo se presentan las principales tecnologías estudiadas y utilizadas para el desarrollo del proyecto. Se presentan junto a una explicación de las mismas, el motivo por el que han sido seleccionadas y su papel dentro del marco general del desarrollo.

3.1. C++ y Qt

C++ es un lenguaje de programación de uso general desarrollado en 1979 como una extensión de C. Una de las principales características que se tuvo en cuenta a la hora de desarrollar el lenguaje es su capacidad para combinar la programación orientada a objetos y la programación estructurada, aportando las ventajas de ambos enfoques [26]. Expande el lenguaje C añadiendo funcionalidades como la herencia, abstracción de datos, polimorfismo o comprobación de tipos, lo que permite a los desarrolladores generar software de mayor calidad, ya que este puede ser más modular y reutilizable.

Este lenguaje es ampliamente utilizado para el desarrollo de todo tipo de aplicaciones: desde sistemas operativos, pasando por aplicaciones de alto rendimiento hasta videojuegos. C++ se ha popularizado gracias a su velocidad y eficiencia, así como a la capacidad que ofrece a los desarrolladores control sobre los recursos del sistema. El precio a pagar por estas ventajas es un alto nivel de exigencia a la hora de interactuar de forma directa con la memoria para evitar errores.

Qt por su parte se trata de un framework desarrollado por el *Qt Group* especializado en el desarrollo de aplicaciones multi-plataforma, con especial enfoque en las interfaces

de usuario [27]. Presenta una gran cantidad de módulos con componentes reutilizables que facilitan el desarrollo de aplicaciones. Qt además cuenta con una extensa documentación y una comunidad activa de usuarios, factores relevantes durante el desarrollo y el mantenimiento del software. Este framework puede emplearse sobre C++, Python o QML, siendo el primero el estándar al facilitar el desarrollo de aplicaciones para un lenguaje con tantas ventajas como C++, al mismo tiempo que trata de resolver algunas de estas con una gestión de memoria más cómoda para el usuario o una sintaxis menos verbosa.

Estas tecnologías han constituido la base principal sobre la cual se ha desarrollado el proyecto debido a todas las ventajas mencionadas, con el objetivo de desarrollar una aplicación eficiente y que disponga de una interfaz intuitiva y funcional. Para aprovechar el potencial conjunto de C++ y Qt se ha empleado *Qt Creator*, un entorno de desarrollo integrado (IDE, del inglés *Integrated Development Environment*) ofrecido por el *Qt Group* que ha sido diseñado específicamente para facilitar el desarrollo de C++ con Qt, ofreciendo todas las herramientas estándar que se esperan en un buen entorno de desarrollo, como auto-completado inteligente o gestión de proyectos con CMake, además de herramientas especializadas para la creación de interfaces gráficas, como un diseñador visual.

3.2. USE: UML-based Specification Environment

Como se ha mencionado en capítulos anteriores, USE se trata de una herramienta para la especificación y validación de modelos, que ofrece la posibilidad de complementar con restricciones e invariantes los modelos descritos [15]. Esta fue desarrollada originalmente en la Universidad de Bremen en el año 1998, y desde sus inicios su propósito principal ha sido ofrecer un entorno práctico para la docencia y la investigación en la Ingeniería del Software Dirigida por Modelos (ISDM), ofreciendo a sus usuarios un medio para experimentar con modelos, validar restricciones y comprobar su consistencia.

A nivel arquitectónico, USE se organiza en torno a tres componentes principales que funcionan de forma conjunta. En primer lugar, dispone de un entorno gráfico limitado que permite interactuar con el segundo componente, su potente motor de validación.

Este constituye el núcleo de la herramienta, ya que es el encargado de interpretar los modelos definidos de forma textual y realizar la validación de las restricciones sobre instancias de estos. Finalmente, cuenta con soporte para ejecutar comandos SOIL, permitiendo interactuar con los diagramas de objetos de los modelos que se están validando de forma directa [16].

Los archivos con los que trabaja esta herramienta combinan el lenguaje específico de dominio de USE con OCL y SOIL, almacenando en un mismo documento la especificación del modelo y las restricciones que se le aplican. Este se trata del principal motivo por el que se ha optado por emplear el formato de archivo de USE como formato de guardado en ModelForge, ya que proporciona una base idónea para los objetivos del proyecto. Otros factores que se han tenido en cuenta han sido tanto sus potentes capacidades como herramienta de validación como su extendido uso en el ámbito académico, puesto que de esta manera, los modelos diseñados usando ModelForge pueden cargarse directamente en USE, proporcionando una mejor experiencia al usuario gracias a esta interoperabilidad.

3.3. ANTLR4: ANother Tool for Language Recognition

ANTLR4 (del inglés, *ANother Tool for Language Recognition*) es un potente generador de analizadores. Dada la descripción formal de un lenguaje, llamada gramática, ANTLR es capaz de generar analizadores léxicos y sintácticos que tienen la habilidad de interpretar el lenguaje descrito.

En una gramática formal se define como la descripción de un lenguaje. En esta se detallan de forma estructurada los símbolos, reglas y relaciones que permiten interpretar de forma correcta las cadenas de entrada [28]. Con esta definición, ANTLR es capaz de generar un analizador léxico (en inglés, *lexer*).

El análisis léxico se trata del proceso de agrupar caracteres individuales en palabras o símbolos, también llamados *tokens*. Siguiendo lo definido en la gramática, los *tokens* que genera el *lexer* se componen del tipo de símbolo del que se trata y del texto que ha sido reconocido como tal.

Una vez realizado el análisis léxico, el analizador sintáctico (en inglés, *parser*) puede

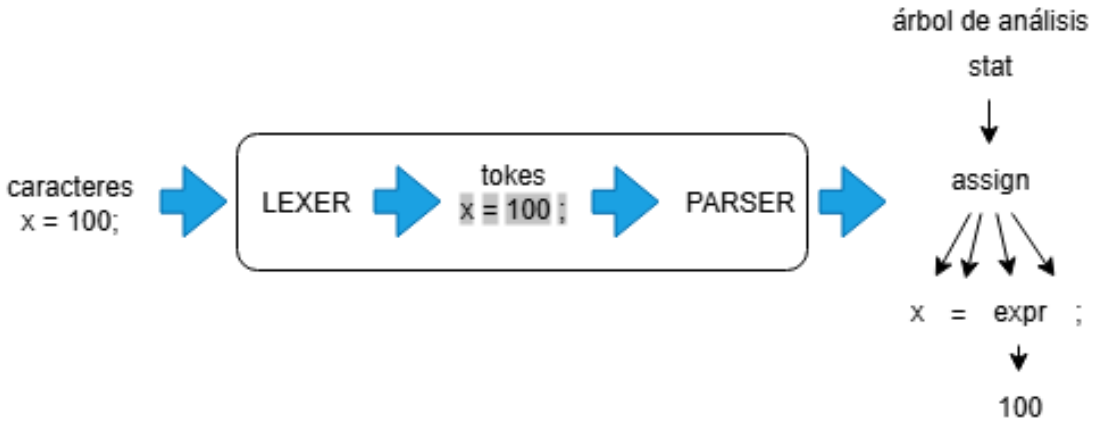


Figura 3.1 Proceso de interpretación

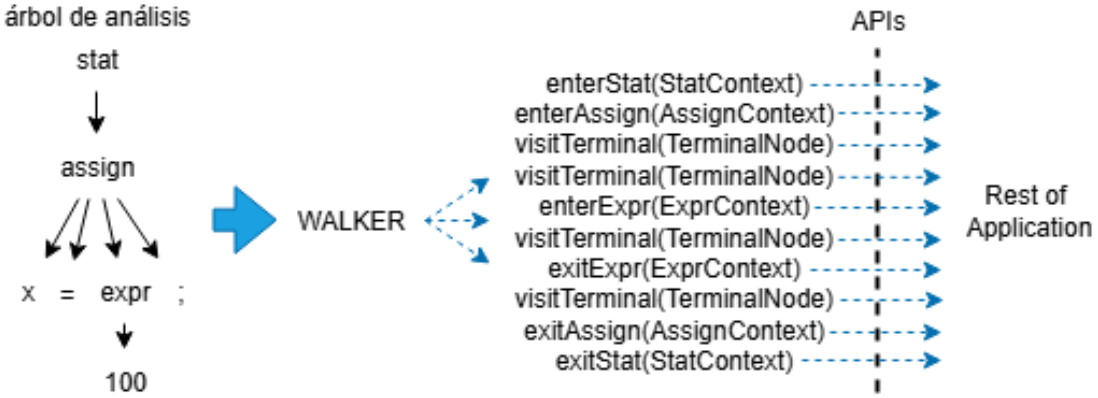


Figura 3.2 Funcionamiento del *listener* de ANTLR4

llevar a cabo el análisis sintáctico. Este consiste en reconocer la estructura de las reglas descritas que corresponden a los símbolos detectados. Como resultado de este proceso, se genera un árbol de análisis, una estructura de datos que almacena cómo el *parser* ha reconocido la estructura de la cadena de entrada. Tal y como se aprecia en la Figura 3.1, en este árbol cada nodo interno representa una regla concreta, mientras que la raíz se trata de la propia cadena de entrada completa y las hojas de los *tokens* reconocidos por el *lexer* [29].

ANTLR también proporciona mecanismos para navegar estas estructuras de datos mediante los patrones de diseño *listener* y *visitor*. Por un lado, se puede optar por usar un *listener* que recibe eventos producidos al visitar el árbol (ver Figura 3.2). El *visitor* por su parte permite controlar de forma precisa el recorrido del árbol de análisis, decidiendo qué acciones tomar en cada tipo de nodo y si continuar recorriendo el árbol o no (ver Figura 3.3).

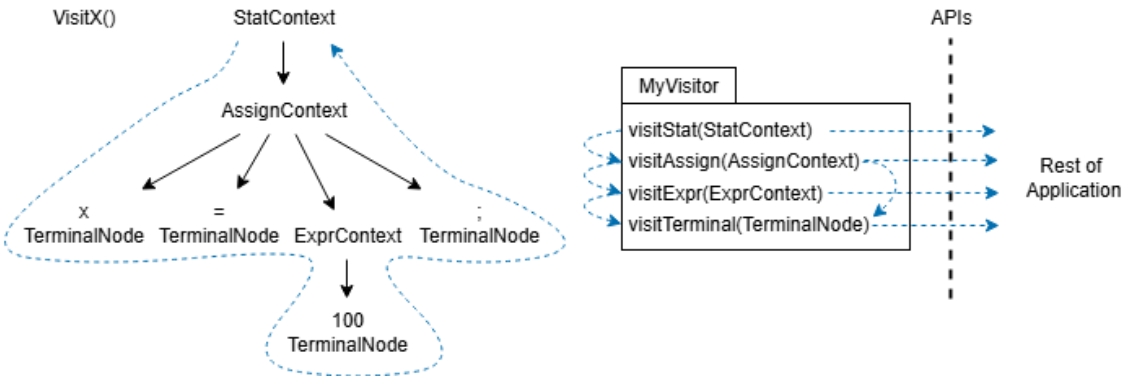


Figura 3.3 Funcionamiento del *visitor* de ANTLR4

En el contexto del proyecto, esta herramienta se ha utilizado para generar analizadores utilizando una versión modificada de la gramática de USE. De esta forma, usando el *visitor* proporcionado por ANTLR se obtiene la capacidad de procesar archivos en formato USE y convertirlos en un metamodelo interno de ModelForge, garantizando que la representación textual se transforme de forma correcta y controlada, siendo consistente con la estructura de datos reconocida por la herramienta.

3.4. Java

Java se trata de un lenguaje de programación de carácter general como una plataforma en sí mismo. Este fue creado en 1995 por *Sun Mycrosystem* y se caracteriza por la portabilidad que ofrece con la Máquina Virtual de Java (en inglés JVM, *Java Virtual Machine*), permitiendo que los programas se ejecuten en cualquier dispositivo [30]. Además, Java cuenta con una comunidad de usuarios extensa y activa, lo que favorece la disponibilidad de una gran cantidad de herramientas para el programador que lo utilice.

Otra de las principales características de Java, y la razón por la que se ha utilizado en este proyecto es su sólida orientación a objetos, permitiendo representar de manera directa las clases, atributos, operaciones, relaciones y demás definidas en los modelos. Esta correspondencia entre los elementos del modelo y las estructuras del lenguaje ha sido la razón por la que se ha seleccionado Java como lenguaje de generación de código, ya que asegura que la implementación sea coherente con el diseño conceptual.

3.5. Visual Paradigm

Como se ha comentado previamente, Visual Paradigm es una plataforma que ofrece un amplio abanico de herramientas que permiten crear una gran variedad de modelos y diagramas diferentes, orientados a mejorar la gestión y el desarrollo de proyectos software [9].

En este proyecto se ha utilizado principalmente para realizar los diagramas de componentes y de clases que se describirán en los próximos capítulos. Ha permitido diseñar los modelos conceptuales necesarios para el desarrollo del proyecto, como los Meta-modelos de OCL o del subconjunto de UML empleado.

3.6. Git

Git es un sistema distribuido de control de versiones extensamente utilizado en la industria debido a las utilidades que ofrece. Entre ellas destacan su control de versiones y trazabilidad, manteniendo un historial completo de todos los cambios realizados en el proyecto, lo que permite un flujo de desarrollo mucho más cómodo y ágil, y facilita la identificación de errores [31].

Al tratarse de un proyecto conjunto, Git se ha empleado para gestionar el código fuente durante el desarrollo, facilitando la colaboración y agilizando el proceso.

3.7. Trello

Trello es una herramienta de gestión de proyectos basada en el uso de tableros Kanban, ofreciendo una interfaz basada en tarjetas que permite organizar las tareas de manera clara en diferentes columnas. Cada tarea puede incluir descripciones, subtareas o comentarios, proporcionando diversos métodos que propician una buena organización y comunicación. Las tareas además pueden marcarse con diferentes etiquetas, facilitando así su organización [32].

Esta herramienta se ha utilizado para poner en práctica la metodología ágil seguida, permitiendo organizar y priorizar las tareas de forma acorde a los *sprints*, y distribuir

el trabajo entre los desarrolladores de forma clara y sencilla.

4

Diseño

En este capítulo se recoge la especificación a alto nivel del sistema, así como una discusión acerca del diseño mismo. Se detallarán las decisiones de diseño tomadas, exponiendo sus características y justificando su elección, pero sin llegar a entrar en detalles de la implementación, que se reserva para el siguiente capítulo. También se comentarán las limitaciones y problemas encontrados durante el desarrollo y las soluciones que se le han dado a los mismos.

4.1. Planteamiento

Para comenzar a hablar acerca del diseño del sistema, en primer lugar es necesario determinar las características del mismo. Si nos centramos en los problemas que pretende resolver esta herramienta, los requisitos principales de la misma resultan evidentes.

En primer lugar, es necesario ofrecer una interfaz de usuario sólida, que permita al usuario realizar el modelado de forma intuitiva y simple. Esta interfaz deberá ofrecer medios para interactuar con las diferentes entidades del modelo, permitiendo crear y modificar clases y asociarlas entre sí. Del mismo modo, el usuario también deberá disponer de alguna manera de definir y modificar restricciones OCL sobre el sistema modelado.

Para soportar este entorno de modelado es indispensable un modelo de datos que se adapte al subconjunto de UML utilizado. Este modelo contendrá los datos de los diagramas de clase diseñados por los usuarios, de forma que se trata de un Meta-

modelo, una representación de los elementos del propio modelo: clases, asociaciones, enumeraciones, etc. Los modelos desarrollados serán por tanto una instancia de este Meta-modelo, por lo que debe estar compuesto por los elementos necesarios para almacenar los datos y ofrecer una interfaz de forma que la interacción con el mismo sea lo más sencilla posible.

En este contexto cobra importancia la transformación de modelos sobre la que se ha discutido en capítulos anteriores. Es necesario realizar una transformación de Texto a Modelo para ser capaces de popular el Meta-modelo de datos en el que se apoya la aplicación a partir de un archivo USE, por lo que es indispensable ser capaces de interpretar este formato de archivos, de la misma forma que lo hace la herramienta USE para evitar incoherencias. De igual forma, es esencial la capacidad de almacenar los modelos diseñados en este mismo formato, por lo que se vuelve necesaria también la transformación de Modelo a Texto.

Adicionalmente, para cumplir con la visión de la ISDM (Ingeniería del Software Dirigida por Modelos) del modelo como producto a desarrollar, se necesita la capacidad de generar código fuente a raíz del modelo. Para esta tarea vuelven a cobrar importancia las transformaciones de Modelo a Texto, teniendo en cuenta las necesidades de Java: diferentes ficheros para cada clase, sintaxis del lenguaje, representación de las asociaciones, etc.

Una vez determinadas las principales características del sistema, el resultado de la captura de requisitos es el siguiente:

- **RF1 -** El sistema debe ser capaz de transformar archivos USE en modelos.
- **RF2 -** El sistema debe ofrecer herramientas de modelado intuitivas.
- **RF3 -** El sistema debe ofrecer al usuario herramientas para la especificación de restricciones OCL.
- **RF4 -** El sistema debe ser capaz de transformar instancias del Meta-modelo en archivos USE.
- **RF5 -** El sistema debe ser capaz de transformar instancias del Meta-modelo en código Java.

4.2. Arquitectura

El diseño arquitectónico del sistema refleja los componentes necesarios para cubrir las necesidades descritas en el apartado anterior. Como se muestra en la figura 4.1, el sistema se divide en tres componentes principales, cada uno implementado en su propio paquete. Con el objetivo de mantener una estructura clara, los paquetes se han diseñado de forma aislada e independiente en la medida de lo posible, favoreciendo así la reusabilidad de los componentes y la futura extensibilidad de la herramienta. Esta arquitectura permite que cada módulo pueda evolucionar o ser sustituido teniendo un impacto mínimo sobre el sistema en conjunto, asegurando una mayor mantenibilidad y adaptabilidad del proyecto.

Sin entrar en demasiados detalles sobre el paquete de la interfaz gráfica, puesto que no constituye el foco principal de esta memoria y se aborda en la memoria conjunta realizada por mi compañero, este se compone de una serie de vistas distintas que permiten al usuario interactuar con el modelo. Para su desarrollo se han empleado las diversas herramientas ofrecidas por el *framework* Qt. Entre ellas destaca el uso del patrón arquitectónico Modelo-Vista [33], una variante del clásico patrón Modelo-Vista-Controlador adaptada por Qt. En este enfoque simplificado el modelo encapsula los datos y la interfaz es responsable de mostrarlos y ofrecer mecanismos de interacción al usuario, separando la gestión de los datos de su representación visual y respetando el principio de separación de preocupaciones [34]. Este patrón se aplica a la hora de representar los datos del Meta-modelo, con una serie de elementos destinados a visualizar el modelo en sí mismo, manteniendo una referencia a las clases del Meta-modelo correspondientes.

Además dentro de este paquete también se ha empleado el patrón Comando [35], un patrón de comportamiento cuya función principal es encapsular cada acción del usuario en un objeto independiente. De este modo se logra desacoplar la invocación de la operación de su ejecución concreta y ha permitido la implementación de funcionalidades para hacer y deshacer acciones determinadas.

El paquete *MetaModel Library* constituye el núcleo de la herramienta, al contener todos los elementos necesarios para definir correctamente el Meta-modelo junto a

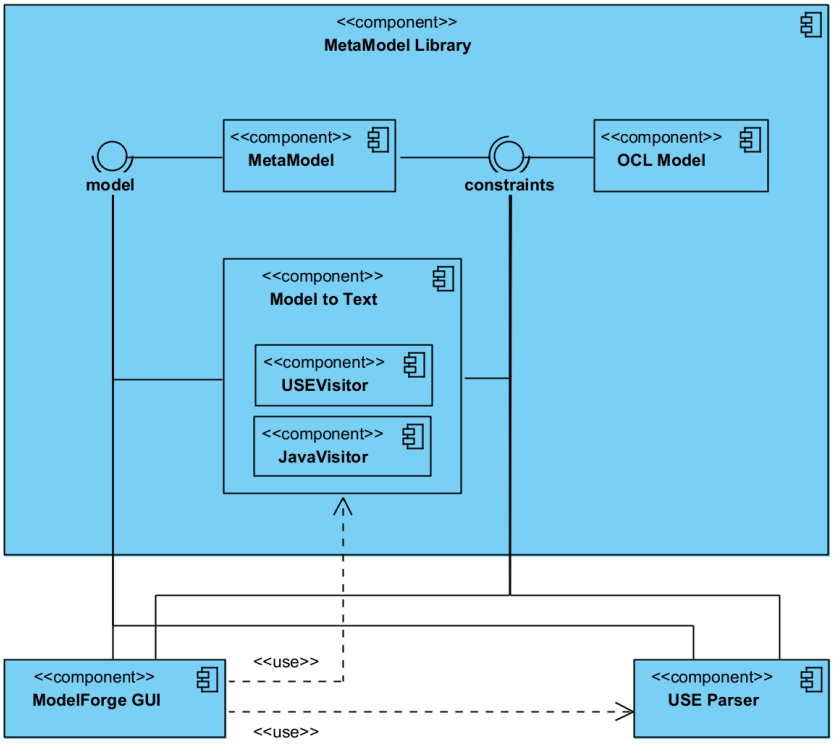


Figura 4.1 Diagram de componentes de ModelForge

las restricciones OCL. Está compuesto por tres sub-componentes distintos. En primer lugar, el paquete *MetaModel*, que incluye las clases fundamentales para representar los elementos básicos de los modelos como *MetaClass*, *MetaAttribute* o *MetaAssociation*. Estas clases no se limitan a almacenar información sobre el modelo, sino que ofrecen una API interna que permite la interacción. Ofrecen métodos mediante los que se puede consultar o modificar la información del Meta-modelo, de forma que este no solo actúa como una representación pasiva del diseño, sino como un componente activo que facilita la manipulación de modelos.

De forma similar, el paquete *OCL Model* incluye las clases necesarias para representar todo el rango de expresiones admitidas por este lenguaje, tales como operaciones, navegaciones, condicionales o expresiones lógicas. Estas estructuras permiten construir una representación interna de las restricciones definidas sobre los modelos. Este componente se integra con el Meta-modelo, puesto que las restricciones se definen sobre elementos concretos como clases u operaciones en forma de invariantes o precondiciones y postcondiciones respectivamente, y estas tienen una expresión OCL asociada.

Finalmente el paquete *Model To Text* engloba las herramientas necesarias para realizar transformaciones de Modelo a Texto a partir de las instancias del Meta-modelo. Para llevar a cabo estas transformaciones se emplea el patrón de comportamiento Visitante [**visitor-pattern**], que permite desacoplar los algoritmos de los objetos sobre los que operan. Al aplicar este enfoque, la mayoría de elementos del Meta-modelo implementan un método que les permite aceptar a un *visitor* pero no es necesario modificarlos para añadir nueva lógica u operaciones sobre ellos. Se establece una interfaz *BaseVisitor*, donde se definen los métodos de visita a los diferentes elementos del Meta-modelo, y serán las diferentes implementaciones las que determinen el comportamiento. Para el desarrollo de este proyecto se han implementado un *USEVisitor* y *JavaVisitor*, que se encargan de visitar los elementos del Meta-modelo y realizar las transformaciones necesarias. El uso de este patrón facilita además la implementación de nuevas transformaciones, ya que permite definir de manera modular cómo debe traducirse cada tipo de elemento sin modificar la estructura del Meta-modelo. Esto garantiza una arquitectura flexible y extensible a largo plazo, preparada para incorporar transformaciones Modelo a Texto a nuevos lenguajes o formatos de salida con un esfuerzo mínimo.

La *MetaModel Library* ha sido diseñada de manera totalmente independiente al *framework* Qt y los componentes de ANTLR, lo que minimiza las dependencias externas y maximiza su reusabilidad. Esta independencia permite que la estructura de datos del Meta-modelo pueda usarse en otros proyectos o contextos sin necesidad de arrastrar componentes de la interfaz gráfica o dependencias innecesarias. Además, al estar desacoplada, facilita el mantenimiento y la evolución del sistema. De esta manera se consigue una estructura modular y flexible, alineada con los principios del buen diseño software.

Por último, el paquete llamado *USE Parser* es el encargado de interpretar un archivo USE y popular los datos de una instancia del Meta-modelo. Este componente engloba tanto los analizadores léxicos y sintácticos como los diferentes archivos que conforman el árbol de análisis y el *visitor* correspondiente generados por ANTLR4, lo que permite convertir un texto de entrada en esta estructura de datos en tiempo de ejecución. Además de permitir generar este árbol, el *parser* constituye la base para la verificación inicial de la corrección sintáctica de los modelos y facilita su posterior procesamiento.

```
model Modelo

end
```

Listing 1 Definición de modelo en USE

Finalmente este paquete también contiene una implementación del *visitor* del árbol de análisis. En ella se define la lógica que controla el recorrido de los distintos nodos del árbol y se establecen los pasos necesarios para ir construyendo las estructuras del Meta-modelo, transformando la representación textual en elementos con significado dentro del sistema.

4.3. Modelo conceptual

Al constituir el núcleo de la aplicación, en esta sección se profundiza en el diseño de los paquetes del Meta-modelo y del modelo OCL, detallando el modelo conceptual realizado y las decisiones de diseño tomadas.

4.3.1. Meta Modelo

Comenzando por el modelo conceptual del Meta-modelo, la mayor consideración ha sido representar fielmente los diferentes elementos que permite especificar la herramienta USE. Como se aprecia en la FIGURA, el resultado final del modelado se trata de un subconjunto de UML con todos los Meta-elementos necesarios para definir un diagrama de clases. A continuación, se describen los principales elementos que lo integran y se muestra cómo la sintaxis textual de USE se traduce en los elementos correspondientes dentro del diagrama de clases así como las decisiones de diseño adoptadas para su representación en el sistema. El modelo conceptual del Meta-modelo se encuentra en el apéndice A debido a su extensión.

Modelo

En USE el modelo se declara con la palabra clave *model* seguida del nombre del modelo (ver Listing 1), y contiene la especificación del resto de elementos.

En consecuencia, en el diseño del sistema el modelo se refleja como una entidad —representada como *MetaModel*— que contiene el nombre del mismo, asociada con

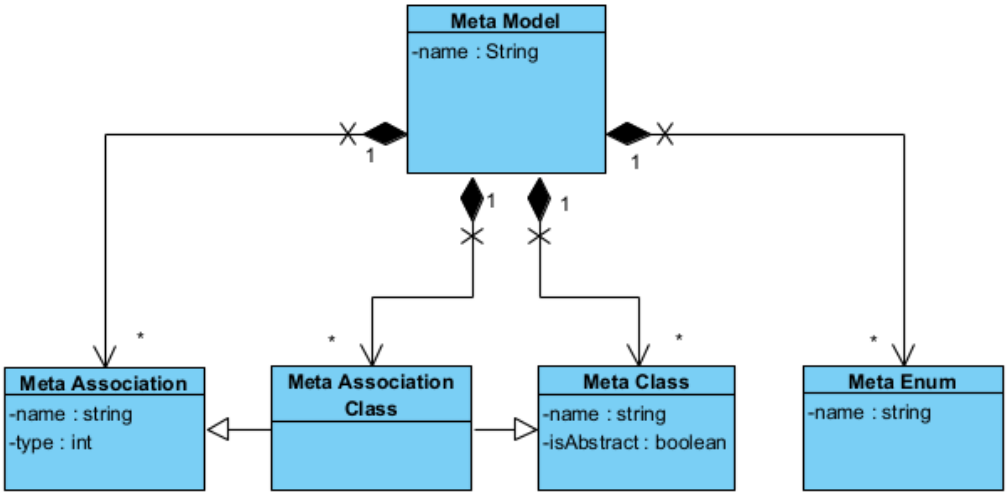


Figura 4.2 Modelado del Meta-modelo

```
enum Enumerado {Elemento1, Elemento2, Elemento3}
```

Listing 2 Definición de enumerado en USE

el resto de Meta-elementos que puede contener, como *MetaClass*, *MetaAssociation* y demás, como se observa en la Figura 4.2.

Enumerado

El enumerado se define con la palabra clave *enum* en USE, seguido del nombre del mismo y un listado entre llaves de sus diferentes elementos, separados por comas, como se muestra en el Listing 2.

Por tanto, se ha modelado *MetaEnum* como una entidad cuyo atributo es su nombre, y asociada a *MetaEnumElement* para representar los elementos del enumerado, como podemos ver en la Figura 4.3.

Clase

En la sintaxis de USE las clases se definen con la palabra *class*, seguida de un nombre

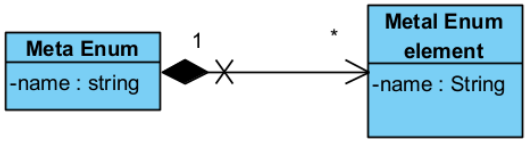


Figura 4.3 Modelado del Meta-enumerado

```
class Persona
attributes
nombre : String
edad : Integer
operations
esMayorDeEdad() : Boolean
constraints
edadPositiva: self.edad > 0
end
```

Listing 3 Definición de clase en USE

de clase no vacío. A continuación se pueden especificar los diferentes atributos de la clase, declarando su nombre, tipo y, opcionalmente, expresiones OCl adicionales si se trata de un atributo derivado o con un valor inicial. Posteriormente se definen las operaciones indicando su nombre, parámetros de entrada, tipo de retorno y, de forma opcional, tanto una declaración de la operación en lenguaje OCL o SOIL como precondiciones y postcondiciones usando expresiones OCl de nuevo. Por último se declaran las restricciones sobre la clase, una colección de invariantes con un nombre específico y una expresión OCL.

Adicionalmente se puede declarar una clase como abstracta mediante la palabra *abstract* o implementar relaciones de generalización y especialización mediante el símbolo "<".

Teniendo en cuenta todas estas propiedades durante el modelado del sistema, la entidad *MetaClass* se define con atributos que representan su nombre y si se trata de una clase abstracta, y mantiene asociaciones con los Meta-elementos que representan el resto de entidades que la componen. Estos Meta-elementos, como *MetaAttribute*, *MetaOperation* o *MetaConstraint* también modelan las propiedades de las entidades que representan, como se aprecia en la Figura 4.4.

Cabe destacar que USE permite definir máquinas de estado dentro de las clases, aunque debido al contexto y los objetivos del proyecto estas no han sido tenidas en cuenta, forman parte del Meta-modelo con la intención de permitir la interoperabiliad

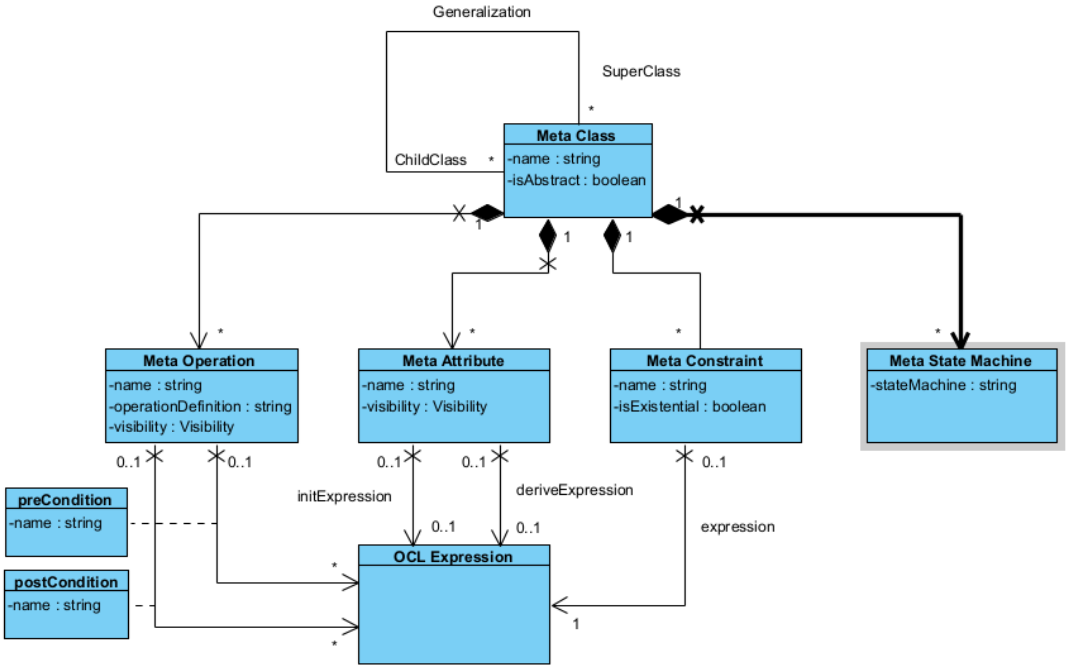


Figura 4.4 Modelado de la Meta-clase

```
abstract class SuperClase
end


class Clase < SuperClase
end
```

Listing 4 Definición de generalización en USE

entre las herramientas.

Como detalle adicional, se ha tomado la decisión de diseño de asociar *MetaClass* con la entidad *MetaAssociationEnd*, representando esta relación las diferentes clases a las que se puede navegar desde una clase determinada a través de las asociaciones que la involucran.

Asociación

Una asociación en USE se define mediante las palabras clave *association*,*aggregation* o *composition*, dependiendo de si se trata de una asociación simple, una agregación o una composición. Seguidamente se declaran los diferentes extremos de la asociación, así como sus roles y multiplicidades.

```
association Paternidad between
Persona [1] role Padre
Persona [0..*] role Hijo
end
```

Listing 5 Definición de asociación en USE

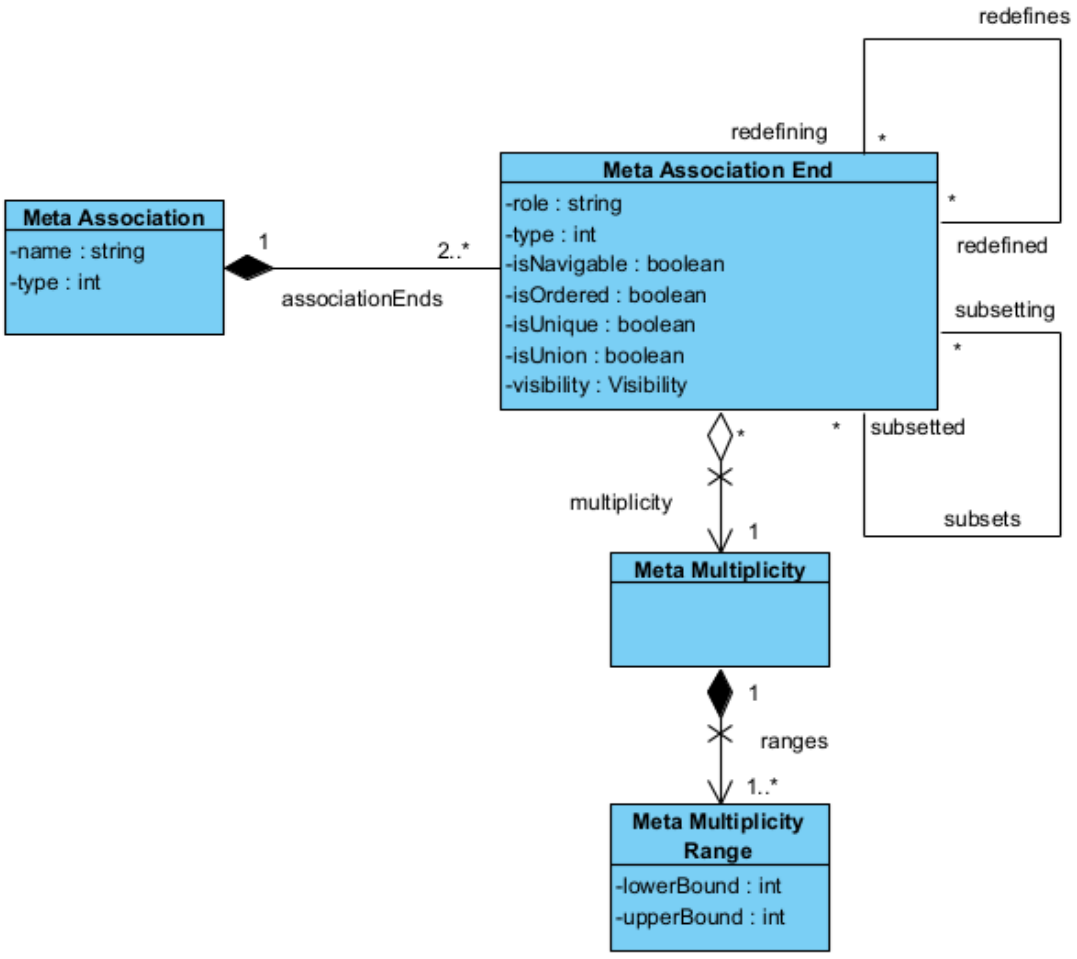


Figura 4.5 Modelado de la Meta-asociación

```
associationclass Contrato between
Empresa [0..1] role Contratador
Persona [1..*] role Contratado
attributes
salario : Real
end
```

Listing 6 Definición de clase asociación en USE

Con el objetivo de representar estos elementos en el diagrama de clases, *MetaAssociation* se ha modelado como una entidad cuyos atributos son su nombre y tipo. Como se muestra en la Figura 4.5, está asociada a *MetaAssociationEnd*, un Meta-elemento que representa los extremos de la asociación y por tanto recoge toda la información necesaria al respecto, como el rol, si los elementos del extremo son únicos u ordenados, entre otros. También está asociado tanto consigo mismo, representando los extremos de relaciones que redefine o de los que es un subconjunto, como con *MetaClass*, representando cual es la clase asociada. Por último, se relaciona con *MetaMultiplicity*, indicando la multiplicidad del extremo de la asociación.

De esta manera se captura toda la información necesaria relacionada con las asociaciones, un elemento clave en el modelado conceptual.

Clase Asociación

Las clases asociación en USE se definen mediante la clave *associationclass* y, como su nombre indica, combinan elementos de las asociaciones y las clases.

Estas entidades permiten especificar características propias de la relación entre elementos. Al tratarse de una combinación de las clases y las asociaciones, a nivel de diseño, como se observa en la Figura 4.6 se ha modelado *MetaAssociationClass* como una entidad que mantiene una relación herencia tanto con *MetaClass* como con *MetaAssociation*, generalizando ambas.

Tipo

Al tratar sobre tipos, es necesario distinguir entre diferentes categorías. En primer lugar, se encuentran los tipos simples, estos se refieren a los tipos primitivos como podrían ser *Integer*, *String* o *Boolean*. A estos se suman los enumerados y las clases —y

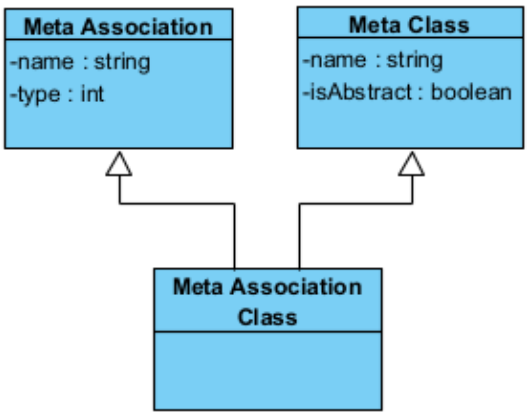


Figura 4.6 Modelado de la Meta-clase-asociación

por extensión, las clases asociación—, que también pueden actuar como tipos simples.

En segundo lugar, existen los tipos de colección, que representan conjuntos de elementos de tipo simple con distintas propiedades, como listas ordenadas o colecciones sin duplicados. Si bien en el modelado conceptual se considera una buena práctica expresar estos tipos mediante relaciones entre entidades, puesto que conllevan un acercamiento del diseño a la implementación, USE permite su definición explícita, por lo que se han incorporado al Meta-modelo con el fin de mantener la interoperabilidad con la herramienta.

Por último, las tuplas en USE no se limitan a representar parejas de valores, sino que constituyen estructuras de datos compuestas en las que es posible definir múltiples atributos con su correspondiente tipo. De este modo, una tupla puede utilizarse para encapsular en un único elemento información acerca de un tipo de datos que, de otra forma, requeriría varias declaraciones independientes.

En consecuencia, todos estos tipos se han modelado como especializaciones de la entidad *MetaType*, siguiendo relaciones de generalización tal y como se muestra en la Figura 4.7. Esta decisión de diseño permite abstraer los diferentes tipos, simplificando su representación así como su tratamiento dentro del sistema.

4.3.2. OCL

En cuanto al modelo conceptual de OCL, el principal objetivo ha sido organizar los diferentes tipos de expresiones en un esquema jerárquico claro, clasificándolos en

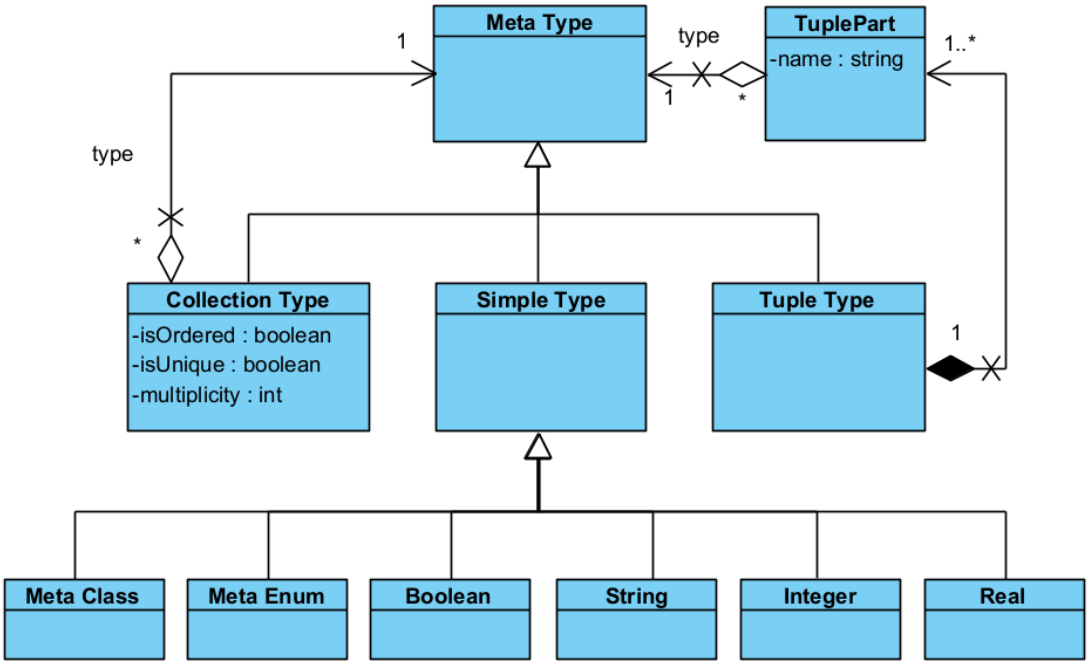


Figura 4.7 Modelado de los Meta-tipos

distintas categorías vinculadas mediante relaciones de especialización y generalización. De esta manera se consigue reflejar la totalidad del lenguaje al tiempo que se mantiene una estructura modular y sencilla. Debido a sus dimensiones, el modelo completo se encuentra anexado en el apéndice B.

Para facilitar la integración de las expresiones se hace uso de las capacidades del polimorfismo, como base del modelo se define una abstracción común que agrupa a todas las expresiones y sus características comunes, como el tipo de dato resultante de la expresión OCL o si está envuelta en paréntesis o no. A partir de esta entidad se derivan subtipos que representan diferentes subgrupos de expresiones, como el conjunto de expresiones binarias o las expresiones de acceso a propiedades. Cada una de estas categorías introduce sus propias características comunes dentro del subgrupo, pero hereda de la jerarquía superior las propiedades compartidas.

Si bien en algunos casos podría haberse optado por una representación más compacta —por ejemplo, modelando todas las expresiones binarias en una única entidad y distinguiéndolas únicamente mediante un atributo que indicase el operador—, se ha decidido además modelar cada tipo de expresión como una entidad independiente, tal y como se muestra en la Figura 4.8 . De esta forma se mejora la claridad del modelo al hacer explícitas las particularidades de cada tipo de expresión, evitando sobrecargar

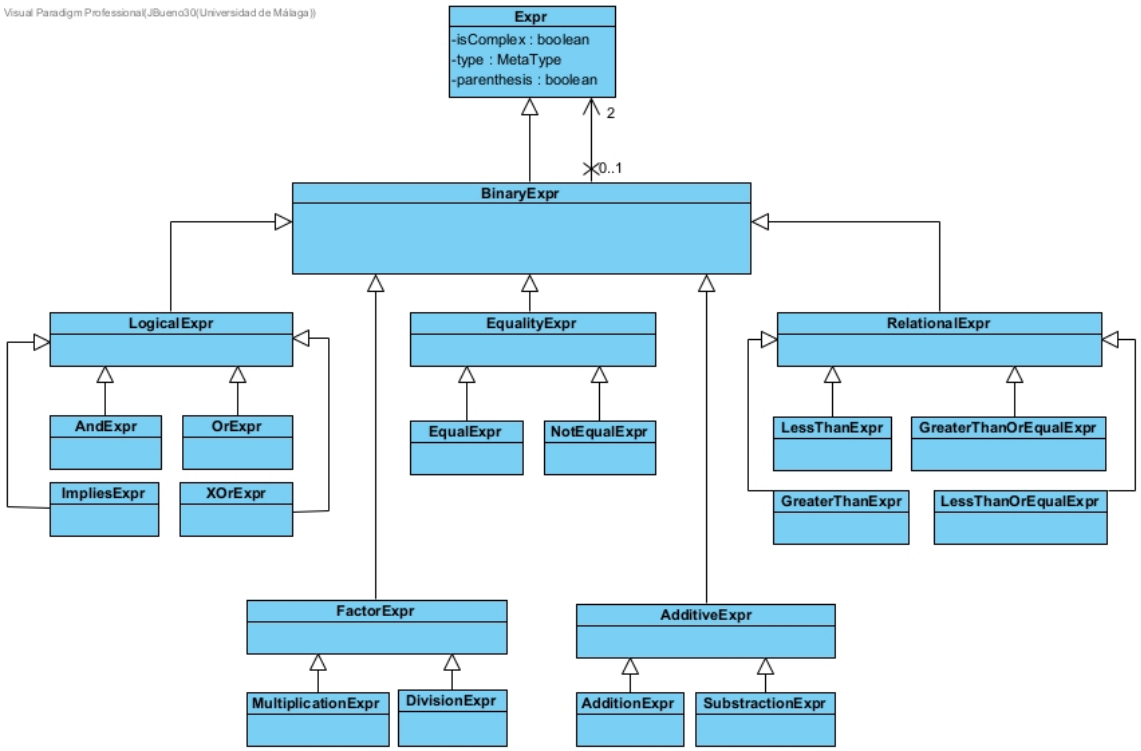


Figura 4.8 Modelado de las expresiones OCL binarias

una sola clase con múltiples responsabilidades. Además, facilita el recorrido de estas estructuras con patrones como *Visitor*, al permitir un tratamiento diferenciado y directo de cada tipo de expresión en lugar de depender de condicionales basados en atributos. También mejora la sostenibilidad, puesto que cada tipo de expresión puede evolucionar de manera aislada sin afectar al resto.

Al optar por este diseño se captura la naturaleza recursiva de OCL, donde una expresión puede contener otras expresiones. Al mismo tiempo, se facilita su integración con el Meta-modelo, ya que los elementos de este relacionados con expresiones, como las restricciones o precondiciones y postcondiciones entre otros, pueden asociarse a una expresión abstracta común gracias al polimorfismo, proporcionando uniformidad y flexibilidad a la representación.

4.4. Limitaciones y soluciones

Durante el proceso de diseño y modelado del sistema se detectaron algunas limitaciones derivadas del propio lenguaje USE, que condicionaban la expresividad del sistema y su alineación con UML. En particular, la herramienta no ofrece soporte de

forma nativa para conceptos como la visibilidad de los elementos o la navegabilidad.

Si bien la definición de ambos conceptos supone un ligero acercamiento a la implementación del sistema, en el contexto de esta herramienta resulta justificado. Al estar orientada también a la generación automática de código, disponer de información de visibilidad en atributos, operaciones y asociaciones, así como de navegabilidad en estas últimas, permite obtener una traducción más fiel al transformar el modelo en texto. No obstante, estas características se han planteado como opcionales dentro del diseño, de modo que el modelado pueda mantenerse en un nivel puramente conceptual cuando así se desee.

En el caso de la visibilidad, se ha optado por incorporarla únicamente en atributos, operaciones y asociaciones. La razón es que estos son los elementos donde esta información resulta realmente práctica y contribuye a la claridad y comprensión del diagrama de clases.

Para incorporar estos conceptos, ha sido necesario extender la gramática original de USE. Esta modificación se diseñó de forma retrocompatible, de modo que los archivos de guardado donde se definan estas propiedades continúan siendo válidos en la herramienta USE.

Por otra parte, algunos conceptos propios de USE y OCL se han simplificado debido a no ser necesarios para los objetivos de ModelForge o por su excesiva complejidad. Las máquinas de estado o las operaciones SOIL de USE, utilizadas en la validación, se conservan en el modelo para no perder la información y mantener la interoperabilidad, pero se ignoran en el procesamiento habitual y no se ofrecen herramientas para interactuar con ellas. De igual forma, las relaciones se han limitado únicamente a dos extremos de asociación para reducir la complejidad.

Así mismo, las expresiones OCL más complejas o que carecen de sentido fuera del contexto de la validación, como las *QueyExpressions*, se identifican mediante un atributo en la clase base de las expresiones, lo que permite tratarlas de forma diferente en operaciones específicas, por ejemplo al popular el Meta-modelo de datos o durante la generación de código.

5

Implementación

Este capítulo se centra en los aspectos técnicos y la implementación de los diferentes aspectos de la herramienta ModelForge que conciernen a esta rama del proyecto. Se abordará la implementación de sus componentes principales y se detallarán las decisiones tomadas para resolver las dificultades surgidas durante el desarrollo.

5.1. Meta-modelo

En cuanto a la implementación del Meta-modelo descrito en el capítulo anterior, cada entidad se ha definido en archivos separados de encabezado y de implementación (`.h` y `.cpp`), lo que permite una clara separación entre la interfaz pública y su lógica interna. Esta metodología permite trabajar con las clases de forma modular y facilita futuras modificaciones o ampliaciones.

En términos generales, las relaciones entre entidades se materializan mediante colecciones de elementos, empleando listas o mapas según las necesidades de cada relación. Todas las colecciones hacen uso de punteros compartidos de C++, como se observa en el Listing 7, garantizando la correcta gestión de la memoria y la reutilización de objetos dentro del Meta-modelo. En aquellos casos donde la relación tiene multiplicidad singular, se ha empleado un único puntero para implementarla, reflejando la cardinalidad definida en el diagrama de clases.

De forma complementaria, la implementación incorpora mecanismos que aseguran el cumplimiento de ciertas restricciones generales de los modelos. Para determinados elementos del modelo, como clases dentro de un mismo modelo, atributos de una

clase o elementos de un enumerado, entre otros, se evita la existencia de duplicados, preservando la consistencia del Meta-modelo según lo establecido por UML. Para lograr esto de manera eficiente, muchas de estas colecciones se implementan como mapas, como se ha mencionado previamente, facilitando tanto la verificación de unicidad como el acceso rápido a los elementos. Esto permite que, incluso a nivel de código, el modelo conserve su validez estructural.

Además, la implementación sigue las relaciones de generalización definidas en el diseño del Meta-modelo. Cada clase base y subclase refleja la jerarquía conceptual establecida, aprovechando el polimorfismo para permitir que las operaciones comunes se definan en niveles superiores mientras que las particularidades de cada subclase se implementan de manera específica.

```
class MetaEnum : public SimpleType, public MetaElement{
private:
    std::string name;
    std::map<std::string, std::shared_ptr<MetaEnumElement>> elements;
public:
    MetaEnum(const std::string& name,
    std::shared_ptr<MetaEnumElement> element=nullptr);

    std::string getName() const;
    void setName(const std::string& name);

    const std::map<std::string, std::shared_ptr<MetaEnumElement>>&
    getElements() const;
    std::shared_ptr<MetaEnumElement> getElement(const std::string& key);
    void addElement(std::shared_ptr<MetaEnumElement> element);
    void removeElement(const std::string& key);

    virtual bool equals(const MetaType& type) const override;

    std::string toString() const override;

    std::any accept(ModelToText::VisitorInterface& visitor) const override;
};
```

Listing 7 Definición de la clase MetaEnum

Cada clase dispone además de métodos que permiten consultar y modificar sus atributos, así como gestionar las asociaciones con otras entidades, garantizando un

acceso controlado y coherente a los datos del modelo, tal como se ejemplifica en el Listing 7. Esta implementación facilita la construcción del Meta-modelo y proporciona una interfaz que puede ser utilizada por los distintos componentes del sistema —en el caso de ModelForge, por la propia interfaz de usuario— para interactuar de manera directa con los elementos del modelo.

Este enfoque proporciona una representación flexible de las distintas entidades del Meta-modelo y sienta una base sólida para operaciones posteriores, como las transformaciones de Texto a Modelo o de Modelo a Texto. Al emplear únicamente mecanismos propios de C++, se asegura la independencia de la implementación y se facilita su potencial reutilización en otros contextos sin necesidad de depender de bibliotecas externas, como ya se mencionó en el capítulo anterior.

5.2. Transformación de Texto a Modelo

La implementación de la transformación de Texto a Modelo, que permite cargar los datos especificados en los archivos `.use`, ha consistido principalmente tanto en las modificaciones realizadas a la gramática para sortear las limitaciones expuestas en el capítulo anterior como en la implementación de un *Visitor* personalizado del árbol de análisis generado por ANTLR4.

5.2.1. Gramática USE y compilación

La gramática de USE utilizada en este proyecto se construye a partir de la especificación oficial proporcionada por la Universidad de Bremen [**GRAMATICA-USE**], la cual contaba con una definición formal compatible con una versión anterior de ANTLR. Sin embargo, para el desarrollo de este proyecto, dicha gramática se ha adaptado a ANTLR4, aprovechando sus mejoras en la generación de analizadores léxicos y sintácticos. A diferencia de versiones anteriores, donde algunas operaciones se definían directamente dentro de la propia gramática —lo que resultaba poco intuitivo y mezclaba responsabilidades—, ANTLR4 separa de forma más clara este tipo de responsabilidades y genera de forma automática herramientas que permiten recorrer el árbol de análisis.

En cuanto a las adiciones realizadas a la gramática, para incorporar los conceptos de

```
NO_NAVIGABLE : '--X';
PUBLIC : '--+';
PRIVATE : '---';
PROTECTED : '--#';
PACKAGE : '--~';
```

Listing 8 Tokens añadidos a la gramática

```
class Persona
attributes
nombre : String --+
operations
esMayorDeEdad() : Boolean --#
end
```

Listing 9 Especificación textual de clase con visibilidades

visibilidad y navegabilidad, se han definido nuevos *tokens*, asegurando que los archivos que los incorporen sigan siendo totalmente compatibles con la herramienta USE. Como se observa en el Listing 8, se han definido de forma que en la versión anterior de la gramática estos elementos se interpretan simplemente como comentarios, por lo que se mantiene la retrocompatibilidad.

Posteriormente, haciendo uso de estos nuevos *tokens* se han creado reglas en la gramática o modificado las ya existentes para añadir la definición de estos conceptos. En el Listing 9 se puede observar la especificación de una clase en un archivo .use con el añadido de la visibilidad.

Gracias a estas nuevas reglas, durante la visita del árbol de análisis es posible extraer dicha información y trasladarla al Meta-modelo, asignando por defecto la visibilidad pública y la navegabilidad permitida en los casos en que no se especifiquen de manera explícita.

5.2.2. Instanciación del Meta-modelo

La instanciación del Meta-modelo se lleva a cabo mediante la implementación de la interfaz *Visitor* generada automáticamente por ANTLR4. De esta manera es posible

controlar de forma minuciosa el recorrido del árbol de análisis generado por el *lexer* y el *parser*.

```
std::any visitAttributeDefinition(
    USEParser::AttributeDefinitionContext *ctx) override {
        std::string name = ctx->ID()->getText();
        std::shared_ptr<MetaModel::MetaType> type =
        std::any_cast<std::shared_ptr<MetaModel::MetaType>>(
                                                    visit(ctx->type()));

        MetaModel::Visibility visibility = MetaModel::Visibility::Public;

        if(ctx->visibilty()){
            visibility =
            std::any_cast<MetaModel::Visibility>(visit(ctx->visibilty()));
        }

        std::shared_ptr<MetaModel::MetaAttribute> attribute =
        std::make_shared<MetaModel::MetaAttribute>(name, type, visibility);

        if(ctx->initDefinition()){
            this->deferredAttributes.push_back(
            {attribute,
            this->currentClassContext,
            ctx->initDefinition()->expression(),
            true});
        }else if(ctx->derivedDefinition()){
            this->deferredAttributes.push_back(
            {attribute,
            this->currentClassContext,
            ctx->derivedDefinition()->expression(),
            false});
        }
        return attribute;
    }
```

Listing 10 Método de visita de la regla que especifica la definición de atributos

En este proceso, se aprovecha la estructura jerárquica del árbol, puesto que cada método del *Visitor* se encarga de traducir una regla concreta de la gramática en la construcción de un elemento del Meta-modelo. Así, las decisiones sobre qué tipo de entidad instanciar se derivan directamente de las reglas sintácticas, mientras que la información específica que se almacena —como nombres o modificadores— se extrae a

partir de los *tokens* identificados por el analizador léxico, como se observa en el Listing 10.

De forma complementaria, durante la construcción se comprueban las restricciones semánticas generales del modelo, que no pueden ser validadas únicamente por el *parser* ni aplicarse de manera aislada a cada entidad. Estos criterios semánticos van más allá de la mera estructura del archivo y son específicos de los elementos que se definen, asegurando, por ejemplo, que al especificar relaciones de herencia o asociaciones en un modelo USE, no se haga referencia a clases u otros elementos que no hayan sido previamente definidos dentro del propio modelo, preservando así su coherencia desde el momento de su instanciación.

Finalmente, el control preciso que otorga el uso de un *Visitor* propio resulta especialmente útil para gestionar dependencias entre distintos elementos: por ejemplo, ha permitido instanciar primero todas las entidades estructurales del modelo y posponer la visita de las expresiones OCL hasta que el contexto esté completamente definido. De este modo, se ha garantizado que cada expresión se procese en un entorno semántico completo y consistente.

A modo de ejemplo de lo expuesto, el Listing 10 muestra la implementación del método de visita correspondiente a la regla que especifica la definición de atributos en el formato USE. En él se observa cómo se extrae el nombre del atributo directamente del *token* correspondiente y cómo se obtiene su tipo mediante la llamada recursiva al *Visitor*, asegurando que la entidad correspondiente del Meta-modelo ya esté instanciada y disponible. La visibilidad se asigna por defecto a pública y se instancia, de nuevo mediante la visita de la regla concreta, en caso de estar especificada. Además, se distingue entre atributos inicializados y derivados, almacenando ambos en la estructura `deferredAttributes`, que permite posponer la evaluación de expresiones hasta que el contexto de la clase esté completamente definido. Se ha seguido esta estrategia durante toda la implementación ya que facilita un control preciso sobre el orden de construcción del modelo y garantiza que todas las restricciones semánticas se evalúen en un entorno completo.

```
class VisitorInterface{

public:
    virtual std::any visit(const MetaModel::MetaModel& metaModel) = 0;

    virtual std::any visit(const MetaModel::MetaEnum& metaEnum) = 0;

    virtual std::any visit(const MetaModel::MetaClass& metaClass) = 0;

    virtual std::any visit(
    const MetaModel::MetaAssociation& metaAssociation) = 0;

    virtual std::any visit(
    const MetaModel::MetaAssociationClass& metaAssociationClass) = 0;

    virtual std::any visit(const MetaModel::MetaAttribute& metaAttribute) = 0;

    virtual std::any visit(const MetaModel::MetaOperation& metaOperation) = 0;

    virtual std::any visit(
    const MetaModel::MetaConstraint& metaConstraint) = 0;

    virtual std::any visit(
    const MetaModel::MetaAssociationEnd& metaAssociatonEnd) = 0;

    virtual std::any visit(const MetaModel::SimpleType& simpleType) = 0;

    virtual std::any visit(
    const MetaModel::CollectionType& collectionType) = 0;

    virtual std::any visit(const MetaModel::TupleType& tupleType) = 0;
};
```

Listing 11 Interfaz del *Visitor* para las transformaciones de Modelo a Texto

```
std::any VisitorUSE::visit(const MetaModel::MetaEnum& metaEnum){
    std::string metaEnumString = "enum " + metaEnum.getName() + " {";

    auto metaEnumElements = metaEnum.getElements();
    auto elementsIterator = metaEnumElements.begin();

    metaEnumString += elementsIterator->second->getName();

    elementsIterator++;

    for(; elementsIterator != metaEnumElements.end(); elementsIterator++){
        metaEnumString += ", " + elementsIterator->second->getName();
    }

    metaEnumString += "}";

    outFile << metaEnumString << "\n\n";

    return 0;
}
```

Listing 12 Método de visita a las entidades *MetaEnun* en *VisitorUSE*

5.3. Transformación de Modelo a Texto

Para la implementación de Modelo a Texto se ha hecho uso exclusivamente del patrón *Visitor*, aprovechando la estructura del Meta-modelo y su jerarquía de clases. Las entidades visitables heredan de una clase abstracta común que define el método de aceptación del *Visitor*, de manera que cada entidad lo implementa llamando al método del *Visitor* correspondiente, con la propia entidad como argumento. De esta forma, la transformación M2T permite generar automáticamente representaciones textuales del modelo, ya sea en forma de archivos `.use` compatibles con la herramienta USE o de código fuente en Java.

Se ha definido también una interfaz para los diferentes *Visitors* que se muestra en el Listing 11, de forma que se declaran todos los métodos de visita que cada implementación concreta tendrá la responsabilidad de definir.

Por tanto, toda la lógica de la transformación y sus particularidades se encapsula dentro de la implementación del *Visitor*. En el caso de *VisitorUSE*, encargado de realizar

```
std::any VisitorJava::visit(const MetaModel::MetaEnum& metaEnum) {
    std::string filePath =
    this->directoryPath + "/" + metaEnum.getName() + ".java";
    std::ofstream outFile(filePath);

    outFile << "package " << packageName << ";\n";

    outFile << "public enum " << metaEnum.getName() << " {\n";
    for(const auto &metaEnumPair: metaEnum.getElements()){
        outFile << "\t" << metaEnumPair.second->getName() << ",\n";
    }
    outFile << "}\n";
    outFile.close();

    return 0;
}
```

Listing 13 Método de visita a las entidades *MetaEnun* en *VisitorJava*

la transformación del modelo en un archivo `.use`, el proceso comienza con la creación del archivo con el nombre correspondiente al modelo, para posteriormente recorrer de manera controlada los diferentes elementos, generando su representación textual a partir de sus datos y escribiéndola en el archivo. Como ejemplo, el Listing 12 muestra la implementación de la visita a una instancia de *MetaEnum*, en la que se construye la cadena que representa la enumeración y se escribe directamente en el archivo de salida.

De manera análoga, *VisitorJava*, responsable de la generación de código fuente en Java, se ha implementado de forma que siga las prácticas habituales del lenguaje. Cada clase se genera en un archivo independiente, con sus respectivos atributos y operaciones, mientras que las relaciones se materializan mediante colecciones y atributos dentro de las clases participantes, preservando la estructura modelada. En el Listing 13 puede observarse la visita a *MetaEnum* en la generación de código Java, en la que se crea el archivo pertinente y se define la enumeración con la sintaxis estándar del lenguaje y la inclusión de todos sus elementos. Además, se generan de forma automática métodos de acceso y modificación —como *getters*, *setters*, *adders* y *removers*— para los atributos y las colecciones, los cuales se incluyen de forma automática dentro de los constructores para su empleo durante la inicialización. Estos métodos también incorporan la

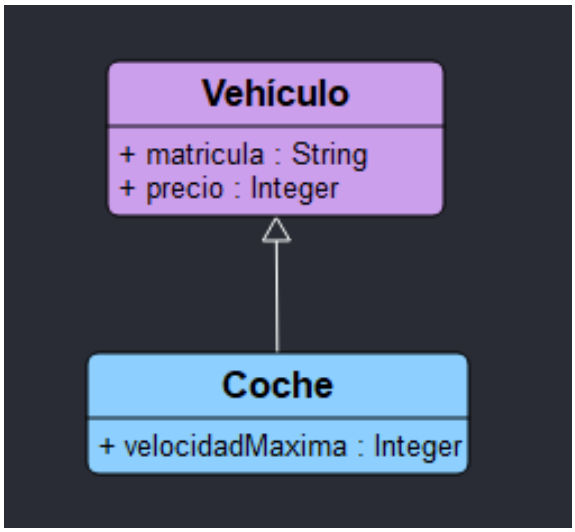


Figura 5.1 Ejemplo de generalización simple modelado en ModelForge

verificación de restricciones estructurales como la multiplicidad de las asociaciones. Las operaciones definidas también se generan como métodos dentro de la clase, donde la implementación y las precondiciones y postcondiciones, en caso de haberse definido, se representan como comentarios para aportar contexto al desarrollador.

Las relaciones de herencia también se reflejan, no solo mediante su declaración, sino modificando el constructor de la clase derivada para inicializar adecuadamente la superclase, como se puede observar al modelar el diagrama simple mostrado en la Figura 5.1 y generar el código obtenido en el Listing 14

5.3.1. Transformación de restricciones OCL

En la implementación del patrón *Visitor* para la transformación de Modelo a Texto, el sistema no se ha extendido para recorrer las expresiones OCL de manera explícita. Esta decisión se ha tomado debido a la complejidad añadida que habría supuesto, además de no ser estrictamente necesaria puesto que cada expresión ya dispone de su propio método `toString()`, capaz de generar la representación textual correspondiente en OCL.

En el caso de *VisitorUSE*, este se limita a insertar las expresiones OCL en la ubicación apropiada dentro del archivo `.use`, respetando la relación con los elementos del modelo sin necesidad de recorrer cada subexpresión de manera individual.

Por otra parte, en la generación de código Java mediante *VisitorJava*, las restricciones

```java
public class Coche extends Vehiculo{

        private Integer velocidadMaxima;

        public Coche(String matricula, Integer precio,
                     Integer velocidadMaxima) {
                super(matricula, precio);
                this.setVelocidadMaxima(velocidadMaxima);
        }

        public Integer getVelocidadMaxima() {
                return this.velocidadMaxima;
        }

        public void setVelocidadMaxima(Integer velocidadMaxima) {
                this.velocidadMaxima = velocidadMaxima;
        }
}
```

Listing 14 Código generado de forma automática a partir del modelo de la Figura 5.1

OCL identificadas como complejas se incluyen como comentarios generales dentro de la clase, de manera que aporten contexto al desarrollador que extienda el sistema en caso de ser necesario. En contraste, las restricciones que hacen referencia únicamente a un atributo específico se integran directamente en el *setter* correspondiente, permitiendo que la validación ocurra de forma automática al modificar el valor del atributo. Este criterio asegura un equilibrio entre la fidelidad al modelo original y la generación de código funcional y mantenible.

Por otra parte, en la generación de código Java mediante *VisitorJava*, las restricciones OCL identificadas como complejas se incluyen como comentarios generales dentro de la clase, de manera que aporten contexto al desarrollador que extienda el sistema en caso de ser necesario. Para las restricciones que no se consideran complejas, se realiza un análisis adicional, y en caso de hacer referencia a varios atributos, también se incorporan como comentarios generales. Por el contrario, si involucran exclusivamente a un único atributo, se añaden como comentarios dentro del *setter* correspondiente. De este modo, se proporciona información contextual relevante al desarrollador, equilibrando la fidelidad al modelo original con la generación de código funcional y mantenible.

6

Pruebas y validación

En este capítulo se detalla la estrategia seguida para el diseño y la ejecución de las pruebas realizadas a los distintos componentes de la implementación de los elementos relacionados con esta rama del proyecto.

6.1. Estrategia de pruebas

Dado que el proyecto combina tanto la definición de un Meta-modelo como la implementación de transformaciones de Texto a Modelo y de Modelo a Texto, se ha considerado necesario realizar una validación en dos niveles complementarios. En términos generales, las pruebas se han diseñado para atender a tres aspectos fundamentales del sistema.

En primer lugar, se ha priorizado la validación del correcto funcionamiento de los componentes individuales del Meta-modelo, asegurando que cada entidad y su comportamiento básico cumplan con la funcionalidad prevista y con las expectativas diseñadas en el modelo. De esta manera se garantiza la fiabilidad de la base sobre la que se apoya el resto del sistema.

En segundo lugar, se ha puesto especial atención en la gestión de errores, asegurando no solo que las inconsistencias léxicas, sintácticas o semánticas sean detectadas, sino también que dichas situaciones se reporten de forma clara y manejable.

Finalmente, se ha considerado la fidelidad de las transformaciones, verificando tanto las transformaciones de Texto a Modelo como las de Modelo a Texto para asegurar que

no se produce perdida de información en ninguna etapa del proceso.

6.2. Pruebas unitarias

La pruebas unitarias han permitido verificar de manera temprana y a lo largo de todo el desarrollo que las diferentes clases del Meta-modelo y sus métodos de acceso y modificación presentan el comportamiento esperado, y que las estructuras internas se comportan como es previsto.

Para implementar estos test se han creado casos de prueba específicos para cada entidad del Meta-modelo, comprobando el correcto funcionamiento de los métodos que estas entidades exponen. En particular, se ha verificado que la adición o eliminación de elementos respete las restricciones estructurales del Meta-modelo, evitando conflictos. De manera complementaria, se han probado los métodos de consulta y modificación, asegurando que devuelvan los valores correctos y actualicen las estructuras internas de forma correcta.

En el Listing 15 se muestran unos ejemplos simples de los test implementados sobre la clase `MetaClass`. Estas pruebas reflejan el enfoque sistemático seguido en el resto de test unitarios para asegurar que cada componente del Meta-modelo se comporta de manera consistente y confiable.

En conjunto, estas pruebas han permitido detectar errores en fases tempranas del desarrollo, asegurando que estos se corrigen de forma localizada y eficiente, y han proporcionado un elevado nivel de confianza en la base sobre la que se han desarrollado las transformaciones de Texto a Modelo y de Modelo a Texto.

6.3. Pruebas de integración

Las pruebas de integración se han centrado en evaluar el comportamiento del sistema cuando los distintos componentes interactúan entre sí. En primer lugar, se han desarrollado tests que verifican que el sistema detecta correctamente los modelos que, si bien son correctos desde el punto de vista sintáctico, presentan inconsistencias semánticas. De esta manera, se asegura que el sistema es capaz de identificar situaciones como la referencia a elementos no definidos dentro del modelo —por ejemplo, la

```
1
2  void MetaClassTest::init() {
3      metaType = MetaModel::String::instance();
4
5      metaClass = new MetaModel::MetaClass("TestClass", false);
6  }
7
8  void MetaClassTest::metaClass_getName_returnsCorrectName(){
9  QCOMPARE(metaClass->getName(), "TestClass");
10 }
11
12 void MetaClassTest::metaClass_isSubClassOf_superClass_returnsTrue(){
13     auto superClass = std::make_shared<MetaModel::MetaClass>(
14     "TestSuperClass",
15     false);
16
17     metaClass->addSuperClass(superClass);
18
19     QCOMPARE(metaClass->isSubClassOf(*superClass), true);
20 }
```

Listing 15 Ejemplos representativos de pruebas unitarias sobre `MetaClass`

especificación de una superclase inexistente en una relación de generalización—, garantizando que tales errores sean reportados de forma clara y consistente. Este tipo de pruebas proporciona confianza en que la herramienta mantiene la integridad semántica del modelo y evita que errores conceptuales pasen inadvertidos durante las transformaciones de Texto a Modelo.

En segundo lugar, se ha comprobado la fidelidad de las transformaciones mediante pruebas de ida y vuelta: un modelo especificado en un archivo `.use` es sometido a una transformación de Texto a Modelo y, posteriormente, se realiza sobre la instancia del Meta-modelo generada la transformación de Modelo a Texto, de forma que se vuelve a generar un archivo `.use` a partir de dicho modelo. Las pruebas confirman que el resultado coincide con el archivo original, asegurando que la información y la estructura del modelo se mantienen intactas a lo largo del proceso.

7

Conclusiones y líneas futuras

Este capítulo recoge las conclusiones a las que se ha llegado tras todo el proceso de desarrollo de la herramienta, así como la discusión sobre las posibles líneas futuras de investigación y mejora.

7.1. Desarrollo de la herramienta

El proceso de desarrollo de esta herramienta ha resultado considerablemente complejo y ha supuesto un reto en varios ámbitos. En primer lugar, ha requerido una gran labor de investigación previa para comprender en profundidad los fundamentos teóricos relacionados con el proyecto, profundizando especialmente en las particularidades de las herramienta y lenguajes USE, UML y OCL, así como las gramáticas y su interpretación.

También, se ha requerido un análisis exhaustivo de la gramática de USE y OCL para obtener una compresión lo suficientemente profunda de ambos lenguajes de cara a su posterior re-implementación en ANTLR4. Esta compresión del subconjunto de UML que recoge USE y de OCL han sido también fundamentales para otro gran reto del desarrollo: la necesidad de diseñar un Meta-modelo coherente y que refleje las características del subconjunto de UML empleado, que ha implicado un proceso de modelado exhaustivo.

A estos desafíos se suma la integración de tecnologías como ANTLR4 para el análisis

sintáctico, lo cual exigió un esfuerzo adicional de adaptación, así como investigación de la validación semántica de USE para asegurar la validez de los modelos generados mediante las transformaciones de Texto a Modelo.

Asimismo, el desarrollo de las transformaciones de Modelo a Texto ha sido otro área a destacar puesto que además del conocimiento del lenguaje USE ya obtenido, la generación de código en Java requirió no solo profundizar en las convenciones propias del lenguaje, sino también en las buenas prácticas relacionadas con la definición de clases, constructores, métodos de acceso y gestión de colecciones.

Si bien el desarrollo de este proyecto ha representado un verdadero desafío debido a su magnitud y complejidad, también ha brindado una valiosa oportunidad de aprendizaje, permitiendo consolidar conocimientos en distintas áreas relevantes, como la ISDM, el modelado conceptual, la interpretación de lenguajes o los patrones de diseño. También se ha ampliado enormemente lo conocido sobre la herramienta USE, su funcionamiento y su proceso de validación, y se han adquirido amplios conocimientos del resto de herramientas empleadas. En definitiva, el desarrollo de esta herramienta ha sido una experiencia increíblemente enriquecedora y ha resultado en un sistema funcional y con una utilidad real en el ámbito de la Ingeniería del Software.

7.2. Líneas futuras

A pesar de lo expuesto en esta memoria y los resultados obtenidos, la herramienta presenta varios aspectos que pueden ser mejorados o ampliados en un futuro, aumentado su utilidad y su aplicabilidad práctica.

En primer lugar, una línea de trabajo especialmente relevante sería la mejora del soporte a OCL, integrando sus expresiones de manera más directa en el flujo de transformaciones de Modelo a Texto. Este avance permitiría no solo enriquecer la fidelidad de las transformaciones, sino también ofrecer un mayor grado de automatización en la generación de código y en la representación textual de los modelos.

Asimismo, como se ha comentado previamente, el diseño modular de la herramienta facilita la extensión hacia otros lenguajes y formatos de salida. En este sentido, la posibilidad de generar transformaciones hacia lenguajes como C++ o hacia formatos

estándar como XMI constituiría un paso decisivo para consolidar la versatilidad del sistema.

De manera complementaria, también resultaría interesante implementar transformaciones de Texto a Modelo a partir de otros formatos textuales, como XMI. Para ello solo habría que definir las gramáticas de estos formatos, si es que no están ya definidas, e implementar el proceso de visita y carga de datos en el Meta-modelo. Esto abriría la puerta a la interoperabilidad entre diferentes herramientas y favorecería el intercambio de modelos.

Finalmente, aunque en un horizonte más lejano y complejo, se plantea como posible línea de investigación la incorporación de mecanismos de evaluación de expresiones y de validación de modelos semejantes a los que ofrece USE. No obstante, este objetivo implicaría desarrollar un sistema completo de soporte para diagramas de objetos, lo que excede el alcance a corto plazo de la herramienta.

Bibliografía

[1] K. S. Martin Fowler, «UML Distilled: A Brief Guide to the Standard Object Modeling Language,» 1999.

[2] M. Richters y M. Gogolla, «OCL: Syntax, semantics, and tools,» en *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, Springer, 2002, págs. 42-68.

[3] S. Kent, «Model Driven Engineering,» en *Integrated Formal Methods*, M. Butler, L. Petre y K. Sere, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, págs. 286-298, ISBN: 978-3-540-47884-3.

[4] F. D. Muñoz, J. T. Castilla y A. V. Moreno, *Desarrollo de software dirigido por modelos*, 2019.

[5] *Object Management Group*. dirección: https://www.omg.org/mda/.

[6] D. W. Embley y B. Thalheim, «Handbook of conceptual modeling,» *Berlin, Heidel*, 2011.

[7] D. Berardi, D. Calvanese y G. De Giacomo, «Reasoning on UML class diagrams,» *Artificial Intelligence*, vol. 168, n.º 1, págs. 70-118, 2005, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2005.05.003. dirección: https://www.sciencedirect.com/science/article/pii/S0004370205000792.

[8] *2 Ejemplos de Diagramas de Clases UML*. dirección: https://www.webyempresas.com/ejemplos-de-diagramas-de-clases-uml/.

[9] *Visual Paradigm*. dirección: https://www.visual-paradigm.com/.

[10] *StarUML*. dirección: https://staruml.io/.

[11] *PlantUML*. dirección: https://plantuml.com/es/.

[12] *Enterprise Architect*. dirección: https://sparxsystems.com/products/ea/.

[13] E. M. Clarke, «Model checking,» en *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh y G. Sivakumar, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, págs. 54-56, ISBN: 978-3-540-69659-9.

[14] *Alloy Analyzer*. dirección: https://alloytools.org/.

[15] M. Richters y M. Gogolla, «Validating UML Models and OCL Constraints,» en *UML 2000 — The Unified Modeling Language*, A. Evans, S. Kent y B. Selic, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, págs. 265-277, ISBN: 978-3-540-40011-0.

[16] *USE Manual*, Universidad de Bremen, 2007. dirección: https://github.com/useocl/use/blob/master/manual/main.md.

[17] P. Hudak, «Domain-specific languages,» *Handbook of programming languages*, vol. 3, n.º 39-60, pág. 21, 1997.

[18] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[19] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal y A.-J. Berre, «Toward Standardised Model to Text Transformations,» en *Model Driven Architecture – Foundations and Applications*, A. Hartman y D. Kreische, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, págs. 239-253, ISBN: 978-3-540-32093-7.

[20] *MOFM2T — MOF Model to Text Transformation Language*, OMG. dirección: https://www.omg.org/spec/MOFM2T.

[21] L. M. Rose, N. Matragkas, D. S. Kolovos y R. F. Paige, «A feature model for model-to-text transformation languages,» en *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, 2012, págs. 57-63. DOI: 10.1109/MISE.2012.6226015.

[22] O. V. Chebanyuk, «An Approach of Text to Model Transformation of Software Models.,» en *ENASE*, 2018, págs. 432-439.

[23] *XMI — XML Metadata Interchange*, OMG. dirección: https://www.omg.org/spec/XMI/.

[24] A. Ferrari, S. Abualhaija y C. Arora, «Model generation from requirements with LLMs: an exploratory study,» *CoRR*, 2024.

[25] A. R. Sadik, S. Brulin y M. Olhofer, «Coding by design: Gpt-4 empowers agile model driven development,» *arXiv preprint arXiv:2310.04304*, 2023.

[26] B. Stroustrup, «An overview of C++,» en *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, 1986, págs. 7-18.

[27] *Introduction to Qt*, Qt Group. dirección: https://doc.qt.io/qt-6/qt-intro.html.

[28] G. Ramos Jiménez y J. d. Campo Ávila, *Cuestiones de teoría de autómatas y lenguajes formales I*, spa. Málaga: [s.l.], 2007, ISBN: 9788461200177.

[29] T. Parr, *The Definitive ANTLR 4 reference*. ene. de 2013. dirección: https://learning.oreilly.com/library/view/the-definitive-antlr/9781941222621.

[30] *Java*. dirección: https://www.java.com/es/download/help/whatis_java.html.

[31] *Git*. dirección: https://git-scm.com/.

[32] *Trello*, Atlassian. dirección: https://trello.com/.

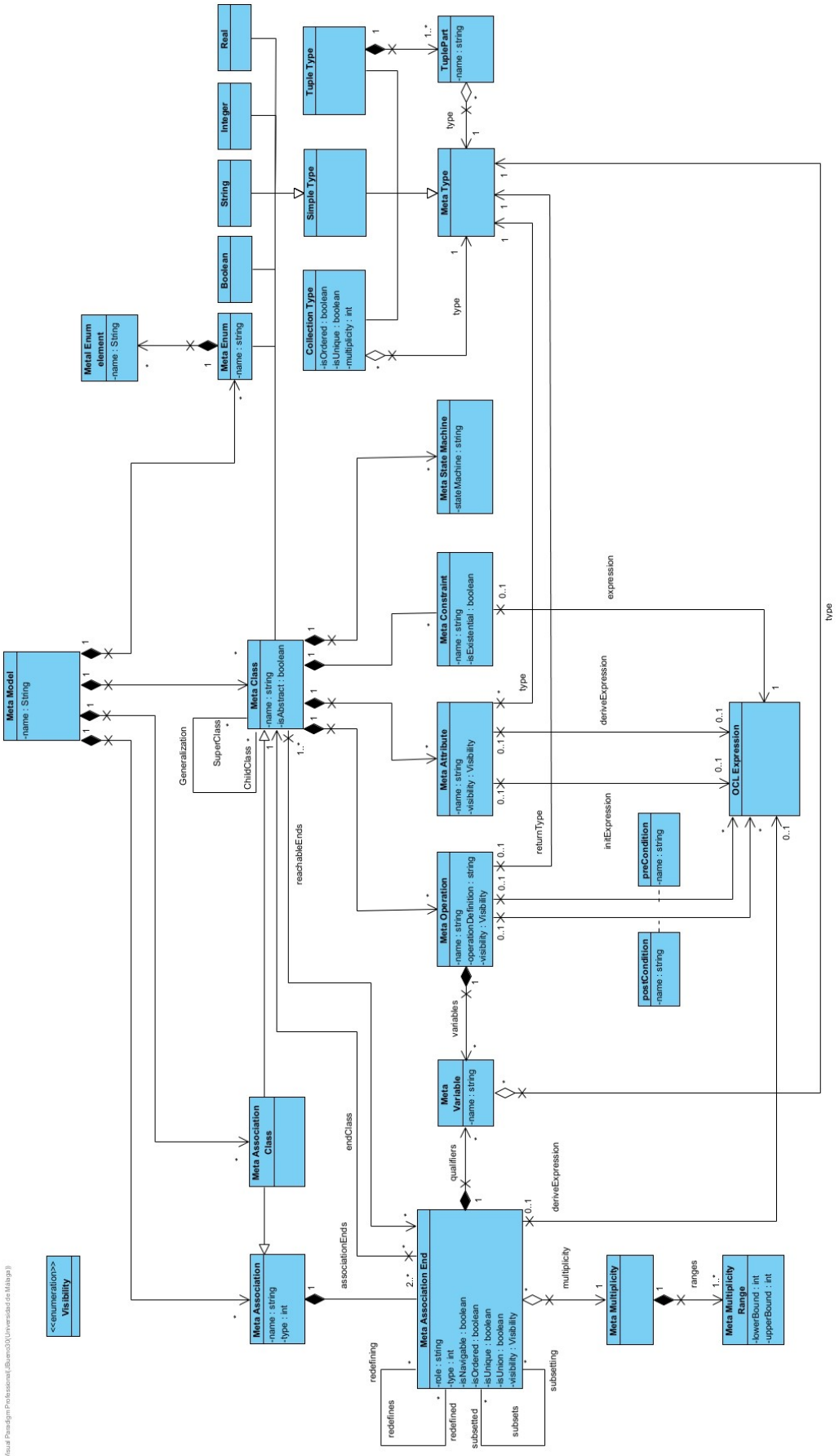
[33] *Model/View Programming*, Qt Group. dirección: https://doc.qt.io/qt-6/model-view-programming.html.

[34] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008, ISBN: 978-0132350884.

[35] V. Sarcar, «Command Pattern,» en *Java Design Patterns: A Hands-On Experience with Real-World Examples*. Berkeley, CA: Apress, 2022, págs. 405-433, ISBN: 978-1-4842-7971-7. DOI: 10.1007/978-1-4842-7971-7_19. dirección: https://doi.org/10.1007/978-1-4842-7971-7_19.

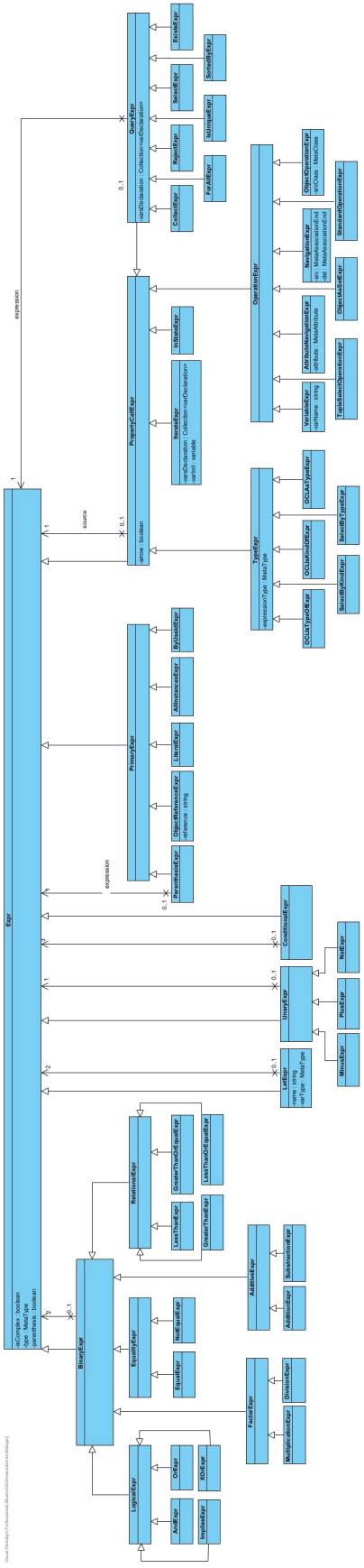
A

Apéndice A. Diagrama de clases del Meta-modelo


Real
Integer
String
Boolean
Tuple Type
TuplePart
-name : string
Simple Type
Meta Type
Collection Type
-isOrdered : boolean
-isUnique : boolean
-multiplicity : int
Metal Enum element
-name : String
Meta Enum
-name : string
Meta State Machine
-stateMachine : string
Meta Constraint
-name : string
-isExistential : boolean
Meta Model
-name : String
Meta Class
-name : string
-isAbstract : boolean
Generalization
SuperClass
ChildClass
Meta Attribute
-name : string
-visibility : Visibility
OCL Expression
preCondition
-name : string
Meta Operation
-name : string
-operationDefinition : string
-visibility : Visibility
postCondition
-name : string
Meta Variable
-name : string
Meta Association Class
Meta Association
-name : string
-type : int
Meta Association End
-role : string
-type : int
-isNavigable : boolean
-isOrdered : boolean
-isUnique : boolean
-isUnion : boolean
-visibility : Visibility
Meta Multiplicity
Meta Multiplicity Range
-lowerBound : int
-upperBound : int
<<enumeration>>
Visibility
type
expression
deriveExpression
returnType
initExpression
reachableEnds
variables
qualifiers
endClass
associationEnds
multiplicity
ranges
redefining
redefines
redefined
subsetted
subsets
subsetting

B

Apéndice B. Diagrama de clases de OCL


Expr
isComplex : boolean
type : MetaType
canBeBool : boolean
expression
source
PrimaryExpr
ObjectReferenceExpr
referred : string
LitExpr
name : string
varType : MetaType
PropertyCallExpr
isArray : boolean
VariableExpr
varName : string
QueryExpr
varDeclaration : Collection<varDeclaration>
OperationExpr
TypeExpr
expressionType : MetaType
BinaryExpr
RelationalExpr
EqualityExpr
AdditiveExpr
FactorExpr
LogicalExpr
UnaryExpr
ConditionalExpr

C

Apéndice C. Guía de instalación

C.1. Instalación del proyecto

En el caso de querer compilar el proyecto desde su código fuente, se deben seguir los siguientes pasos:

- Instalar Git.
- Clonar el repositorio del proyecto desde GitHub:

```
git clone https://github.com/JoseBueno30/ModelForge.git
```

- Instalar el kit de desarrollo de Qt, versión 6.8.1.
- (OPCIONAL) Instalar Qt Creator, ya que ofrece más facilidades a la hora de utilizar recursos de Qt.
- Abrir el proyecto en el IDE.
- Configurar el proyecto para que use el kit de desarrollo instalado.
- Compilar y ejecutar el proyecto con CMake.

C.2. Instalación de la aplicación

En el caso de querer instalar la aplicación directamente:

1. Se debe descargar el zip desde la página de lanzamientos del repositorio del proyecto en GitHub: `https://github.com/JoseBueno30/ModelForge/releases`
2. Descomprimir el archivo zip en la ubicación deseada.
3. Ejecutar la aplicación `ModelForge.exe`.

D

Apéndice D. Manual de Usuario

Este apéndice describe el uso de la aplicación, sus principales características y las instrucciones necesarias para que un usuario pueda trabajar con ella de forma eficaz.

D.1. Inicio de la aplicación

Al ejecutar la aplicación, el usuario accede a la ventana principal (*MainWindow*), que contiene la barra de menús, las herramientas de modelado y el área de trabajo central en el que se mostrará el modelo gráficamente.

Figura D.1 Ventana principal de la aplicación.

D.2. Funciones principales

D.2.1. Gestión de archivos

En cuanto a la gestión de archivos, el usuario puede crear, abrir y guardar modelos UML mediante las opciones disponibles en la barra de menús o los atajos de teclado y, cargar y guardar la disposición de elementos. Las opciones son las siguientes:

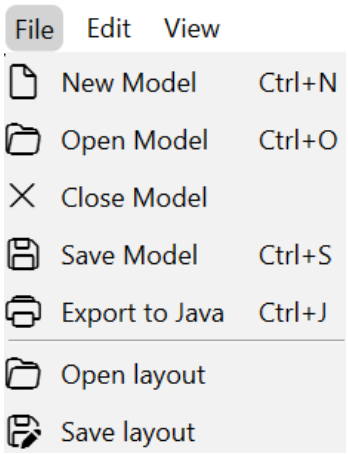


Figura D.2 Menú de gestión de archivos.

- **New Model**: crea un modelo vacío, habilitando todas las acciones relacionadas con él, al que se le puede asignar un nombre posteriormente (Figura D.3).

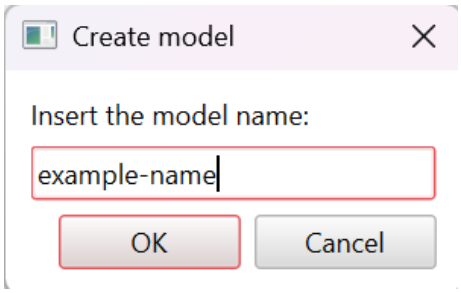


Figura D.3 Creación de un nuevo modelo.

- **Open Model**: permite cargar archivos en formato `.use`.
- **Save Model**: almacena el modelo actual en formato `.use`.
- **Close Model**: cierra el modelo actual, avisando al usuario.
- **Export To Java**: genera el código en `Java` correspondiente al modelo actual.

- **Open Layout**: carga la disposición de los elementos desde un archivo `.clt` (formato utilizado por USE) o `.json`.
- **Save Layout**: guarda la disposición de los elementos en un archivo `.json`.

D.2.2. Edición del modelo

A continuación, se describen las acciones disponibles para editar el modelo UML. Los botones para añadir nuevos elementos al modelo se encuentran en la caja de herramientas inferior izquierda (Figura D.4). Todos los elementos requieren que su nombre sea único dentro del modelo para ser válidos.

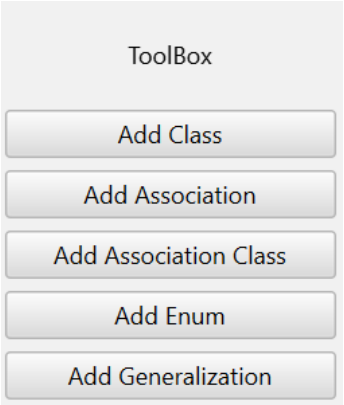


Figura D.4 Caja de herramientas de edición del modelo.

- **Crear y modificar clases UML**: para realizar esta acción encontramos el botón *Add Class*. Al hacer clic en este, se abrirá la ventana de propiedades de la nueva clase (Figura D.5). En dicha ventana podremos editar el nombre de la clase, marcar si es abstracta o no, y añadir atributos, operaciones y restricciones OCL. Una vez creada la clase, se añade al área central de trabajo.

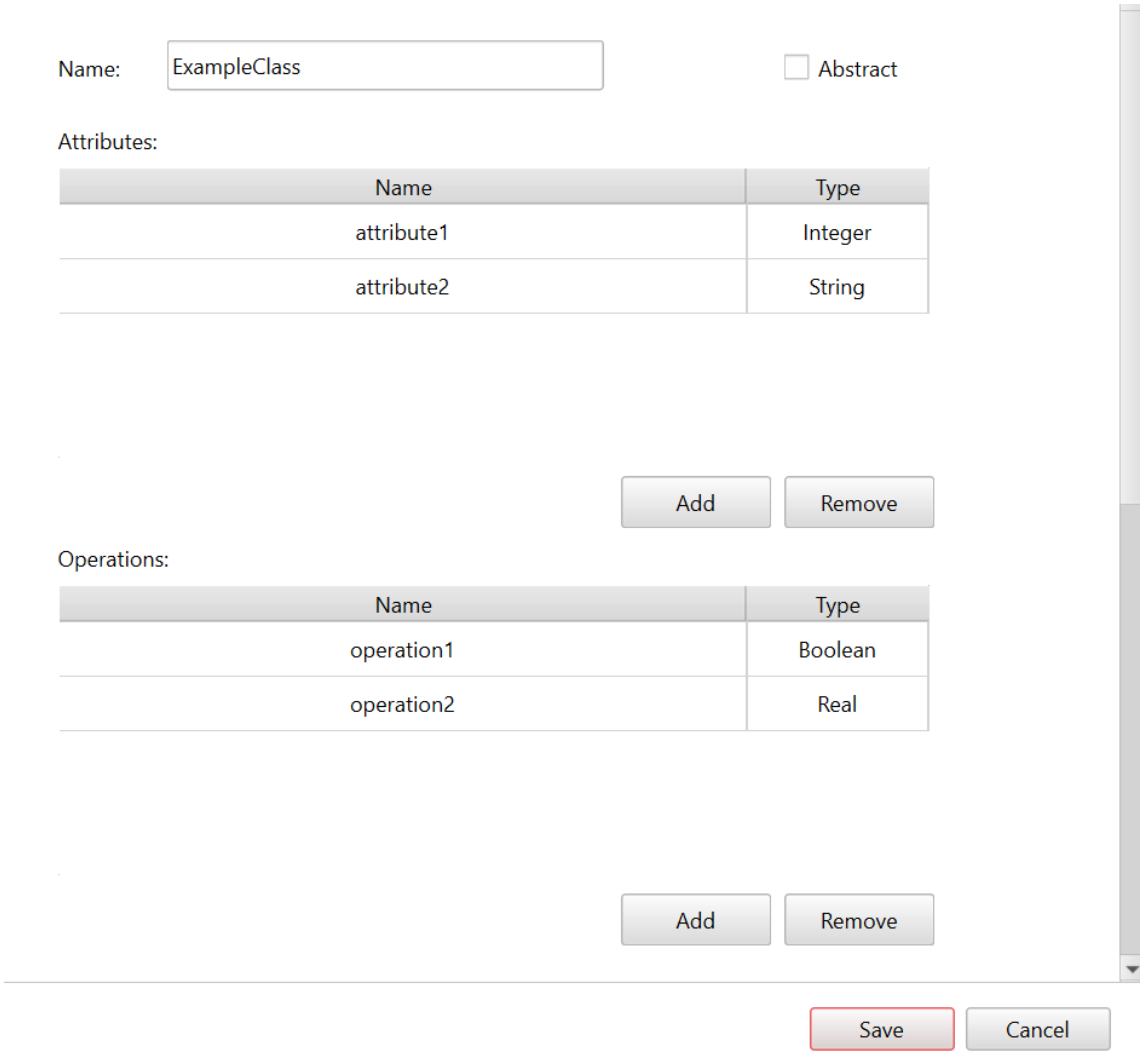


Figura D.5 Creación de una nueva clase UML.

- **Crear y modificar atributos**: Una vez creada la clase, dentro de su ventana de edición encontraremos la tabla de atributos. Al darle click a su respectivo botón *Add*, o al hacer docle click sobre un atributo existente, se abrirá la ventana de edición de atributos (Figura D.6). En esta podremos definir el nombre del atributo, su tipo (de los tipos básicos disponibles), su visibilidad, si tiene expresión derivada y si tiene expresión de inicialización.

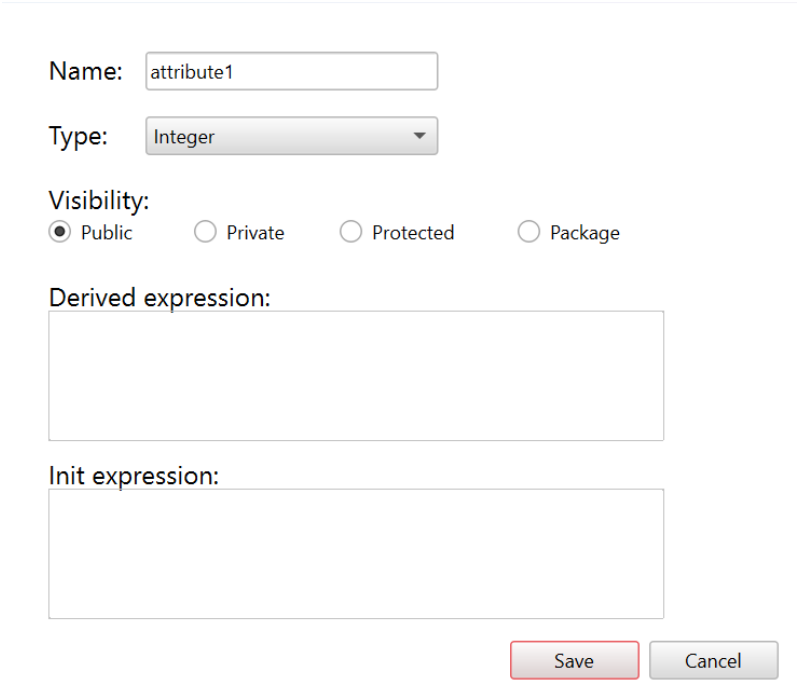


Figura D.6 Creación de un nuevo atributo.

- **Crear y modificar operaciones**: De forma similar a los atributos, en la tabla de operaciones de la ventana de edición de la clase, al darle click a su respectivo botón *Add* se abrirá la ventana de edición de operaciones (Figura D.7). En esta podremos definir el nombre de la operación, su tipo de retorno (de los tipos básicos disponibles), su visibilidad, sus variables de entrada (que se podrán añadir de igual forma con el botón *Add*, o editarlos haciendo doble click sobre ellos) y, sus precondiciones y postcondiciones (de nuevo, haciendo uso de su respectivo botón *Add*, o pulsando sobre alguna existente, en la sección correspondiente y rellenar un campo de texto con la expresión y establecer su tipo).

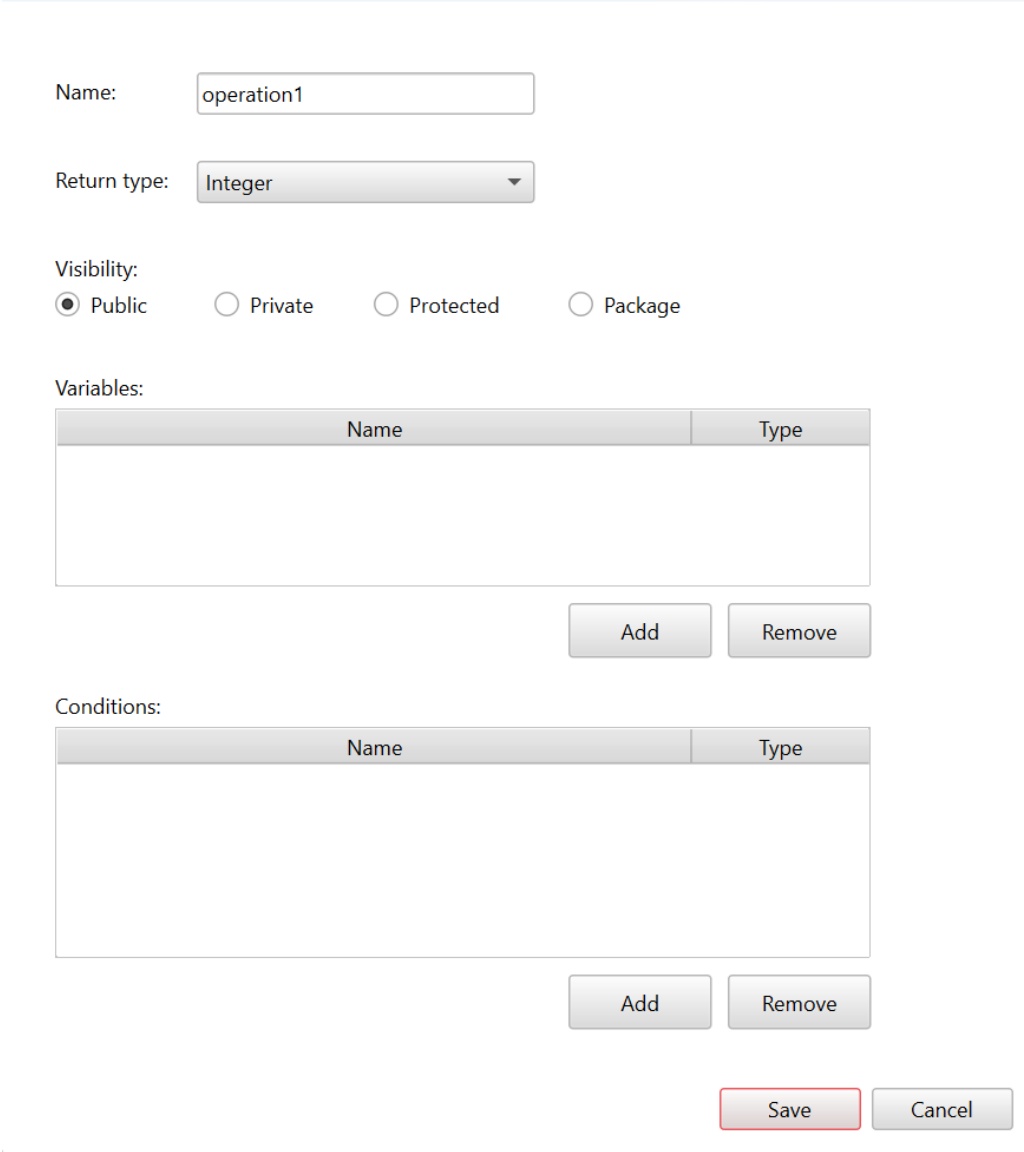


Figura D.7 Creación de una nueva operación.

- **Crear y modificar enumerados UML**: para realizar esta acción encontramos en la caja de herramientas inferior el botón *Add Enum*. Al hacer clic en este, se abrirá la ventana de propiedades del nuevo enumerado (Figura D.8). En dicha ventana podremos editar el nombre del enumerado y añadir sus literales. Una vez creado el enumerado, se añade al área central de trabajo. También se pueden editar los enumerados haciendo doble click sobre ellos en el área de trabajo.

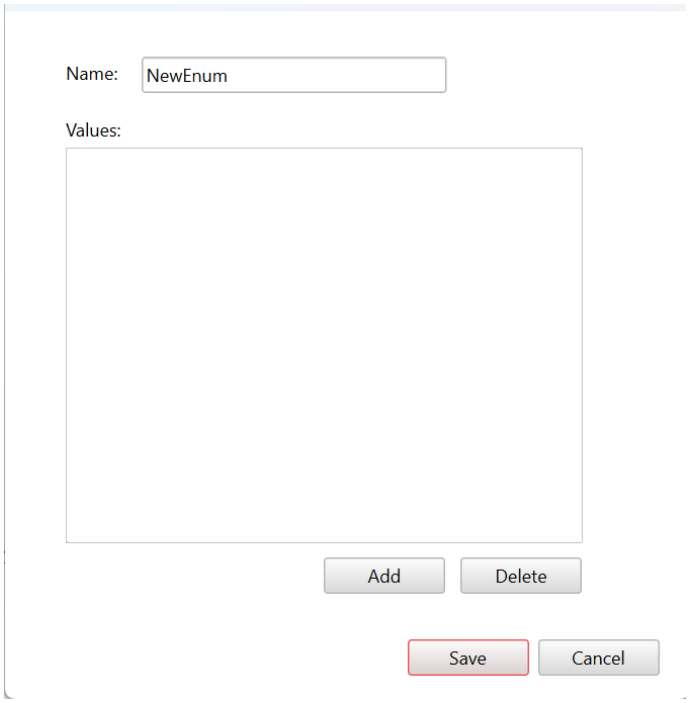


Figura D.8 Creación de un nuevo enumerado UML.

- **Crear y modificar asociaciones UML**: para crear relaciones se deberá hacer click en el botón *Add Association*, abriendo la ventana de edición de una asociación. A continuación, se deberá establecer el nombre, el tipo de asociación (Asociación normal, Generalización o Agregación), y rellenar correctamente los datos de los extremos de la asociación (rol único, multiplicidad válida, visibilidad y su clase). Para editar una relación existente, se deberá hacer doble click sobre ella en el área de trabajo, o si es una asociación reflexiva (de una clase a ella misma) se podrá editar eligiéndola en la lista de asociaciones reflexivas que presentan las clases en su ventana de edición.

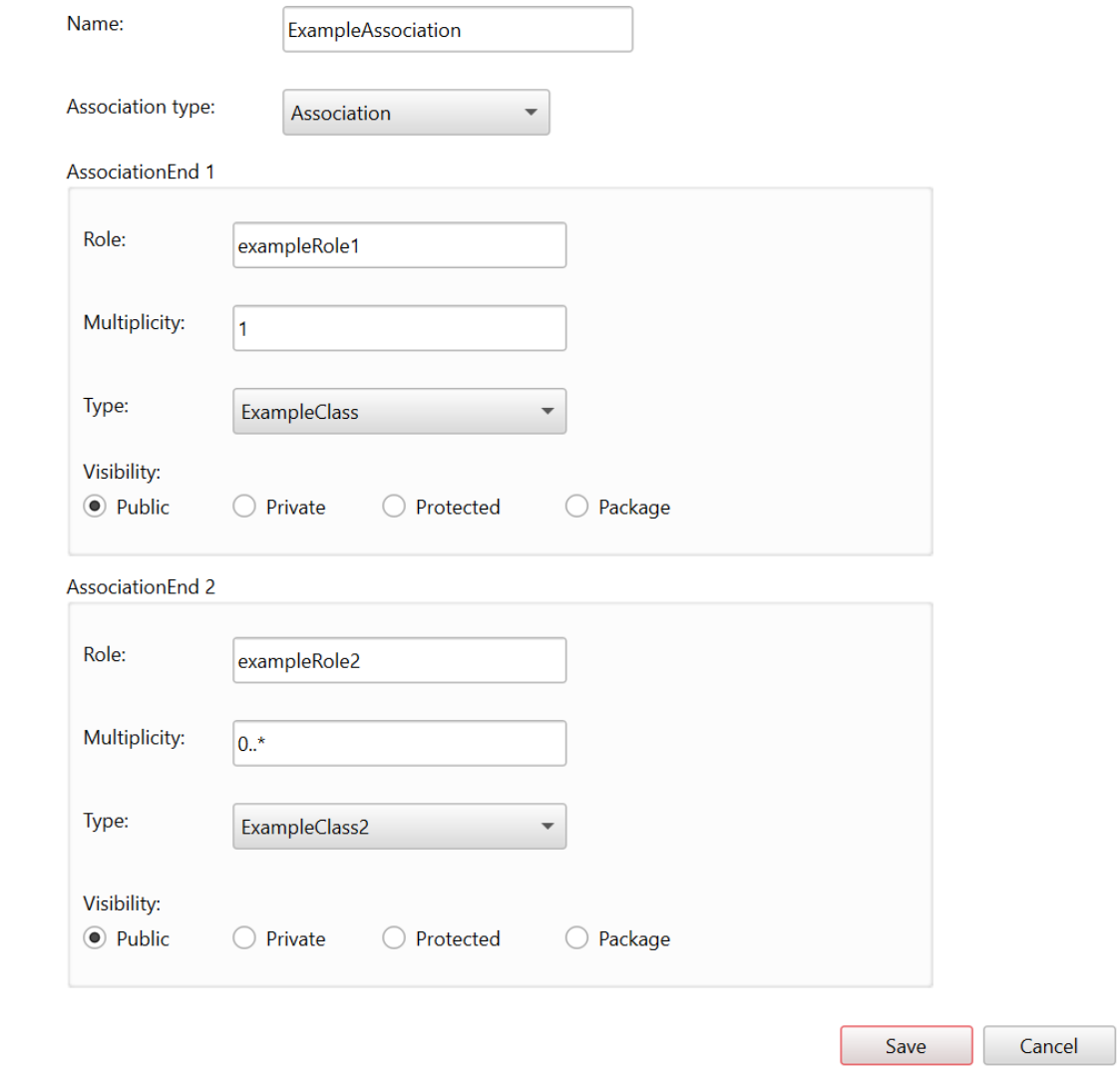


Figura D.9 Creación de una nueva asociación UML.

- **Crear y modificar clases asociación UML**: para crear este tipo de relaciones se deberá hacer click en el botón *Add Association Class* y se abrirán en orden, la ventana de edición de una clase y después la de una asociación. En cada una de estas ventanas se deberá rellenar la información necesaria como se ha explicado anteriormente. Para editarla posteriormente, se deberá hacer doble click sobre alguna parte de la clase asociación y se podrá editar la información de dicha parte. Es decir, si se hace click sobre la parte de la clase, se abrirá la ventana de edición de la clase, y si se hace click sobre la línea de la asociación, se abrirá la ventana de edición de la asociación.

- **Eliminar elementos seleccionados**: para eliminar cualquier elemento gráfico del modelo, simplemente se ha de seleccionar, haciendo click una vez sobre el elemento, y pulsar la tecla `Supr`. Para saber qué elemento está seleccionado, este se mostrará con un borde azul (Figura D.10).

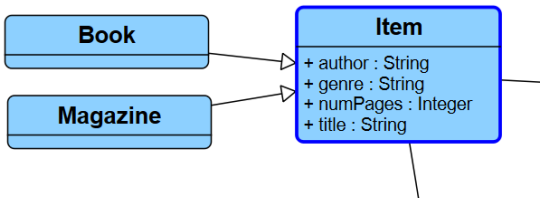


Figura D.10 Elemento seleccionado con el borde azulado.

D.2.3. Controles del área de trabajo

- Zoom in/out del área de trabajo con la rueda del ratón.
- Desplazamiento del lienzo con arrastre. Se puede realizar manteniendo pulsado el botón izquierdo del ratón o el botón central (rueda).
- Desplazamiento de los elementos gráficos con arrastre. Dichos elementos son las clases y los enumerados. Simplemente se ha de hacer click sobre el elemento y arrastrarlo a la posición deseada.

D.3. Gestión de temas

El usuario puede alternar entre *tema claro* y *tema oscuro* desde el menú de Vista, *View*, en la barra superior de herramientas. Ahí encontrará el botón *Toogle view*. También puede utilizar el atajo de teclado `Ctrl+T`. Esto mejora la experiencia de uso y la accesibilidad en diferentes entornos de trabajo.

D.4. Resolución de problemas comunes

- **El archivo no se abre**: comprobar que tiene extensión `.use` válida.
- **No aparece el diagrama**: verificar que el archivo contiene clases o relaciones definidas.

- **Errores de OCL**: revisar la sintaxis de las restricciones.
- **No se puede guardar el archivo**: asegurarse de tener permisos de escritura en la ubicación seleccionada.

D.5. Acciones reversibles

La aplicación soporta acciones de deshacer y rehacer, permitiendo al usuario revertir cambios recientes. Estas acciones se encuentran en la barra superior de herramientas (en el menú *Edit*), representadas por los iconos de flechas hacia la izquierda (deshacer) y hacia la derecha (rehacer). También se pueden utilizar los atajos de teclado `Ctrl+Z` para deshacer y `Ctrl+Y` para rehacer. Dichas acciones son crear, editar y eliminar una clase, una asociación, una clase asociación y un enumerado, y mover una clase o un enumerado.

D.6. Portapapeles gráfico

La aplicación incluye un portapapeles gráfico que permite copiar, cortar y pegar elementos dentro del área de trabajo. Estas acciones se encuentran en la barra superior de herramientas (en el menú *Edit*), representadas por los iconos de dos hojas (copiar), una hoja con una tijera (cortar) y una hoja con un portapapeles (pegar). También se pueden utilizar los atajos de teclado `Ctrl+C` para copiar, `Ctrl+X` para cortar y `Ctrl+V` para pegar. Al pegar un elemento, este se colocará con un nombre auxiliar para evitar conflictos de nombres. El elemento cortado o copiado perderá todas sus asociaciones. Solo se pueden copiar o cortar las clases y los enumerados.

D.7. Recopilación de atajos de teclado

- `Ctrl+N`: Nuevo modelo.
- `Ctrl+O`: Abrir modelo.
- `Ctrl+S`: Guardar modelo.
- `Supr`: Eliminar elemento seleccionado.

- `Ctrl+Z / Ctrl+Y`: Deshacer / Rehacer.
- `Ctrl+T`: Alternar tema claro/oscuro.
- `Ctrl+C / Ctrl+X / Ctrl+V`: Copiar / Cortar / Pegar, respectivamente.
- `Ctrl+J` : Exportar a Java.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga