Graduado en Ingeniería del Software

ModelForge: Herramienta de modelado conceptual y generación de código

Interfaz de Usuario y funcionalidad de alto nivel

ModelForge: A tool for conceptual modeling and code generation

User Interface and high-level functionality

Realizado por
Víctor Pérez Armenta

Tutorizado por
Javier Troya Castilla

Departamento
Lenguaje y ciencias de la computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2025

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

ModelForge: Herramienta de modelado conceptual y generación de código

Interfaz de Usuario y funcionalidad de alto nivel

ModelForge: A tool for conceptual modeling and code generation

User Interface and high-level functionality

Realizado por
Víctor Pérez Armenta

Tutorizado por
Javier Troya Castilla

Departamento
Lenguaje y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2025

Fecha defensa: septiembre de 2025

Resumen

El modelado constituye una práctica esencial en la ingeniería de software, ya que permite representar de forma abstracta y precisa la estructura y el comportamiento de los sistemas antes de su implementación, facilitando la comunicación entre los distintos actores y reduciendo errores en el proceso de desarrollo. Además, resulta clave para garantizar la usabilidad de las aplicaciones, especialmente en entornos educativos y profesionales donde la facilidad de uso impacta directamente en la productividad y el aprendizaje.

Sin embargo, el principal problema radica en la complejidad y fragmentación de las herramientas existentes, que dificultan su adopción por parte de usuarios no expertos y limitan la integración de funcionalidades como el modelado visual, la validación formal y la generación automática de código. Muchas de estas aplicaciones presentan interfaces poco intuitivas, carecen de soporte para restricciones o requieren recurrir a múltiples herramientas para completar el flujo de trabajo, lo que afecta negativamente a la eficiencia y coherencia del proceso de diseño.

El objetivo de este trabajo es desarrollar una herramienta moderna, de código abierto, centrada en la usabilidad y la eficiencia, que permita a los usuarios crear y editar diagramas de clases de forma gráfica, validar modelos y restricciones sintácticamente, y transformar dichos modelos en código fuente. La aplicación busca ofrecer una solución orientada tanto a entornos académicos como profesionales, facilitando el aprendizaje, la experimentación y la productividad en el modelado de software.

Palabras clave: Modelado, Diagrama de Clase, UML, Interfaz de Usuario, Usabilidad.

Abstract

Modeling is an essential practice in software engineering, enabling the abstract and precise representation of system structure and behavior prior to the implementation, improving communication among stakeholders and reducing errors during development. Additionally, usability is crucial, especially in educational and professional environments where ease of use directly impacts productivity and learning.

However, the main problem is the complexity and fragmentation of existing tools, which hinder non-expert users and limit the integration of features such as visual modeling, formal validation, and automatic code generation. Many current applications have unintuitive interfaces, lack support for constraints, or require multiple tools to complete the workflow, negatively affecting the efficiency and consistency of the design process.

The goal of this work is to develop a modern, open-source tool focused on usability and efficiency, enabling users to create and edit class diagrams graphically, validate models using constraints, and transform these models into source code. The application aims to provide a solution for both academic and professional environments, facilitating learning, experimentation, and productivity in software modeling.

Keywords: Modeling, Class Diagram, UML, User Interface, Usability.

Índice

Índice de figuras **9**

1. Introducción **11**

1.1. Motivación 11

1.2. Planteamiento del problema 13

1.3. Objetivo 14

1.4. Organización de la memoria 14

2. Marco teórico **17**

2.1. UML: Lenguaje de Modelado Unificado 17

2.2. OCL: Object Constraint Language 19

2.3. Lenguajes Formales y Gramáticas 20

2.4. Interacción Persona-Ordenador 21

3. Estado del Arte **23**

3.1. Antecedentes 23

3.2. Herramientas CASE actuales 23

4. Tecnologías **29**

4.1. C++ 29

4.2. Qt Framework 30

4.3. Qt Creator 30

4.4. Figma 31

4.5. ANTLR4 31

4.6. Git 32

4.7. Trello . . . 32

5. Metodologías de trabajo **35**

5.1. Adaptación de Scrum al equipo . . . 35

5.2. Sprints . . . 36

5.3. Product Backlog . . . 36

5.4. Sprint Backlog . . . 36

5.5. Reuniones y seguimiento . . . 37

5.6. Fases del desarrollo . . . 37

5.7. Ventajas del enfoque adoptado . . . 38

6. Requisitos del sistema **39**

6.1. Requisitos funcionales . . . 39

6.2. Requisitos no funcionales . . . 41

7. Modelado y diseño **43**

7.1. Diseño de la Interfaz de Usuario . . . 43

7.1.1. Consistencia visual y funcional . . . 43

7.1.2. Personalización y accesibilidad . . . 44

7.1.3. Visibilidad del estado del sistema . . . 45

7.1.4. Organización y jerarquía de la información . . . 45

7.1.5. Simplicidad y reducción de carga cognitiva . . . 45

7.1.6. Prevención y recuperación de errores . . . 46

7.2. Diseño Conceptual del Modelo de Datos . . . 46

7.3. Modelado de la Interfaz de Usuario . . . 51

8. Implementación y pruebas **53**

8.1. Implementación de las ventanas . . . 53

8.1.1. La ventana principal: MainWindow . . . 53

8.1.2. Conexión de acciones con la barra superior . . . 55

8.1.3. Ventanas de edición de elementos UML . . . 56

8.1.4. Ventanas de alerta y confirmación . . . 59

8.2. Implementación de `ModelGraphicsView` . . . 60

8.2.1. Eventos personalizados en `ModelGraphicsView` . . . 61

8.2.2. Extensión de `QGraphicsScene`: `ModelGraphicsScene` 65
8.3. Implementación de los `QGraphicsItem` personalizados 66
8.4. Implementación del patrón `Command` 72
8.5. Implementación de un portapapeles gráfico 74
8.6. Gestión de Temas y Estilos Visuales 76
8.7. Guardado y carga de la disposición de la escena 77
8.8. Pruebas Unitarias del Metamodelo 77
8.8.1. Objetivos de las pruebas 78
8.8.2. Metodología aplicada 78
8.8.3. Resultados obtenidos 79

9. Conclusión **81**
9.1. Experiencia 81
9.2. Líneas futuras 82

A. Guía de instalación **89**
A.1. Instalación del proyecto 89
A.2. Instalación de la aplicación 90

B. Manual de Usuario **91**
B.1. Inicio de la aplicación 91
B.2. Funciones principales 92
B.2.1. Gestión de archivos 92
B.2.2. Edición del modelo 93
B.2.3. Controles del área de trabajo 97
B.3. Gestión de temas 97
B.4. Resolución de problemas comunes 97
B.5. Acciones reversibles 98
B.6. Portapapeles gráfico 98
B.7. Recopilación de atajos de teclado 98

Índice de figuras

1.1. Fases del proceso de desarrollo de software, Pressman 2010. [2] 12

2.1. Lista de elementos UML, extraída de [6] 18

3.1. Captura de la herramienta Enterprise Architect, extraída de [13] 24
3.2. Captura de la herramienta Visual Paradigm, extraída de [15] 25
3.3. Captura de la herramienta StarUML, extraída de [17] 25
3.4. Captura de la herramienta USE 26

5.1. Organización de los sprints en Trello. 37

7.1. Dark mode prototype 44
7.2. Light mode prototype 44
7.3. Light mode prototype 45
7.4. UML/USE-Diagram 50
7.5. Diagrama de clases para la interfaz de usuario basado en Qt 52

8.1. Estructura básica de la ventana principal (`MainWindow`). 54
8.2. Acciones principales en la `Barra superior de herramientas`. 56
8.3. Ventana de edición de una clase. 57
8.4. Ventana de edición de una asociación. 58
8.5. Ventana de error. 59
8.6. Ventana de confirmación. 59
8.7. Representación visual de una clase UML. 68
8.8. Representación visual de una clase abstracta UML. 69
8.9. Representación visual de un enumerado UML. 69

8.10. Representación visual de una clase de asociación UML. 71
8.11. Representación visual de una generalización UML. 72
8.12. Acciones de la pila en la barra superior. 73
8.13. Acciones de portapapeles en la barra superior. 75

B.1. Ventana principal de la aplicación. 91
B.2. Menú de gestión de archivos. 92
B.3. Creación de un nuevo modelo. 92
B.4. Caja de herramientas de edición del modelo. 93
B.5. Creación de una nueva clase UML. 93
B.6. Creación de un nuevo atributo. 94
B.7. Creación de una nueva operación. 95
B.8. Creación de un nuevo enumerado UML. 95
B.9. Creación de una nueva asociación UML. 96
B.10. Elemento seleccionado con el borde azulado. 97

Capítulo 1

Introducción

En este capítulo se presenta el contexto general del trabajo, así como la motivación que impulsa su desarrollo y los objetivos que se persiguen. Se parte de una reflexión sobre la importancia del modelado conceptual y la interacción persona-ordenador en el ámbito de la ingeniería de software, destacando su relevancia en la mejora de la calidad de los sistemas y en la reducción de errores durante el proceso de desarrollo. A continuación, se plantean los objetivos específicos que guiarán la realización del proyecto, los cuales buscan sentar las bases para el diseño e implementación de una herramienta CASE (*Computer-Aided Software Engineering*) moderna, usable y extensible.

1.1. Motivación

En un proyecto de desarrollo software solemos encontrar las siguientes etapas (Figura 1.1): análisis de requisitos, modelado y diseño, implementación, pruebas y mantenimiento. Dentro de estas fases, el modelado de software se ha consolidado como un pilar fundamental. Permite representar de manera abstracta y precisa los sistemas con anterioridad a su implementación [1]. Teniendo una buena representación del sistema, podemos evitar numerosos problemas en la fase de implementación, como por ejemplo, la repetición de código.

Dicha disciplina resulta esencial ya que: permite construir sistemas eficientes, escalables, robustos y mantenibles, gracias a la abstracción y estructuración que proporciona. También

ayuda a reducir el nivel de complejidad del sistema, debido a la capacidad de descomponerlo en componentes más manejables. Y además, contribuye a mejorar la comunicación entre los distintos actores involucrados y garantizar la coherencia entre los diferentes niveles de diseño, facilitando así el trabajo colaborativo y la integración de diferentes perspectivas.

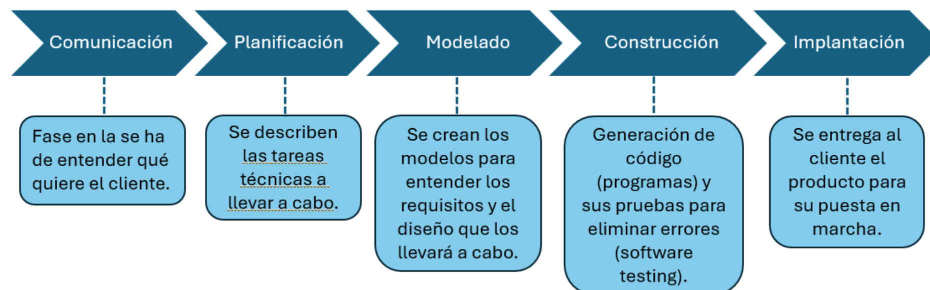

Figura 1.1 Fases del proceso de desarrollo de software, Pressman 2010. [2]

Existen diversas maneras de afrontar la fase modelado: una perspectiva externa (mostrando el contexto y el entorno donde funcionará nuestro sistema), una perspectiva de interacción (mostrando la interacción del sistema con el entorno), una perspectiva de comportamiento (modelando el comportamiento frente a eventos) y una perspectiva estructural (definiendo la organización y estructura del sistema). Para estas diferentes perspectivas se hace uso de lenguajes estandarizados como UML (*Unified Modeling Language*), que nos permite representar los elementos de un sistema de manera formal, y OCL (*Object Constraint Language*), que nos proporciona mecanismos para establecer restricciones sobre nuestro sistema. Esto facilita la especificación formal de los modelos, proporcionando una base común que favorece la interoperabilidad entre herramientas y el intercambio de conocimiento en el ámbito académico y profesional. Estas representaciones no solo permiten documentar y validar sistemas, sino que también sirven como punto de partida para la generación automática de código, aumentando la productividad y reduciendo el riesgo de errores.

Por otro lado, la interacción persona-ordenador y la usabilidad adquieren un papel central en el diseño de aplicaciones [3]. Una interfaz gráfica clara, intuitiva y bien estructurada favorece que los usuarios puedan concentrarse en las tareas sin que la herramienta suponga una barrera adicional. Este aspecto resulta especialmente relevante en entornos educativos, donde la facilidad de uso de las aplicaciones impacta directamente en la asimilación de conceptos teóricos por parte del alumnado.

En el ámbito académico, disponer de herramientas de modelado usables, accesibles y potentes contribuye a reforzar la enseñanza de conceptos clave de la informática, como el uso de patrones de diseño, la especificación de restricciones o la validación de modelos. Además, permite establecer un vínculo más estrecho entre la teoría y la práctica, favoreciendo que los estudiantes y profesionales puedan experimentar de manera directa con las metodologías y estándares utilizados en la industria. En definitiva, son necesarios entornos de modelado software que combinen rigor formal, usabilidad y accesibilidad, ofreciendo a la vez un apoyo sólido para la ingeniería del software brindando un conjunto de funcionalidades que permitan facilitar el trabajo.

1.2. Planteamiento del problema

Aun siendo el modelado de software una actividad esencial en el desarrollo de sistemas complejos, persisten diversas dificultades que limitan su aprovechamiento en contextos tanto académicos como profesionales. Muchas de las herramientas actuales presentan un nivel elevado de complejidad, lo que dificulta su adopción en entornos educativos, donde los estudiantes deben centrarse en asimilar los fundamentos teóricos y prácticos del modelado. La carencia de interfaces simples y usables genera una barrera de entrada que afecta negativamente al aprendizaje y a la experimentación con los lenguajes formales.

También, se detecta una fragmentación entre las distintas fases del proceso de modelado. Con frecuencia, las herramientas permiten crear diagramas gráficos, pero no ofrecen un soporte integrado para la validación de modelos mediante restricciones OCL, o la transformación automática hacia otros formatos, como código fuente o representaciones textuales. Esta falta de integración obliga a recurrir a múltiples aplicaciones, lo que repercute en la eficiencia y en la coherencia del flujo de trabajo, afectando negativamente en aquellos entornos profesionales donde la productividad es clave.

A partir de estas carencias, surge la necesidad de desarrollar un sistema que ofrezca la capacidad de modelar y verificar sistemas, integrando funcionalidades y características que ayuden a agilizar el proceso de diseño de los usuarios.

1.3. Objetivo

El objetivo principal de este trabajo conjunto consiste en el desarrollo de una herramienta CASE [4] moderna, de código abierto, usable e intuitiva, para el ámbito del modelado y diseño del software, y que tenga una clara orientación a la eficiencia y productividad tanto en entornos académicos como profesionales. Para lograr dicho objetivo, se plantean los siguientes objetivos secundarios:

1. Permitir la creación de diagramas de clase para modelado conceptual, de forma gráfica.
2. Permitir la conversión de archivos texto a modelos UML gráficos.
3. La creación y modificación de elementos del metamodelo UML.
4. Comprobar la sintaxis y establecer invariantes sobre los modelos.
5. La transformación de modelos UML a otros formatos, como código fuente o representaciones textuales.

En el capítulo 6 puede encontrarse una recopilación de los requisitos del sistema de forma más detallada.

Esta parte del trabajo se ha encargado principalmente del diseño, y desarrollo de la interfaz de usuario usable e intuitiva, junto a las funcionalidades de alto nivel principales, como la creación y edición de diagramas de clase, la representación gráfica de los modelos, etc.

1.4. Organización de la memoria

A continuación se describe brevemente la organización de la memoria, indicando el contenido principal de cada capítulo:

- **Capítulo 1: Introducción**
  Presenta el contexto, la motivación y los objetivos del trabajo, así como los requisitos funcionales y no funcionales de la herramienta desarrollada.
- **Capítulo 2: Marco teórico**
  Expone los fundamentos teóricos necesarios para el desarrollo del proyecto, incluyendo UML, OCL, lenguajes formales y gramáticas.

- **Capítulo 3: Estado del Arte**
  Analiza antecedentes, la importancia del modelado conceptual, la interacción persona-ordenador y las principales herramientas CASE existentes.
- **Capítulo 4: Tecnologías**
  Describe las tecnologías empleadas en el desarrollo de la aplicación, justificando su elección y explicando su papel en el proyecto.
- **Capítulo 5: Metodologías de trabajo**
  Detalla la metodología ágil adoptada (Scrum), la organización del equipo, la gestión de tareas y las fases del desarrollo.
- **Capítulo 6: Requisitos del sistema**
  Define los requisitos funcionales y no funcionales de la herramienta, así como las restricciones y supuestos considerados durante su desarrollo.
- **Capítulo 7: Modelado y diseño**
  Explica el proceso de modelado conceptual y de la interfaz de usuario, así como los diagramas y estructuras que sustentan la arquitectura de la aplicación.
- **Capítulo 8: Implementación y pruebas**
  Profundiza en la implementación de la herramienta, abordando aspectos técnicos, decisiones de diseño, pruebas realizadas y funcionalidades clave.
- **Capítulo 9: Conclusión**
  Recoge las conclusiones del trabajo, la experiencia adquirida y posibles líneas futuras de mejora y desarrollo.
- **Apéndice: Manual de usuario**
  Incluye un manual básico para el usuario, ejemplos de uso y referencias adicionales.

Capítulo 2

Marco teórico

El presente capítulo tiene como objetivo proporcionar los fundamentos teóricos que sustentan el desarrollo de la aplicación. Se abordan los conceptos esenciales relacionados con el modelado de software, en particular el Lenguaje de Modelado Unificado (UML), el Lenguaje de Restricciones OCL, y nociones fundamentales de lenguajes formales y gramáticas, necesarios para la interpretación y validación de modelos.

2.1. UML: Lenguaje de Modelado Unificado

El Lenguaje de Modelado Unificado (UML, *Unified Modeling Language*) [5] es un lenguaje gráfico utilizado para visualizar, especificar, construir y documentar los artefactos de un sistema software. UML no es un método de desarrollo en sí, sino un lenguaje de notación que puede integrarse en diferentes metodologías. Fue estandarizado por la OMG (Object Management Group) y desde entonces ha evolucionado hasta convertirse en el estándar por defecto para el modelado orientado a objetos [1].

UML permite representar distintos aspectos de un sistema a través de varios tipos de diagramas. Algunos de los elementos más utilizados en el modelado conceptual y estático los podemos encontrar en la Figura 2.1.

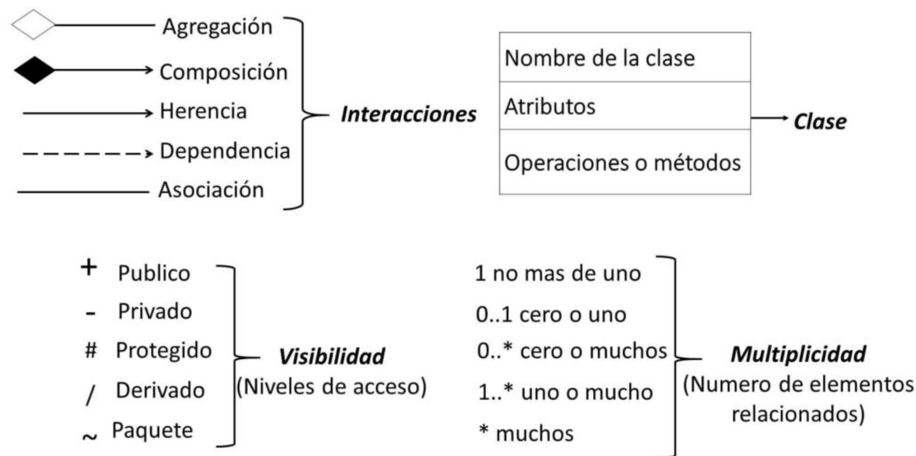

Figura 2.1 Lista de elementos UML, extraída de [6]

- **Clases**: Representan entidades del sistema, definiendo sus atributos y métodos. Son el componente básico del modelado orientado a objetos.
- **Atributos**: Propiedades o características que definen el estado de una clase.
- **Relaciones**: Vínculos entre clases. Pueden ser de varios tipos:
  - *Asociaciones*: Relación estructural entre dos clases.
  - *Generalización*: Herencia entre clases (relación padre-hijo).
  - *Agregación*: Relación todo-parte, donde una clase (todo) contiene a otra (parte), pero ambas pueden existir de forma independiente.
  - *Composición*: Relación todo-parte más estricta, donde la parte no puede existir sin estar relacionada con el todo.
- **Visibilidad**: Controla el acceso a los atributos y métodos de una clase. Puede ser pública, privada, protegida, de paquete o derivada.
- **Multiplicidad**: Indica cuántas instancias de una clase pueden estar relacionadas con una instancia de otra.
- **Diagramas de clases**: Representación estática que muestra las clases del sistema, sus atributos, métodos y relaciones.

En el contexto de la aplicación desarrollada, UML ha sido el marco base para permitir a

los usuarios construir modelos conceptuales mediante una interfaz gráfica. Este trabajo se centra en los diagramas de clase. Aunque existen otros tipos de diagramas, como puede ser el diagrama de componentes, estos quedan fuera del alcance de este proyecto.

2.2. OCL: Object Constraint Language

UML permite modelar la estructura y comportamiento de un sistema, sin embargo, no proporciona mecanismos suficientemente precisos para expresar reglas o restricciones complejas. Cuando nos referimos a reglas o restricciones, hablamos de condiciones que deben cumplirse en el modelo. Para ello se puede emplear OCL (Object Constraint Language), un lenguaje formal que permite definir condiciones sobre modelos UML de forma declarativa [7]. OCL permite expresar, entre otros:

- **Invariantes**: Condiciones que debe cumplir una clase en todo caso.
- **Precondiciones y postcondiciones**: Reglas que deben cumplirse antes y después de ejecutar una operación, respectívamente.
- **Restricciones sobre asociaciones y multiplicidades**: Condiciones que deben cumplirse en las relaciones entre las instancias de las clases del sistema.

Su sintaxis, basada en expresiones similares a las de lenguajes funcionales, permite una descripción formal pero legible. Por ejemplo:

Dada una entidad "Persona", con el atributo edad.

```
class Persona
attributes:
    edad: Integer
end
```

La siguiente expresión:

```
context Persona
    inv: self.edad >= 0
```

Indica que la edad de una persona debe ser siempre mayor o igual a cero. En la aplicación desarrollada, se ha incorporado soporte para expresiones OCL mediante un motor de validación basado en gramáticas formales, lo que permite verificar automáticamente la

consistencia de los modelos definidos por el usuario.

2.3. Lenguajes Formales y Gramáticas

El procesamiento de los modelos textuales en la aplicación requiere el análisis sintáctico de entradas definidas por el usuario. Para ello, se han aplicado conceptos fundamentales de teoría de lenguajes formales y gramáticas.

Lenguajes regulares y gramáticas

Un lenguaje formal es un conjunto de cadenas formadas a partir de un alfabeto y definidas por una gramática. Las gramáticas formales se clasifican típicamente en la jerarquía de Chomsky [8]:

- **Gramáticas regulares (Tipo 3)**: Son las más simples y se utilizan para definir lenguajes que pueden reconocerse mediante autómatas finitos. Se emplean, por ejemplo, en expresiones regulares y análisis léxico.
- **Gramáticas libres de contexto (Tipo 2)**: Se usan para describir la estructura sintáctica de la mayoría de los lenguajes de programación y modelado. Son reconocidas por autómatas de pila.
- **Gramáticas sensibles al contexto (Tipo 1)**: Más complejas, permiten definir lenguajes donde el contexto afecta la producción de cadenas.

En el proyecto, se ha utilizado la herramienta ANTLR4 (*Another Tool for Language Recognition*) para procesar la gramática del lenguaje utilizado por la herramienta USE (*UML-based Specification Environment*) [9]. ANTLR permite definir gramáticas y generar automáticamente analizadores sintácticos en varios lenguajes, incluido C++. Esto ha facilitado la integración de una capa textual avanzada para interpretar modelos, validar restricciones y generar código.

Parsing y validación

El *parsing* o análisis sintáctico consiste en leer una secuencia de tokens y construir una estructura (como un árbol de derivación) que represente su significado según una gramática.

En este caso, se ha implementado un analizador capaz de procesar entradas escritas en el lenguaje USE/OCL, generando estructuras internas sobre las que se puede aplicar validación semántica, generar código y detectar errores sintácticos de forma precisa.

2.4. Interacción Persona-Ordenador

La interacción persona-ordenador (IPO) es una disciplina que estudia cómo las personas interactúan con los sistemas informáticos, buscando optimizar esta interacción para mejorar la usabilidad, eficiencia y satisfacción del usuario [10]. En el contexto del desarrollo de herramientas de modelado, la IPO adquiere una relevancia especial, ya que la complejidad inherente a estas aplicaciones puede dificultar su adopción y uso efectivo. Para lograr esto se busca un enfoque centrado en el usuario, que considere sus necesidades, habilidades y limitaciones.

Los 10 principios clave de la IPO según Jakob Nielsen [11] son:

- **Visibilidad del estado del sistema**: El usuario debe ser capaz de entender en todo momento qué está ocurriendo en el sistema.
- **Correspondencia entre el sistema y el mundo real**: La interfaz debe utilizar conceptos, términos y metáforas familiares para el usuario.
- **Control del usuario y libertad**: Los usuarios deben sentirse en control de la aplicación, pudiendo deshacer acciones y navegar libremente.
- **Consistencia y estándares**: La interfaz debe ser coherente en su diseño y comportamiento, siguiendo convenciones establecidas.
- **Reconocimiento en lugar de recuerdo**: La interfaz debe minimizar la carga cognitiva, facilitando el acceso a opciones y funciones sin necesidad de memorizar información.
- **Prevención de errores**: La aplicación debe ayudar a los usuarios a evitar errores mediante un diseño cuidadoso.
- **Flexibilidad y eficiencia de uso**: La herramienta debe adaptarse a diferentes niveles de experiencia, permitiendo atajos para usuarios avanzados.
- **Diseño estético y minimalista**: La interfaz debe ser visualmente atractiva pero sin sobrecargar al usuario con información innecesaria.

- **Ayuda a los usuarios para reconocer, diagnosticar y recuperarse de errores**: Los mensajes de error deben ser claros y ofrecer soluciones prácticas.
- **Ayuda y documentación**: Aunque la interfaz debe ser intuitiva, es importante proporcionar documentación accesible para resolver dudas.

Capítulo 3

Estado del Arte

En este capítulo se examinan las herramientas CASE (*Computer-Aided Software Engineering*) existentes más notables actualmente en el mercado y más centradas en las fases de diseño y modelado de sistemas, indicando sus características que se pueden mejorar, así como sus puntos fuertes.

3.1. Antecedentes

UML surgió a mediados de los años 90 como una respuesta a la necesidad de estandarizar los métodos de modelado en ingeniería de software. Antes de su aparición, existían múltiples metodologías y notaciones, lo que generaba fragmentación y dificultades de comunicación.

El lenguaje fue desarrollado por Grady Booch, Ivar Jacobson y James Rumbaugh, consolidando sus metodologías previas (Booch, OOSE y OMT) en un lenguaje común. Posteriormente, se estandarizó, convirtiéndolo en la referencia para sistemas orientados a objetos.

3.2. Herramientas CASE actuales

Para el diseño y representación de los modelos se suele hacer uso de las conocidas herramientas CASE (del inglés, *Computer-Aided Software Engineering*) [4]. Hoy en día

podemos encontrar un catálogo muy amplio de estas aplicaciones, pero en este caso sería conveniente puntualizar las herramientas *Upper Case* puesto que es el tipo de herramienta CASE que se quiere desarrollar en este trabajo (son las que ofrecen soporte en la fase de planificación y diseño). En cuanto a las herramientas comerciales podemos destacar las tres siguientes:

- **Enterprise Architect** [12]: una herramienta comercial ampliamente utilizada en la industria para modelado UML, análisis de requerimientos y generación de código. Esta herramienta presenta una interfaz que puede llegar a ser muy abrumadora para nuevos usuarios, pero presenta una cantidad inmensa de funcionalidades además de las mencionadas anteriormente.

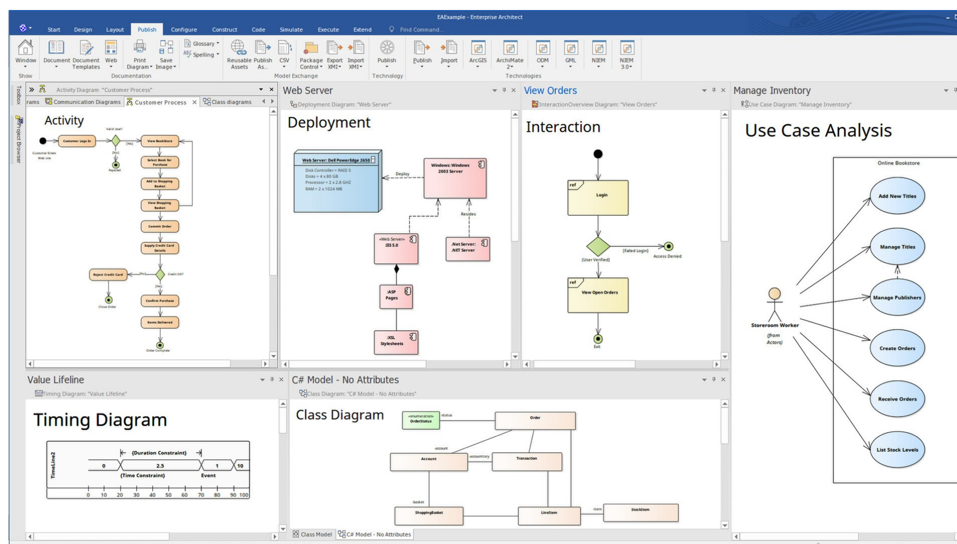

Figura 3.1 Captura de la herramienta Enterprise Architect, extraída de [13]

- **Visual Paradigm** [14]: orientada a facilitar el modelado visual con funcionalidades avanzadas como ingeniería directa e inversa, simulación de modelos para comprobar la calidad de estos, y exportación de diagramas. Visual Paradigm nos permite hacer muchos tipos de diagramas UML además del de clases. Pero el trabajo colaborativo con esta herramienta puede llegar a ser torpe, debido a que al ser archivos binarios, no se puede hacer uso de sistemas de control de versiones como Git.

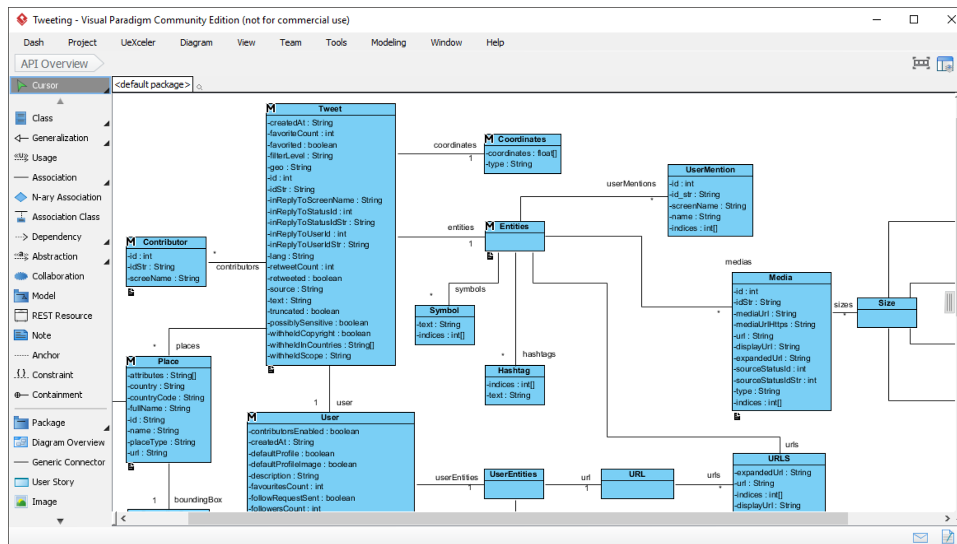

Figura 3.2 Captura de la herramienta Visual Paradigm, extraída de [15]

- **StarUML** [16]: al igual que Visual Paradigm, está orientada a diseñar el modelo mediante elementos gráficos, e integra una funcionalidad de ingeniería inversa para (generar un modelo en base a un código). Al contrario que Visual Paradigm, en la docuemntación de StarUML podemos encontrar qué utilizan para verificar el modelo creado. Para ello, hacen uso de las llamadas *Well-formedness rules*, que son reglas para garantizar que el modelo sea correcto sintácticamente.

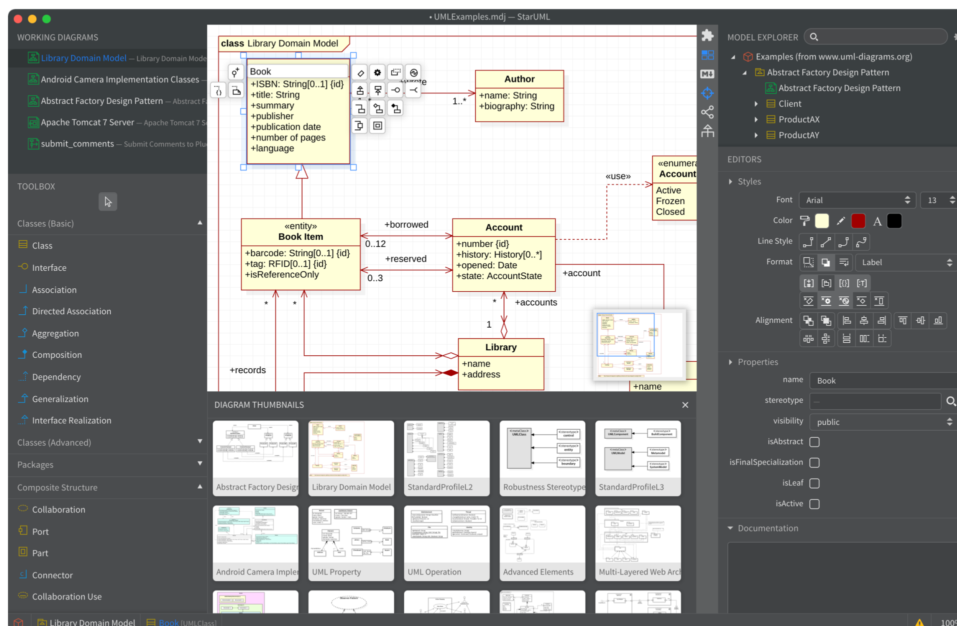

Figura 3.3 Captura de la herramienta StarUML, extraída de [17]

Con respecto a las herramientas de código abierto, podemos encontrar:

- **USE (*UML-based Specification Environment*)** [9]: una herramienta académica que permite validar modelos UML con restricciones OCL. Al contrario que las anteriores herramientas que se centran en el diseño del modelo mayormente, USE busca garantizar la consistencia léxica y semántica del modelo. La consistencia léxica es referida a aquellas posibles equivocaciones en la especificación del modelo, mientras que la semántica son aquellos fallos que aparecen al no cumplir con las reglas impuestas por UML, por ejemplo: que existan dos clases con un nombre idéntico en el mismo modelo.

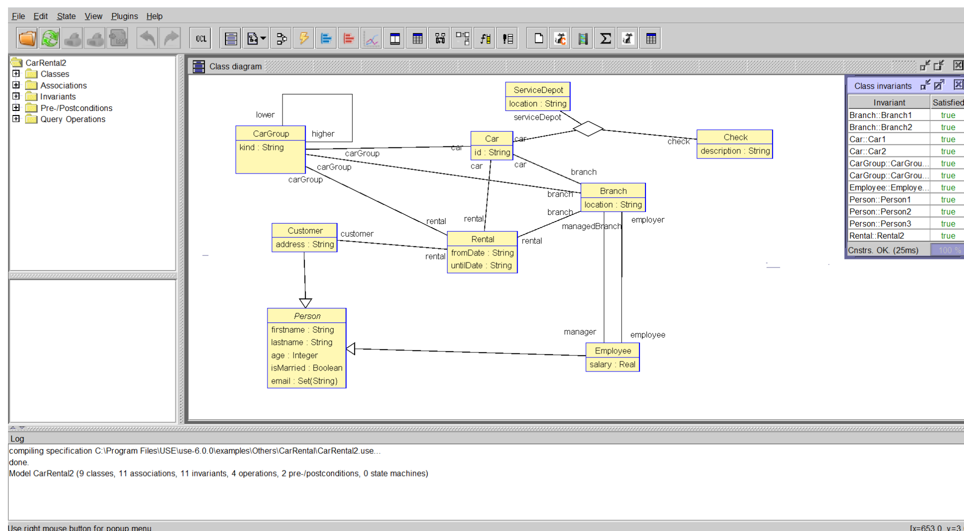

Figura 3.4 Captura de la herramienta USE

- **PlantUML** [18]: este programa permite diseñar numerosos tipos de diagramas, tanto los declarados en UML, como otros que pueden ser encontrados en su pagina oficial. Para realizar estos diagramas, PlantUML utiliza un lenguaje de marcado sencillo que permite a los usuarios describir los diagramas de forma textual. Esto facilita la creación y modificación de diagramas, así como su integración en archivos de código. Tambíen tienen disponibles plugins para editores de texto como Visual Studio Code o Atom, y para otros entornos de desarrollo.

Si bien estas herramientas aportan soluciones efectivas en muchos aspectos del modelado, también presentan limitaciones que deben ser consideradas. Por ejemplo, muchas herramientas comerciales requieren licencias costosas, lo que restringe su uso en entornos educativos o en pequeñas organizaciones. Otras herramientas de código abierto, como USE, pueden carecer de mantenimiento activo, presentar interfaces desactualizadas y poco intuitivas, o

no presentar funcionalidades que podríamos llegar a encontrar en una obtenida con licencia.

En particular, una de las deficiencias más comunes observadas en estas herramientas es la falta de integración de funcionalidades como el modelado visual, la validación formal y la generación automática de código. En muchos casos, las funcionalidades están fragmentadas, o la experiencia del usuario se ve comprometida por interfaces poco amigables y flujos de trabajo complicados. Además, algunas de estas aplicaciones no permiten fácilmente la exportación o interoperabilidad entre formatos, dificultando su adopción en entornos mixtos o proyectos colaborativos.

Capítulo 4

Tecnologías

Para el desarrollo de la aplicación propuesta se ha recurrido a un conjunto de tecnologías seleccionadas estratégicamente por su robustez, compatibilidad con los objetivos del proyecto, rendimiento y soporte comunitario. A continuación, se describen brevemente cada una de ellas, junto con la justificación de su elección.

4.1. C++

C++ [19] ha sido el lenguaje de programación principal utilizado en el desarrollo de la aplicación. Se trata de un lenguaje compilado, de propósito general y orientado a objetos, ampliamente reconocido por su eficiencia y control sobre los recursos del sistema. La elección de C++ se justifica por varios motivos:

- **Rendimiento**: C++ permite la creación de aplicaciones altamente eficientes, lo cual es importante para mantener la fluidez de la interfaz gráfica y el procesamiento de modelos.
- **Compatibilidad con Qt**: Qt, el framework elegido para el desarrollo de la interfaz, está escrito en C++, lo que asegura una integración nativa y óptima.
- **Madurez y soporte**: C++ cuenta con una amplia comunidad de desarrolladores y una gran cantidad de bibliotecas disponibles, lo que facilita la resolución de problemas y la extensión de funcionalidades.

4.2. Qt Framework

Qt [20] es un framework multiplataforma para el desarrollo de interfaces gráficas y aplicaciones de escritorio. Proporciona un conjunto de herramientas robustas para el diseño de GUI (*Graphical User Interfaces*). Las razones para su elección incluyen:

- **Desarrollo visual avanzado**: Qt facilita la creación de interfaces modernas, interactivas y bien estructuradas, gracias a componentes como QWidgets, QML y layouts flexibles.
- **Multiplataforma**: La capacidad de compilar la aplicación en distintos sistemas operativos sin reescribir el código favorece la portabilidad y distribución de la herramienta.
- **Documentación extensa y comunidad activa**: Qt cuenta con abundante documentación y una comunidad muy activa, lo cual es clave durante el desarrollo y mantenimiento del software.

4.3. Qt Creator

Qt Creator [21] ha sido el entorno de desarrollo integrado (IDE) utilizado en el proyecto. Es el IDE oficial del framework Qt, diseñado específicamente para facilitar el desarrollo en C++, o Python, con Qt. Ofrece herramientas como autocompletado inteligente, depurador gráfico, diseño visual de interfaces, y una gestión simplificada de proyectos CMake.

Sus principales ventajas son:

- **Integración total con Qt**: Permite crear interfaces gráficas con facilidad, diseñar formularios visuales y conectarlos directamente con el código fuente.
- **Facilidad de configuración y compilación**: Qt Creator gestiona automáticamente los entornos de compilación, facilitando el trabajo en distintas plataformas.
- **Productividad y eficiencia**: Las funcionalidades integradas, como la navegación rápida por código, depuración en vivo y soporte para pruebas, aceleran significativamente el desarrollo.

4.4. Figma

Figma [22] es una herramienta de diseño de interfaces y prototipado colaborativo basada en la nube. Se ha utilizado en la fase inicial del proyecto para el diseño conceptual de la interfaz gráfica, permitiendo representar visualmente el flujo de pantallas, la disposición de componentes y la interacción del usuario con el sistema.

Las razones de su uso incluyen:

- **Diseño centrado en el usuario**: Figma permite crear prototipos visuales interactivos, lo que facilita validar decisiones de diseño antes de su implementación.
- **Colaboración en tiempo real**: Ideal para proyectos donde participan varios miembros en la etapa de diseño, ya que permite la edición simultánea.
- **Exportación y especificaciones visuales**: Figma facilita la extracción de medidas, colores y componentes para su implementación directa en Qt.

4.5. ANTLR4

ANTLR4 (*Another Tool for Language Recognition*) [23] es una herramienta poderosa para la construcción de analizadores léxicos y sintácticos a partir de gramáticas definidas. En este proyecto, ANTLR4 se ha utilizado para interpretar y procesar la gramática del lenguaje de especificación *USE* (UML-based Specification Environment), permitiendo así la generación y validación de modelos desde una sintaxis textual.

Las motivaciones para esta elección son:

- **Compatibilidad con gramáticas formales**: ANTLR permite definir de manera clara y modular la sintaxis de lenguajes específicos, como el de USE, facilitando su interpretación y análisis.
- **Generación de analizadores en C++**: ANTLR soporta la generación de código en múltiples lenguajes, incluyendo C++, lo que permite su integración directa en la aplicación.
- **Documentación y ejemplos**: Su amplia documentación y comunidad hacen de ANTLR una opción confiable para el procesamiento de lenguajes personalizados.

4.6. Git

Git [24] es el sistema de control de versiones distribuido que se ha utilizado a lo largo del desarrollo del proyecto para gestionar el código fuente, realizar seguimiento de cambios y facilitar la colaboración en etapas de desarrollo iterativo.

Las razones principales para su elección son:

- **Control y trazabilidad**: Permite mantener un historial completo de los cambios realizados en el proyecto, facilitando la identificación de errores, la reversión de modificaciones problemáticas y la comprensión de la evolución del sistema.
- **Integración con plataformas colaborativas**: Git se integra fácilmente con plataformas como GitHub, utilizada en este proyecto, lo que permite alojar el repositorio de forma remota y habilitar funcionalidades adicionales como revisión de código y seguimiento de incidencias.
- **Estándar de la industria**: Git es ampliamente utilizado en el mundo profesional, lo que garantiza un flujo de trabajo alineado con las mejores prácticas de ingeniería de software.

4.7. Trello

Trello [25] es una herramienta de gestión de proyectos basada en tableros Kanban que se ha utilizado durante la planificación y seguimiento del desarrollo del proyecto. Su interfaz visual basada en tarjetas permite organizar tareas de manera clara y flexible.

Las ventajas de su utilización han sido las siguientes:

- **Visualización del progreso**: La disposición de las tareas en columnas organizadas según la metodología ágil seguida, que será explicada más adelante.
- **Facilidad de uso**: Trello no requiere configuración compleja y su curva de aprendizaje es muy baja, permitiendo su uso inmediato sin distracciones técnicas.
- **Organización eficiente de tareas**: Cada tarjeta puede incluir descripciones, listas de subtareas, etiquetas, fechas de vencimiento y comentarios, lo que facilita la documentación de cada etapa del desarrollo.

- **Gestión del tiempo y prioridades**: El uso de etiquetas de colores y la asignación de fechas límite permite priorizar tareas y mantener un ritmo constante de trabajo.

Capítulo 5

Metodologías de trabajo

Para la organización, planificación y ejecución del desarrollo del proyecto se ha adoptado un enfoque ágil basado en la metodología Scrum [26]. Este marco de trabajo ágil se ha consolidado como una de las metodologías más utilizadas en la industria del software gracias a su capacidad para adaptarse a los cambios, fomentar la colaboración continua y entregar valor de forma incremental y sostenida.

Aunque Scrum está concebido originalmente para equipos multidisciplinares de entre tres y nueve miembros con roles claramente definidos (*Product Owner, Scrum Master y Development Team*), en el caso de este proyecto, al tratarse de un equipo reducido de dos personas, se ha realizado una adaptación del marco a nuestras necesidades y limitaciones particulares, manteniendo su esencia pero flexibilizando ciertos elementos organizativos.

5.1. Adaptación de Scrum al equipo

En nuestro contexto, no se ha establecido una división formal de roles, sino que ambos integrantes han compartido de forma colaborativa las responsabilidades propias del equipo de desarrollo y de la gestión del proyecto. Las decisiones han sido tomadas de manera conjunta, promoviendo la comunicación continua y la responsabilidad compartida. Esta adaptación ha resultado efectiva para mantener la agilidad del proceso, minimizar la sobrecarga organizativa y garantizar el cumplimiento de los objetivos.

5.2. Sprints

El desarrollo del proyecto se ha organizado en sprints, que son iteraciones temporales de duración fija (en nuestro caso, de aproximadamente dos semanas), al final de las cuales se debía entregar una versión funcional o un avance significativo del sistema. Cada sprint ha comenzado con una planificación en la que se seleccionaban las tareas a abordar y finalizaba con una revisión y retrospectiva para analizar los resultados obtenidos y reflexionar sobre posibles mejoras en el proceso.

5.3. Product Backlog

Se ha utilizado un Product Backlog como lista priorizada de funcionalidades, requisitos, correcciones y tareas pendientes del proyecto. Este backlog ha estado en constante evolución a medida que se obtenía nueva información o se identificaban nuevas necesidades, permitiendo una gestión flexible de los objetivos. Las tareas han sido redactadas en forma de ítems concretos y desglosables, y se han etiquetado para facilitar su organización (por ejemplo, ‘UI/UX‘, ‘Backend‘, ‘Test‘, etc.).

5.4. Sprint Backlog

A partir del Product Backlog, se ha construido un Sprint Backlog al inicio de cada sprint, seleccionando las tareas que se iban a abordar específicamente durante dicha iteración. Estas tareas han sido asignadas entre los dos miembros del equipo de forma equilibrada, teniendo en cuenta la carga de trabajo y la naturaleza de cada ítem. La visualización y gestión del sprint backlog se ha realizado mediante la herramienta Trello, como se puede observar en la Figura 5.1, organizada con listas tipo Kanban (“To do”, “In progress”, “Done”), lo que ha permitido hacer seguimiento en tiempo real del avance del sprint.

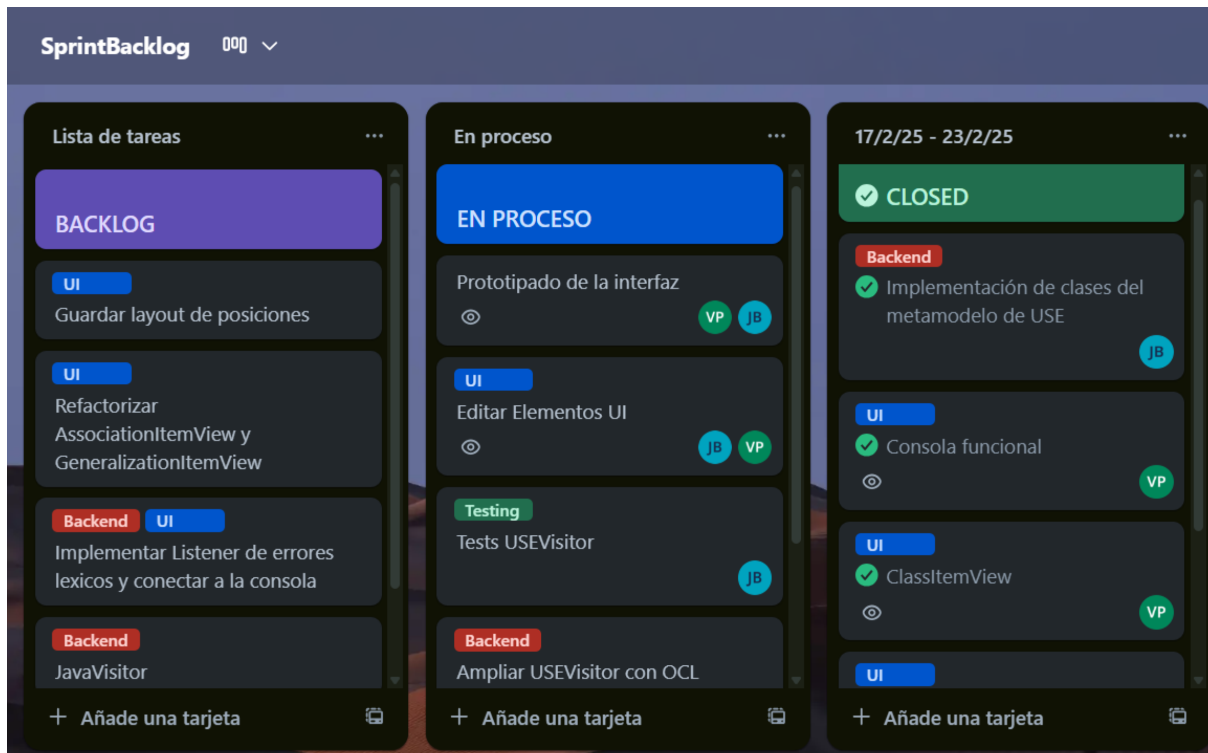

Figura 5.1 Organización de los sprints en Trello.

5.5. Reuniones y seguimiento

Dado el tamaño reducido del equipo, las reuniones han sido sustituidas por comunicaciones continuas e informales, ya sea en persona o mediante mensajería instantánea. Estas comunicaciones han cumplido la función de las *daily stand-ups*, permitiendo mantener alineadas las tareas, resolver bloqueos de forma rápida y ajustar prioridades si era necesario. Además, al final de cada sprint se ha realizado una revisión informal para analizar los entregables del sprint, así como una retrospectiva en la que se discutían aspectos a mejorar en los siguientes ciclos. Por último, cada dos/tres sprints se han organizado reuniones con el tutor para realizar demos y actualizar el seguimiento del proyecto.

5.6. Fases del desarrollo

Aunque el enfoque Scrum es iterativo e incremental, se pueden identificar las siguientes fases en el desarrollo del proyecto:

- **Fase de planificación**: Ha incluido la definición del alcance del proyecto, el estudio y elección de las tecnologías, la elaboración de prototipos iniciales en Figma, y la creación del Product Backlog con una primera estimación de tareas.
- **Fase de diseño**: se ha detallado la arquitectura del sistema, se diseñaron los primeros diagramas conceptuales y se definió la estructura general de la interfaz y la lógica interna.
- **Fase de desarrollo incremental**: A lo largo de varios sprints se han implementando progresivamente las distintas funcionalidades del sistema, incluyendo la interfaz gráfica, la integración con ANTLR para la carga y validación de archivos USE y los mecanismos de generación de código Java.
- **Fase de validación y pruebas**: Se han dedicado sprints específicos a la revisión funcional del sistema, la corrección de errores, pruebas con diferentes modelos y ajustes de usabilidad en la interfaz.
- **Fase de documentación y cierre**: Finalmente, se ha elaborado la documentación técnica y académica del proyecto, se preparó la entrega y se realizó una última retrospectiva sobre el proceso completo.

5.7. Ventajas del enfoque adoptado

El uso de Scrum, incluso adaptado a un equipo pequeño, ha permitido:

- Mantener una buena organización de tareas y prioridades.
- Detectar y corregir errores de manera temprana.
- Adaptarse a cambios de requisitos o mejoras inesperadas.
- Fomentar una planificación realista, centrada en entregables concretos en cada sprint.
- Promover la motivación mediante la visibilidad del progreso alcanzado iterativamente.

Capítulo 6

Requisitos del sistema

Los requisitos del sistema definen las funcionalidades y características que la herramienta debe cumplir para satisfacer las necesidades del usuario y los objetivos del proyecto. Estos requisitos se dividen en dos categorías principales: requisitos funcionales y no funcionales. Los requisitos no funcionales establecen criterios sobre el rendimiento, la usabilidad, la fiabilidad y el comportamiento del sistema. Mientras que los requisitos funcionales describen las funciones específicas que el sistema debe realizar.

Por lo que partiendo del objetivo general presentado en la introducción, en este capítulo se presentan los requisitos que definirán nuestra herramienta CASE.

6.1. Requisitos funcionales

A partir del objetivo general, se identifican los siguientes requisitos funcionales que deberá cumplir la herramienta:

RF 1 Transformar archivos `.use` en modelos gráficos: transformar el texto escrito en gramática USE/OCL en un modelo gráfico UML.

- **RF 1**.1 Manejar posibles excepciones

  - **RF 1**.1.1 Manejar posibles errores sintácticos de OCL: se deben implementar mecanismos para detectar y reportar errores en las expresiones OCL, proporcionando

información clara sobre la naturaleza del error y su ubicación.

RF 1.1.2 Manejar posibles errores de manejo de archivos: se deben implementar mecanismos para detectar y reportar errores en la carga y guardado de archivos, proporcionando información clara sobre la naturaleza del error y su ubicación.

RF 1.2 Poder recargar archivos para ver cambios si se han realizado modificaciones en el archivo original abierto.

RF 2 Transformar modelos gráficos en archivos `.use`: transformar un modelo gráfico UML en texto escrito en gramática USE/OCL.

RF 2.1 Crear y sobreescribir archivos `.use`: se deben implementar mecanismos para crear y sobreescribir archivos `.use` a partir de los modelos gráficos, garantizando la integridad y consistencia de los datos.

RF 3 Ofrecer herramientas de modelado UML

RF 3.1 Permitir manejar clases UML

RF 3.1.1 Permitir establecer tipo de clase (abstracta, normal)

RF 3.1.2 Permitir crear clases UML

RF 3.1.3 Permitir eliminar clases UML

RF 3.1.4 Permitir modificar clases UML

RF 3.1.4.1 Cambiar nombre y posición

RF 3.1.4.2 Añadir, eliminar y modificar atributos

RF 3.1.4.3 Añadir, eliminar y modificar operaciones

RF 3.2 Permitir manejar relaciones UML

RF 3.2.1 Permitir crear relaciones entre clases UML

RF 3.2.2 Permitir eliminar relaciones entre clases UML

RF 3.2.3 Permitir modificar relaciones entre clases UML

RF 3.2.3.1 Cambiar nombre, multiplicidad y visibilidad de la relación

RF 4 Permitir manejar clases asociación UML

RF 4.1 Permitir crear clases asociación UML

RF 4.2 Permitir eliminar clases asociación UML

RF 4.3 Permitir modificar clases asociación UML

RF 5 Permitir manejar enumerados UML

RF 5.1 Permitir crear enumerados UML

RF 5.2 Permitir eliminar enumerados UML

RF 5.3 Permitir modificar enumerados UML

RF 5.3.1 Cambiar nombre

RF 5.3.2 Añadir, eliminar y modificar literales

RF 6 Ofrecer herramientas de modelado OCL:

RF 6.1 Permitir manejar restricciones OCL en clases.

RF 6.2 Permitir manejar condiciones OCL para las operaciones.

RF 7 Permitir guardar y cargar la estructura visual del diagrama: se deben implementar mecanismos para guardar y cargar la estructura visual del diagrama, garantizando que se mantenga las posiciones de los elementos gráficos.

RF 8 Transformar modelos gráficos en código Java: se deben implementar mecanismos para transformar el metamodelo en código Java, generando la estructura de clases, atributos y métodos de acuerdo con las especificaciones del lenguaje.

RF 9 Permitir deshacer y rehacer cambios significativos en el modelo.

6.2. Requisitos no funcionales

Además de las funcionalidades descritas, la herramienta deberá cumplir con una serie de requisitos no funcionales que garanticen su calidad y sostenibilidad:

RNF 1 Abrir archivos del formato '`.use`'.

RNF 2 Guardar modelo en formato '`.use`'.

RNF 3 Tener una interfaz organizada, simple y funcional.

RNF 4 Mostrar el estado del sistema en el proceso de carga de modelo, generación de código

y errores.

Capítulo 7

Modelado y diseño

Este capítulo describe el proceso de modelado y diseño llevado a cabo durante el desarrollo de la aplicación, abarcando tanto los aspectos visuales como los conceptuales y estructurales.

7.1. Diseño de la Interfaz de Usuario

En el diseño de la aplicación se ha buscado adaptar y seguir diversos principios de Interacción Persona-Ordenador (IPO), los cuales están más presentes en aplicaciones web, con el objetivo de mejorar la usabilidad, accesibilidad y eficiencia de la herramienta de escritorio.

7.1.1. Consistencia visual y funcional

Los elementos del diagrama, como clases, enumeraciones y asociaciones, mantienen un formato homogéneo en su representación, incluyendo forma, tipografía y disposición de atributos y operaciones. Asimismo, se emplea un esquema de colores coherente para cada tipo de elemento, permitiendo un reconocimiento rápido. Este esquema utilizado está basado en la estandarización de facto presente en la industria: morado/rosado para las clases abstractas y azul para las normales. Esta consistencia se conserva tanto en el modo claro como en el modo oscuro.

7.1.2. Personalización y accesibilidad

Se ofrece la posibilidad de trabajar en modo claro o modo oscuro, lo que permite adaptar la visualización a las preferencias del usuario y reducir la fatiga visual. Los colores han sido seleccionados para mantener la legibilidad en ambos modos y evitar posibles confusiones debido al contraste.

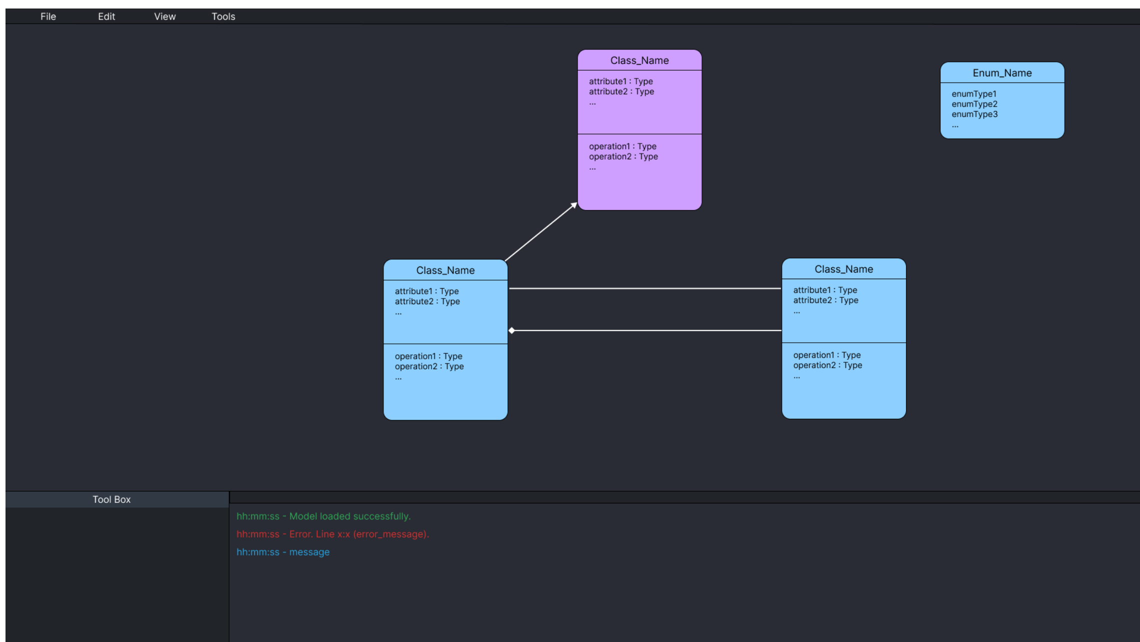

Figura 7.1 Ventana principal en modo oscuro.

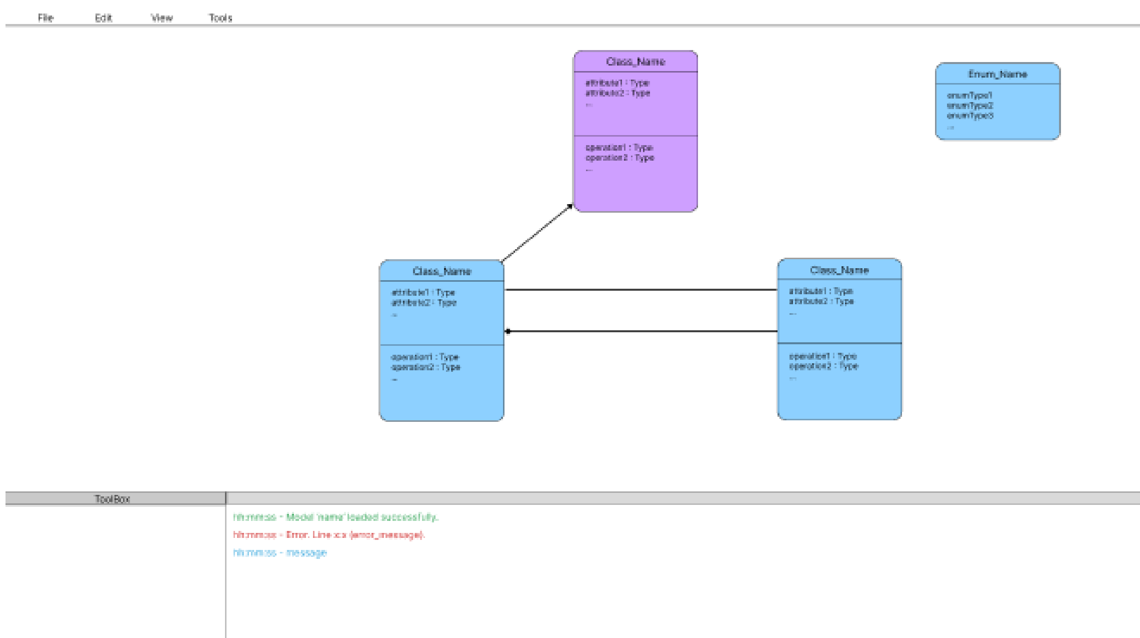

Figura 7.2 Ventana principal en modo claro.

7.1.3. Visibilidad del estado del sistema

Se ha integrado una consola para mostrar mensajes de estado relevantes (Figura 7.3), tales como confirmación de carga del modelo, errores con indicación de línea y mensajes informativos. Esto sigue la heurística de visibilidad del estado del sistema, manteniendo al usuario informado de las acciones y resultados.

```
11:07:24   Loading 'Biblioteca.use'
11:07:25   Model 'Library' was succesfully loaded.
11:07:47   Model already contains element named: Magazine
11:08:16   Model already contains element named: Book
11:08:31   The file was successfully saved.
11:08:36   Java code succesfully generated.
```

Figura 7.3 Captura de la consola mostrando mensajes de estado del sistema.

7.1.4. Organización y jerarquía de la información

La disposición de los diagramas evita solapamientos, garantizando claridad visual. La interfaz se divide en áreas diferenciadas: un área principal para los diagramas, un área inferior para los mensajes del sistema y una zona de herramientas; cumpliendo con el principio de *espacialidad funcional*.

Las ventanas de edición se han estructurado mediante la proximidad de los elementos que siguen una tarea común, para que causen una relación mental en el usuario, cumpliendo así el principio de *compatibilidad por proximidad*.

7.1.5. Simplicidad y reducción de carga cognitiva

Se evita la sobrecarga de iconos y elementos innecesarios en la pantalla. Los nombres de clases, atributos y operaciones son renderizadas con una fuente sencilla y con un tamaño de letra adecuado, facilitando su comprensión inmediata.

7.1.6. Prevención y recuperación de errores

Para la prevención de errores se ha decidido implementar un sistema para rehacer y deshacer acciones que provoquen un cambio significativo en el modelo, como puede ser la creación, o modificación, de una clase. También se ha incluido una confirmación para dichas acciones que alteren el estado del sistema sin la posibilidad de deshacer o rehacer dicha acción. Pueden ser, por ejemplo, cerrar la ventana de editar sin guardar, borrar un atributo de una clase, etc.

7.2. Diseño Conceptual del Modelo de Datos

En el proceso de modelado conceptual de la aplicación, ha sido fundamental identificar las entidades principales y las relaciones entre ellas. Es estas entidades se puede observar el uso de la preprosición *meta-*, cuyo significado, proviniente del griego, se centra en una abstracción del concepto con el que se utiliza la preposición. Estas entidades conforman la base del metamodelo que describe la estructura interna de la herramienta y su comportamiento, para su definición se ha partido de la gramática y del modelo aceptado por la herramienta USE. A continuación se explica el significado de estas:

MetaModel Constituye la entidad raíz que engloba la definición completa del modelo conceptual. Actúa como contenedor principal de todas las metaentidades, incluyendo metaclases, metaasociaciones y metatipos. Representa el marco de referencia sobre el que se estructuran los elementos y sus relaciones.

MetaClass Describe la estructura y el comportamiento de las entidades que se quieren modelar. Contiene información sobre atributos, operaciones y restricciones asociadas a una clase. Su definición es abstracta, sirviendo de plantilla para la creación de instancias concretas en el modelo final.

MetaAttribute Representa las propiedades de una `MetaClass`, definiendo sus características y restricciones. Cada `MetaAttribute` está asociado a una clase específica y puede incluir información sobre su tipo, visibilidad y valor por defecto. Y puede tener una expresión OCL

derivada de otro `MetaAttribute` y una expresión para establecer su inicialización en el diagrama de objetos.

MetaOperation Representa las operaciones que pueden ser realizadas sobre en una `MetaClass`. Cada metaoperación está asociada a una metaclase específica y puede incluir información sobre su nombre, parámetros, tipo de retorno, definición y su pre/post condición. Además, puede tener una expresión OCL que defina su comportamiento.

MetaVariable Representa una variable asociada a una `MetaOperation`, permitiendo almacenar información temporal durante la ejecución del modelo. Cada metavariable está vinculada a una `MetaOperation`, o un `MetaAssociationEnd` específica y puede incluir información sobre su nombre, tipo y valor actual.

MetaConstraint Define las restricciones que pueden aplicarse a `MetaClass`, especificando condiciones que deben cumplirse para que un modelo sea considerado válido. Cada metaconstraint está asociado a una metaclase específica y puede incluir información sobre su nombre, su expresión OCL y si es existencial (una restricción existencial solo comprueba que existe al menos una isntancia que cumpla la condición).

MetaStateMachine Representa una máquina de estados asociada a una `MetaClass`, definiendo los estados posibles y las transiciones entre ellos. Permite modelar el comportamiento dinámico de las entidades en el sistema. En nuestro sistema no son utilizadas, por lo que solo guardan su definición en un string.

MetaAssociation Define las relaciones existentes entre dos o más `MetaClass`. Incluye información sobre cardinalidades, roles y nombre de la asociación, permitiendo representar de forma precisa cómo se vinculan las distintas entidades del modelo.

MetaAssociationEnd Representa un extremo de una `MetaAssociation`, definiendo su rol y cardinalidad en la relación. Cada metaasociación puede tener uno o más extremos, que a su vez están vinculados a metaclases específicas. Esta entidad permite modelar de forma precisa cómo se conectan las distintas entidades en el metamodelo. También contiene varios

booleanos para indicar si el conjunto es ordenado, único, navegable o si es la unión de otros conjuntos (`MetaAssociationEnd`).

MetaMultiplicity y MetaMultiplicityRange La clase `MetaMultiplicity` simplemente funciona como un contenedor para guardar los rangos de la multiplicidad de las asociaciones en el metamodelo. Por otro lado, la clase `MetaMultiplicityRange` se utiliza para definir rangos de cardinalidad, permitiendo especificar intervalos como "1..*"ø"0..1".

MetaAssociationClass Es una entidad que combina las propiedades de una `MetaClass` con las de una `MetaAssociation`. Se utiliza para modelar relaciones que, además de conectar entidades, poseen atributos u operaciones propias, comportándose como una clase intermedia.

MetaType Define los tipos de datos que pueden ser empleados en atributos y parámetros de operaciones dentro de las metaclases. Su función es garantizar la coherencia y la validez de los datos utilizados en el modelo.

CollectionType Define los tipos de colección que pueden ser utilizados en los atributos y parámetros de operaciones dentro de las metaclases. Permitiendo modelar estructuras de datos más complejas y flexibles. Dicha colección almacena múltiples elementos del mismo tipo.

TupleType y TuplePart Define los tipos de tupla que pueden ser utilizados en los atributos y parámetros de operaciones dentro de las metaclases. Permitiendo modelar estructuras de datos que agrupan diferentes tipos de elementos.

SimpleType Define los tipos de datos simples que pueden ser utilizados en atributos y parámetros de operaciones dentro de las metaclases. `SimpleType` generaliza a `Integer`, `String`, `Boolean`, `Real`, `MetaClass` y `MetaEnum`.

MetaEnum y MetaEnumElement La clase `MetaEnum` representa un conjunto de valores literales, permitiendo definir tipos de datos enumerados dentro del metamodelo. Cada `MetaEnum` puede contener múltiples `MetaEnumElement`, que son los valores individuales de la enumeración.

OCL Expression Esta entidad va a ser la encargada de representar y abordar las numerosas expresiones OCL existentes. Dicha expresión va a poder llegar a tener una expresión padre. Y un número fijo de expresiones hijas según el tipo de la expresión padre. Dichas expresiones van a poder ser encontradas en: la definición de los finales de las metaasociaciones, en las condiciones de las operaciones, en la definición de un atributo y en las restricciones aplicadas a las metaclases.

El diagrama del metamodelo, representado en la Figura 7.4, representa gráficamente las entidades fundamentales y las relaciones que sustentan la arquitectura conceptual de la aplicación. Su objetivo es ofrecer una visión clara y estructurada de cómo se organizan los elementos que intervienen en el modelado. En este marco del trabajo no nos centraremos en el modelo de las expresiones OCL, ya que no se ha usado directamente en la implementación de la interfaz de usuario. En resumen, el diagrama no solo refleja la estructura jerárquica del metamodelo, sino también la lógica de interacción entre sus componentes, proporcionando una base sólida para la implementación y validación del sistema.

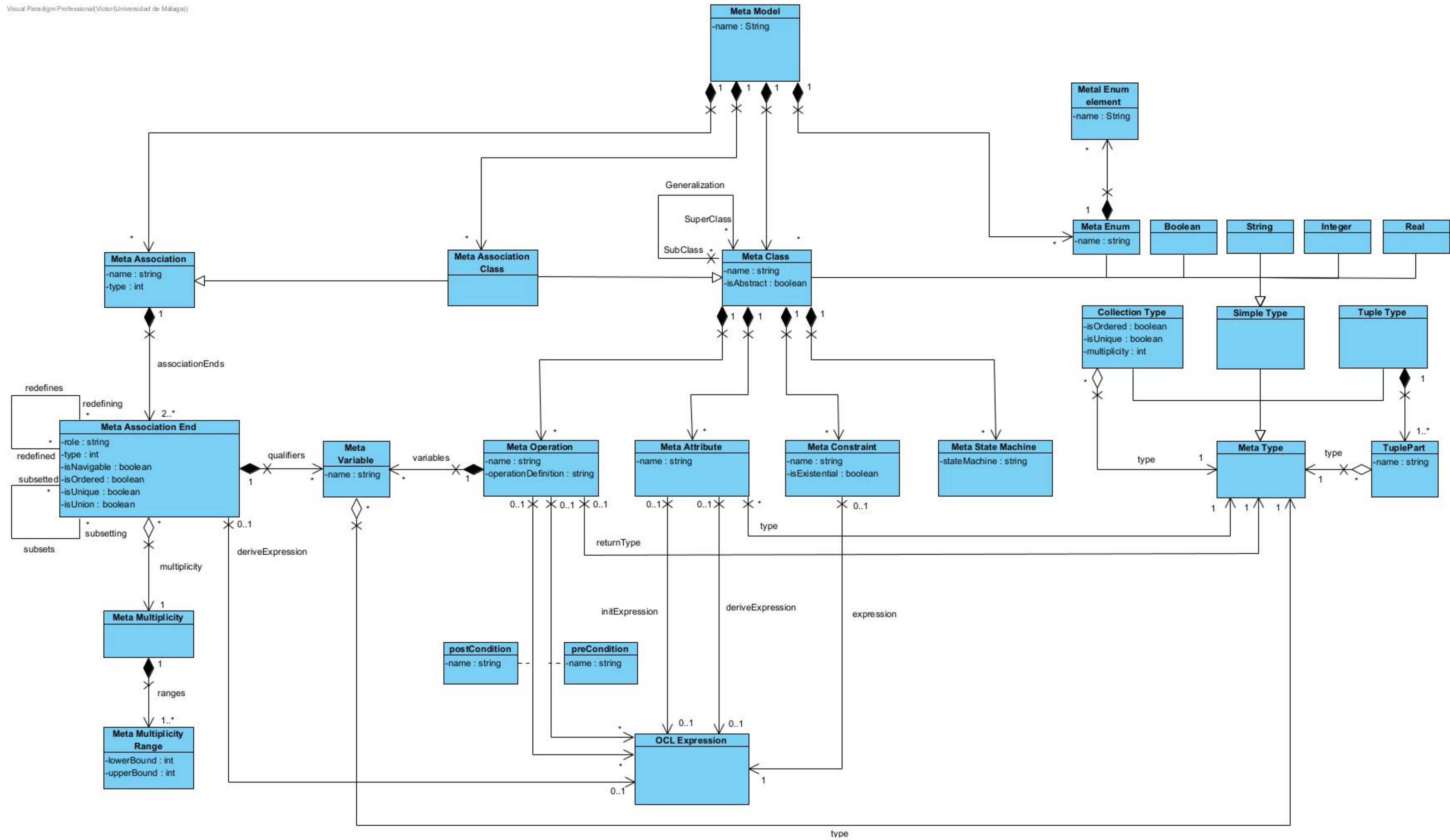

Figura 7.4 Modelo conceptual de UML aceptado por USE modificado.

7.3. Modelado de la Interfaz de Usuario

El diagrama presentado en la Figura 7.5 representa la estructura y relaciones de los elementos que componen la interfaz de usuario de la aplicación. A diferencia de un diagrama conceptual, cuyo propósito es abstraer el dominio del problema sin entrar en detalles de implementación, este diagrama se sitúa en un nivel más cercano al código, describiendo directamente las clases y componentes gráficos empleados, muchos de ellos pertenecientes a la biblioteca Qt.

El núcleo del sistema gráfico está constituido por la clase `QGraphicsItem`, la cual actúa como superclase de la mayoría de los elementos visuales. A partir de ella se derivan distintas especializaciones que representan los diferentes componentes visuales del editor, como:

- **BoxItemView** y **GeneralizationItemView**: elementos para la representación gráfica de clases y relaciones de generalización.
- **ClassItemView** y **EnumItemView**: vistas asociadas a las entidades `MetaClass` y `MetaEnum`, encargadas de mostrar gráficamente clases y enumeraciones.
- **AssociationClassItemView** y **AssociationItemView**: elementos que representan gráficamente asociaciones simples y clases-asociación, vinculadas a sus correspondientes modelos (`MetaAssociation` y `MetaAssociationClass`). Podemos observar una pequeña diferencia entre cómo relacionamos las clases asociacion con las asociaciones y las clases respecto a los modelos. En los *ItemView* la clase asociación se relaciona con las dos, en vez de heredar de ambas como hace el modelo. El porqué de esta decisión será explicado más adelante en el capítulo de implementación.

En un nivel superior de la jerarquía, se encuentra la clase `ModelGraphicsScene`, que hereda de `QGraphicsScene` y actúa como contenedor de todos los elementos gráficos. Esta escena es visualizada a través de `ModelGraphicsView`, una especialización de `QGraphicsView` que incorpora funcionalidades específicas para la interacción con el usuario y se conecta directamente con el modelo de datos (`MetaModel`).

Este diagrama refleja, por tanto, una arquitectura *Model-View*, que será explicada más adelante, adaptada al contexto del editor UML. Las clases de tipo *View* (por ejemplo, `ClassItemView`) mantienen una asociación con sus contrapartes del modelo (como `MetaClass`),

asegurando la sincronización entre la representación visual y la información lógica del modelo.

Al trabajar con un framework específico como Qt, este diseño no se limita a describir conceptos abstractos, sino que define explícitamente cómo se implementan y relacionan los componentes gráficos, lo que lo aleja del enfoque puramente conceptual y lo sitúa en el ámbito del diseño de la interfaz concreta.

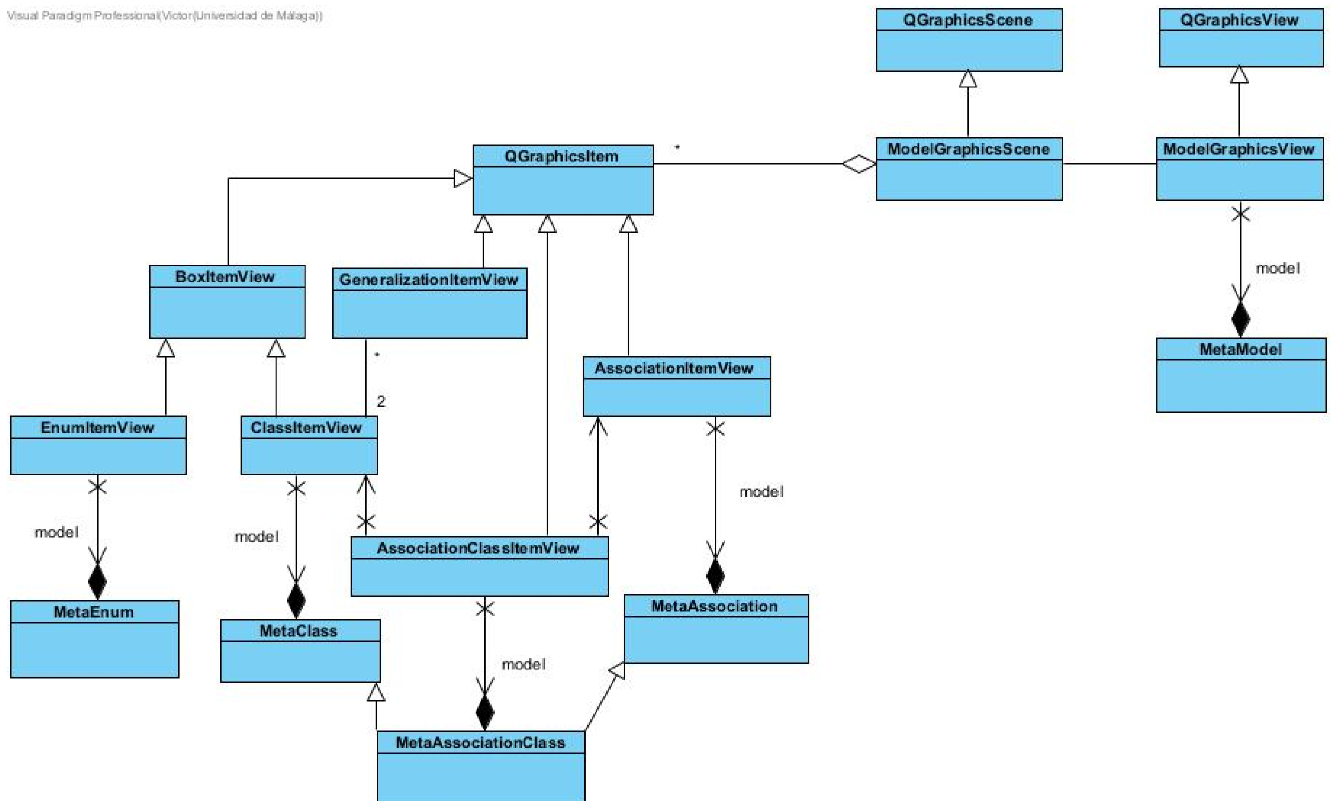

Figura 7.5 Diagrama de clases para la interfaz de usuario basado en Qt

Capítulo 8

Implementación y pruebas

En este capítulo se explicarán los detalles relacionados con la implementación de los elementos de la interfaz de usuario. Explicando con detalle las decisiones tomadas y los beneficios que nos aportan.

8.1. Implementación de las ventanas

La primera fase de la implementación de la aplicación se ha centrado en la construcción de las distintas ventanas que conforman la interfaz gráfica. Entre ellas destaca la `MainWindow`, que constituye el núcleo de interacción principal con el usuario, y un conjunto de ventanas secundarias destinadas a la edición de elementos UML y a la gestión de alertas o diálogos de confirmación. La correcta estructuración de estas ventanas ha resultado fundamental para garantizar la usabilidad y la coherencia de la aplicación.

8.1.1. La ventana principal: MainWindow

Como podemos observar en la Figura 8.1, el resultado final de la implementación de la `MainWindow` ha resultado ser muy similar al diseño original, aunque con algunas diferencias visuales en los componentes. Además se ha optado finalmente por añadir un `QLineEdit` para mostrar y editar el nombre del modelo, frente a la primera opción que era mediante un botón de la caja inferior de herramientas.

Figura 8.1 Estructura básica de la ventana principal (`MainWindow`).

La clase `MainWindow` fue diseñada como la interfaz central de la aplicación, actuando como contenedor del área de trabajo donde se despliega el editor gráfico, así como de los menús, barras de herramientas y elementos de navegación adicionales. Esta clase hereda de `QMainWindow`, lo que proporciona de forma nativa una estructura jerárquica basada en menús, barras de estado y áreas de widgets centrales, simplificando enormemente la construcción de la ventana.

Uno de los aspectos clave en la implementación de la `MainWindow` y de las demás ventanas es el uso del `namespace Ui`, generado automáticamente por Qt Designer. Este espacio de nombres actúa como puente entre la descripción declarativa de la interfaz y su representación programática en C++. En la práctica, el archivo `.ui` define la disposición de los elementos gráficos, mientras que el compilador de Qt (`uic`) [27] transforma esta descripción en una clase auxiliar accesible desde el código. De este modo, la clase `MainWindow` puede interactuar de manera directa con los widgets definidos en el archivo `.ui`, reduciendo el acoplamiento entre lógica de negocio y diseño visual. Esta separación permite, además, una mayor flexibilidad a la hora de realizar cambios en la interfaz sin afectar al código fuente principal.

Cabe señalar que para la representación del área de trabajo gráfico se ha optado por el uso del componente `QGraphicsView`, en combinación con `QGraphicsScene` y `QGraphicsItem`. Esta decisión se fundamenta en las amplias ventajas que ofrece dicho módulo de Qt, entre las que destacan la capacidad de renderizar figuras de manera vectorial, el soporte para

espacios virtualmente infinitos, la existencia de un sistema de coordenadas completamente integrado y la gestión de un gran número de eventos de interacción de manera nativa.

Aunque en este apartado únicamente se introduce de manera general su uso, la explicación detallada de su implementación y de los elementos que lo componen será tratada en profundidad más adelante. Con ello se pretende separar la explicación de la estructura básica de ventanas de los aspectos más avanzados del motor gráfico de la aplicación.

Otro elemento que encontramos es la implementación de la consola, que está ubicada en la ventana principal, usando el componente *QTextEdit*. Para acceder a dicha consola se ha implementado un manejador estático, del que se hablará más adelante.

Y por último, pero no menos importante, encontramos una declaración estática de una pila de acciones. El comportamiento de este componente se explicará detalladamente en el punto donde se desarrolla la implementación de acciones reversibles.

8.1.2. Conexión de acciones con la barra superior

Uno de los mecanismos más utilizados en la `MainWindow` es la barra superior de menús y acciones, Figura 8.2. Cada opción del menú (crear, abrir, guardar, exportar, etc.) está asociada a una acción de Qt (`QAction`), la cual se conecta mediante el sistema `signal-slots`. Este sistema [28] se basa en la interacción entre objetos que lanzan y reaccionan a eventos. Dichos eventos pueden ser lanzados por un objeto que tenga la macro `Q_OBJECT`, haciendo uso de `Q_SIGNAL`. Y para reaccionar a estos eventos, se utilizan los `Q_SLOTS`. Ambos deberán ser conctados mediante la función `connect`.

Por ejemplo, la acción `New model` puede estar conectada con un slot que abre el cuadro de diálogo de edición de clases, mientras que la acción `Save Model` se vincula con un slot encargado de serializar el metamodelo en el formato correspondiente. De este modo, se logra una interacción fluida y natural entre la interfaz visual y la lógica funcional.

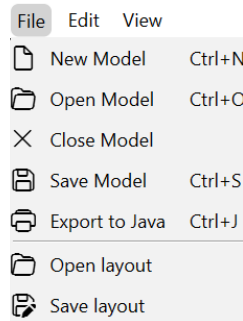

Figura 8.2 Acciones principales en la `Barra superior de herramientas`.

8.1.3. Ventanas de edición de elementos UML

Además de la `MainWindow`, la aplicación dispone de diversas ventanas secundarias destinadas a la edición de elementos específicos del modelo UML. Estas ventanas heredan de `QDialog`, lo que las convierte en cuadros de diálogo modales o no modales según sea necesario. Cada una de ellas permite al usuario introducir, modificar o validar la información asociada a entidades como clases, enumeraciones, asociaciones o atributos.

Al igual que en la ventana principal, estas clases hacen uso del `namespace Ui`, garantizando que la interfaz definida en Qt Designer se integre de manera sencilla con el código. La lógica de cada diálogo se implementa en la clase correspondiente (*...EditDialog.h/cpp*), donde se gestionan las señales emitidas por los widgets (botones, cajas de texto, listas desplegables, etc.) y se conectan a los `slots` que encapsulan la funcionalidad deseada. Por ejemplo, al confirmar la creación de una nueva clase UML, el cuadro de diálogo emite una señal con la información introducida, la cual es capturada en la `MainWindow` para actualizar tanto el modelo de datos como la representación gráfica.

Este enfoque modular permite desacoplar el flujo de interacción con el usuario de la lógica central de la aplicación. Cada ventana actúa como un componente independiente, responsable de validar y transmitir la información que el usuario introduce, lo que facilita la escalabilidad y el mantenimiento del proyecto.

Normalmente, la estructura en código de estas ventanas es la siguiente: Primero, se realizan todas las conexiones y desconexiones de las acciones necesarias para controlar el comportamiento de estas. Si el elemento a editar presenta una lista de algún otro tipo de elemento, como pueden ser los atributos en una clase, la ventana tendrá métodos para añadir y eliminar este tipo de elementos. Además, tendrá un método para guardar los cambios y

otro para cancelar los cambios.

Respecto a todas las ventanas de edición que nos podemos encontrar en este proyecto, las más importantes son *ClassEditDialog* y *AssociationEditDialog*, mostradas en la Figura 8.3 y 8.4, respectívamente. Estas ventanas se diferencian del resto de ventanas de edición en la forma de guardar los cambios. Mientras que el resto simplemente modifica el puntero compartido con la información del metaelemento a editar, las ventanas de edición de clases y de asociaciones guardan una acción reversible en la pila declarada en `MainWindow`. Este comportamiento es explicado más adelante en la sección 8.4.

Name: ExampleClass
Abstract

Attributes:

Name	Type
attribute1	Integer
attribute2	String

Add
Remove

Operations:

Name	Type
operation1	Boolean
operation2	Real

Add
Remove

Save
Cancel

Figura 8.3 Ventana de edición de una clase.

Name: ExampleAssociation

Association type: Association

AssociationEnd 1

Role: exampleRole1

Multiplicity: 1

Type: ExampleClass

Visibility: Public Private Protected Package

AssociationEnd 2

Role: exampleRole2

Multiplicity: 0..*

Type: ExampleClass2

Visibility: Public Private Protected Package

Save Cancel

Figura 8.4 Ventana de edición de una asociación.

Por último, la comprobación de la cardinalidad de las asociaciones, se realiza mediante la siguiente expresión regular:

```
1   static const QRegularExpression
     ↪ regex(R"((\d+|\*)(\.\.(\d+|\*))?(,(\d+|\*)(\.\.(\d+|\*))?)*)");
```

Se le aplica a los campos de texto en los que se escriben la multiplicidad. Esta expresión regular permite validar cadenas que representan rangos de cardinalidad en el formato

aceptado por UML, como "1", "0..*", "1,3..5", etc. Si la cadena introducida no coincide con el patrón definido por la expresión regular, se considera inválida y se notifica al usuario mediante una ventana de alerta.

8.1.4. Ventanas de alerta y confirmación

Las ventanas de alerta y confirmación, que ocupan un papel muy importante en el apartado de usabilidad, están implementadas mediante `QMessageBox`. Estas ventanas cumplen una función crítica: guiar al usuario en situaciones excepcionales (errores de validación, acciones destructivas, confirmaciones previas a operaciones sensibles, etc.). Su implementación se realizó siguiendo criterios de simplicidad y claridad, buscando minimizar la ambigüedad en la comunicación con el usuario. Para ello se hizo uso de un widget nativo de Qt llamado *QMessageBox*. En relación a dicho widget encontramos también varios subtipos, pero solo se han usado: *QMessageBox::critical* (para mostrar errores) y *QMessageBox::question* (para pedir la confirmación del usuario).

En el caso de los errores, se lanza la ventana mediante la estructura *try/catch*, o usando la instrucción condicional *if*. Esto garantiza que, por ejemplo, al intentar eliminar un elemento del modelo, y dicha acción no sea reversible, se pueda invocar automáticamente un cuadro de confirmación antes de proceder con la acción. Esta filosofía de interacción asegura un mayor control por parte del usuario y reduce la probabilidad de errores irreversibles.

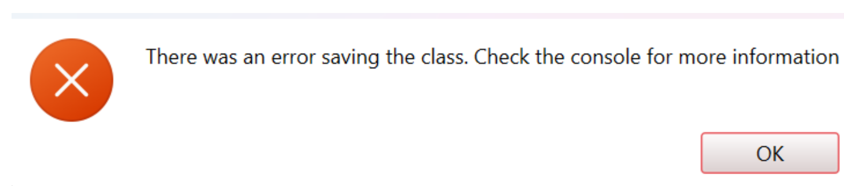

Figura 8.5 Ventana de error.

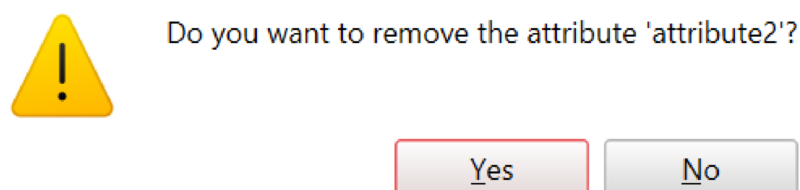

Figura 8.6 Ventana de confirmación.

En el siguiente código se puede observar las llamadas a las funciones ya implementadas por Qt, `QMessageBox::critical` y `QMesssageBox::warning`. Y en la segunda función podemos ver que se devuelve la respuesta obtenida para ser procesada por las ventanas de edición.

```
#include <utils/MessageBox.h>
#include <QMessageBox>

void showExceptionMessageBox(QString title, QString msg, QWidget* parent){
    QMessageBox::critical(parent, title, msg);
}

QMessageBox::StandardButton showQuestionMessageBox(QString title, QString msg,
  ↪ QWidget* parent){
    auto reply = QMessageBox::warning(parent, title, msg, QMessageBox::Yes |
      ↪ QMessageBox::No);
    return reply;
}
```

8.2. Implementación de `ModelGraphicsView`

Uno de los componentes más relevantes de la implementación de la interfaz gráfica es la clase `ModelGraphicsView`. Esta clase constituye una generalización desarrollada a partir de `QGraphicsView`, adaptada a las necesidades específicas de la aplicación. Su objetivo principal es extender la funcionalidad básica del visor gráfico de Qt, proporcionando un conjunto de eventos personalizados que optimizan la interacción del usuario con los elementos del metamodelo representados en la escena.

La elección de `QGraphicsView` como clase base se debe a las amplias ventajas comentadas anteriormente que ofrece este componente dentro del framework de Qt: capacidad de renderizado vectorial, soporte para espacios de coordenadas virtualmente ilimitados, gestión eficiente de múltiples objetos gráficos y un sistema de eventos bien estructurado que facilita la interacción directa con la escena. No obstante, las necesidades concretas del proyecto requerían un control más preciso sobre determinadas interacciones del usuario, lo que motivó la creación de esta subclase. Este componente no contiene los elementos a renderizar

en sí, sino que contiene una escena gráfica (QGraphicsScene) que contiene a estos y se encarga de pintarlos.

8.2.1. Eventos personalizados en ModelGraphicsView

La clase ModelGraphicsView incorpora varios eventos redefinidos y ampliados para cubrir los siguientes casos de uso fundamentales:

Zoom mediante la rueda del ratón

Una de las primeras funcionalidades añadidas fue el soporte para zoom interactivo utilizando la rueda del ratón. Aunque QGraphicsView proporciona mecanismos básicos para transformar el área de visualización, se implementó un evento específico para controlar de forma precisa la escala aplicada a la vista.

Este evento captura la señal de desplazamiento vertical de la rueda (wheelEvent) y aplica un factor de escala incremental o decremental sobre la transformación de la vista. De este modo, el usuario puede acercar o alejar dinámicamente la escena gráfica sin necesidad de recurrir a controles externos. El resultado es una experiencia de navegación más intuitiva, especialmente útil en diagramas UML de gran tamaño donde la vista completa no cabe en pantalla.

El siguiente fragmento de código ilustra la implementación de este evento, donde se define un factor de escala al 15 % y se ajusta la transformación de la vista en función de la dirección del desplazamiento de la rueda. Dicho zoom siempre estará en el rango newScale–maxScale.

```
void ModelGraphicsView::wheelEvent(QWheelEvent *event){
    constexpr double scaleFactor = 1.15;
    double factor = (event->angleDelta().y() > 0) ? scaleFactor : (1.0 / scaleFactor);

    double newScale = currentScale * factor;
    if (newScale >= minScale && newScale <= maxScale) {
        scale(factor, factor);
        currentScale = newScale;
    }}
```

Movimiento del viewport

El segundo evento implementado en `ModelGraphicsView` corresponde al desplazamiento del área visible de la escena, comúnmente denominado movimiento del *viewport*. Para lograrlo, se redefine el comportamiento asociado a los eventos de arrastre del ratón (`mousePressEvent`, `mouseMoveEvent` y `mouseReleaseEvent`), habilitando al usuario a mover la cámara virtual de la vista al mantener pulsado un botón del ratón y arrastrar.

Esta funcionalidad resulta esencial para facilitar la exploración de diagramas de gran escala, en los que se requiere moverse de forma ágil entre diferentes áreas sin perder el contexto global del modelo. El siguiente fragmento de código muestra la implementación del evento de pulsación del ratón, donde se activa el modo de arrastre al detectar la pulsación del botón central o izquierdo del ratón.

```
1  void ModelGraphicsView::mousePressEvent(QMouseEvent *event){
2      if(event->button() == Qt::MiddleButton || event->button() == Qt::LeftButton){
3          viewport()->update();
4          setDragMode(QGraphicsView::ScrollHandDrag);
5  
6          QMouseEvent fakeEvent(
7              QEvent::MouseButtonPress,
8              event->position(),
9              event->globalPosition(),
10             Qt::LeftButton, Qt::LeftButton, Qt::NoModifier
11         );
12         QGraphicsView::mousePressEvent(&fakeEvent);
13     }else{
14         QGraphicsView::mousePressEvent(event);
15     }
16 }
```

Evento de finalización de movimiento

Además del arrastre en sí mismo, se implementó un evento adicional para detectar cuándo el usuario deja de mover el viewport. Este evento, gestionado a través de `mouseReleaseEvent`, permite ejecutar acciones posteriores al desplazamiento, como actualizar indicadores de posición, recalcular referencias de coordenadas o activar funcionalidades auxiliares.

Con ello se consigue un control más granular sobre la navegación, garantizando que la aplicación responda adecuadamente a la finalización de los movimientos y mantenga la coherencia del estado interno del sistema.

```
void ModelGraphicsView::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::MiddleButton) {
        viewport()->update();
        setDragMode(QGraphicsView::NoDrag);

        QMouseEvent fakeEvent(
            QEvent::MouseButtonPress,
            event->position(),
            event->globalPosition(),
            Qt::LeftButton, Qt::LeftButton, Qt::NoModifier
        );
        QGraphicsView::mouseReleaseEvent(&fakeEvent);
    } else {
        QGraphicsView::mouseReleaseEvent(event);
    }
}
```

Eliminación de elementos mediante la tecla Suprimir

Finalmente, una funcionalidad crítica añadida a la clase `ModelGraphicsView` es la gestión de la eliminación de elementos de la escena gráfica a través de la tecla `Supr`. Esta característica permite que el usuario pueda seleccionar uno o varios elementos dentro del diagrama y eliminarlos de forma inmediata, mejorando la eficiencia del modelado y evitando la necesidad de menús adicionales.

El evento se implementa redefiniendo `keyPressEvent`, comprobando si la tecla presionada corresponde a la tecla de borrado y, en ese caso, eliminando los elementos seleccionados de la escena.

```
void ModelGraphicsView::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Delete) {
        QList<QGraphicsItem *> selectedItems = scene()->selectedItems();
```

```
5           try{
6               auto scene = dynamic_cast<ModelGraphicsScene*>(this->scene());
7               for (QGraphicsItem *item : selectedItems) {
8                   if(auto classItemView = qgraphicsitem_cast<ClassItemView*>(item)){
9                       RemoveMetaClassCommand* removeClassCommand =
10                          new RemoveMetaClassCommand(
11                              classItemView, scene, this->model
12                          );
13                      MainWindow::undoStack->push(removeClassCommand);
14                  }else if(auto enumItemView = qgraphicsitem_cast<EnumItemView*>(item)){
15                      RemoveMetaEnumCommand* removeMetaEnumCommand =
16                          new RemoveMetaEnumCommand(
17                              enumItemView, scene, this->model
18                          );
19                      MainWindow::undoStack->push(removeMetaEnumCommand);
20                  }
21                  else if(auto associationItemView =
                    ↪  qgraphicsitem_cast<AssociationItemView*>(item)){
22                      RemoveMetaAssociationCommand* removeAssociationCommand =
23                          new RemoveMetaAssociationCommand(
24                              associationItemView, scene, this->model
25                          );
26                      MainWindow::undoStack->push(removeAssociationCommand);
27                  }
28                  else if(auto generalizationItemView =
                    ↪  qgraphicsitem_cast<GeneralizationItemView*>(item)){
29                      RemoveMetaGeneralizationCommand* removeGeneralizationCommand =
30                          new RemoveMetaGeneralizationCommand(
31                              generalizationItemView, scene, nullptr
32                          );
33                      MainWindow::undoStack->push(removeGeneralizationCommand);
34                  }
35              }
36          }catch(int err){
37              qDebug() <<"code " << err;
38          }
39      } else {
40          QGraphicsView::keyPressEvent(event); // Propaga el evento si no es Delete
41      }
```

```
42 }
```

8.2.2. Extensión de `QGraphicsScene`: `ModelGraphicsScene`

La clase `ModelGraphicsScene` constituye una extensión de la clase base `QGraphicsScene`, adaptada a las necesidades específicas de la herramienta. Su diseño busca centralizar la gestión de los elementos gráficos y proporcionar un conjunto de señales y operaciones adicionales que facilitan la interacción entre el modelo de datos y la vista. A continuación, se detallan sus principales características.

Gestión de señales personalizadas

Uno de los principales aportes de la clase es la definición de nuevas señales que notifican cambios relevantes en la escena. Estas señales permiten que otros componentes de la aplicación reaccionen a eventos sin necesidad de acoplarse directamente a la lógica interna de la vista:

- `itemMoved(QGraphicsItem * item, const QPointF& pos)`: se emite cuando un ítem de la escena es movido, notificando tanto la referencia al objeto como su nueva posición. Esto resulta útil para sincronizar la posición gráfica con el modelo lógico.
- `editAssociation(AssociationItemView * association)`: se dispara cuando una asociación es editada, facilitando que el controlador abra diálogos de edición o actualice propiedades del modelo subyacente.

Para emitir estas señales de forma explícita, se proporcionan los métodos `emitMoveSignal` y `emitEditAssociationSignal`, lo que permite encapsular la lógica de notificación dentro de la escena.

Soporte para portapapeles

La clase incorpora un portapapeles gráfico mediante un puntero a `ItemViewClipboard`. Esta estructura permite copiar, cortar y pegar elementos de la escena de manera coherente, manteniendo tanto las referencias gráficas como los modelos de metadatos asociados.

El uso de `setClipboard` y `setClipboardModel` permite inicializar el portapapeles con la referencia al modelo conceptual y garantizar que los elementos pegados mantienen

coherencia con la estructura de datos de alto nivel.

Mapa de elementos gráficos

`QGRaphicsScene` contiene una lista de los ítems contenidos en la escena, pero no dispone de métodos para poder acceder a estos de manera específica. Para facilitar esto, la clase mantiene un mapa interno denominado `modelItemViewElementsMap`. Este mapa asocia claves únicas (tipo `std::string`) con punteros a objetos gráficos (`QGraphicsItem*`). Gracias a esta estructura, es posible recuperar o eliminar elementos de forma eficiente:

- `getModelItemView(const std::string& key)`: devuelve la referencia al ítem gráfico identificado por la clave.
- `addModelItemView(const std::string& key, QGraphicsItem *item)`: añade un nuevo ítem al mapa y lo registra como parte de la escena.
- `removeModelItemView(const std::string& key)`: elimina el ítem asociado a la clave tanto del mapa como de la escena.

Gracias a esta extensión, la escena no solo actúa como contenedor de elementos gráficos, sino que también se convierte en un mediador activo entre el modelo de metadatos y la representación visual, mejorando la modularidad y escalabilidad de la herramienta.

8.3. Implementación de los `QGraphicsItem` personalizados

Una parte fundamental de la implementación de la herramienta consiste en la creación de `QGraphicsItem` personalizados, denominados *ItemView*. Estos elementos permiten representar visualmente las entidades del meta-modelo UML en la escena gráfica, manteniendo una relación estrecha con las estructuras internas de datos y garantizando la sincronización entre modelo y vista.

Antes de entrar en detalle sobre los ítems implementados, sería conveniente puntualizar los aspectos más importantes de la clase `QGraphicsItem`. Primero, cabe destacar el método `paint`, que se llama automáticamente desde la escena y se encarga de pintar el ítem en esta. Después, encontramos el método `boundingRect` que es un método que devuelve un rectángulo en cuyo interior debe estar nuestro ítem. Este método se utiliza sobre todo para comprobar si el ítem necesita ser pintado de nuevo o no. Y también podemos destacar sus

métodos para devolver la posición del ítem relativa a la escena.

Los principales *ItemView* implementados son: `ClassItemView`, `AssociationItemView`, `AssociationClassItemView`, `GeneralizationItemView`, `EnumItemView` y `BoxItemView`. A continuación se detalla el rol de cada uno de ellos.

BoxItemView

La clase`BoxItemView` actúa como una clase base para aquellos elementos que comparten una representación rectangular en pantalla, como las clases o los enumerados. Proporciona métodos genéricos para la gestión de dimensiones mínimas, posición, ancho y alto, así como una implementación de `boundingRect`. Además, redefine el método `itemChange` para destacar visualmente el borde de los elementos cuando son seleccionados en la interfaz. El método a destacar de esta clase sería `calcualteMinimumSize`, el cual es reimplementado tanto en la clase como en el enum. Ejemplo de la implementación en la clase `ClassSItemView`. Podemos observar que primero se calcula la altura y la anchura que va a tener el nombre de la clase, y si tiene atributos u operaciones se hace lo mismo para cada uno. Finalmente todas estas medidas se han sumado junto a los *paddings* establecidos, y dando lugar a las dimensiones mínimas que debe tener la clase.

```
1  void ClassItemView::calculateMinimumSize(){
2      QFontMetrics fm(QFont("Arial", 13, QFont::Bold));
3      int minHeight = fm.height() + NAME_PADDING + ATTS_PADDING;
4      int classNameWidth =
        ↪  fm.horizontalAdvance(QString::fromStdString(this->model->getName())) +
        ↪  PADDING;
5      this->setMinWidth(qMax(int(this->getDimensions().x()), classNameWidth));
6
7      fm = QFontMetrics(QFont("Arial", 10, QFont::StyleNormal));
8
9      for(const auto& pair : this->model->getAttributes()){
10         int attrWidth =
            ↪  fm.horizontalAdvance(QString::fromStdString(pair.second->toString())) +
            ↪  HORIZONTAL_PADDING;
11         this->setMinWidth(qMax(int(this->getMinDimensions().x()), attrWidth));
12
13         minHeight += ATTS_HEIGHT;
```

```
14      }
15
16      if(!this->model->getOperations().empty()){
17          minHeight += 2*ATTS_PADDING;
18      }
19
20      for(const auto& pair: this->model->getOperations()){
21          int attrWidth =
            ↪ fm.horizontalAdvance(QString::fromStdString(pair.second->toString())) +
            ↪ HORIZONTAL_PADDING;
22          this->setMinWidth(qMax(int(this->getMinDimensions().x()), attrWidth));
23
24          minHeight += ATTS_HEIGHT;
25      }
26
27      this->setMinHeight(minHeight+ATTS_PADDING);
28      this->setDimensions(this->getMinDimensions());
29  }
```

ClassItemView

La clase `ClassItemView` hereda de `BoxItemView` y representa las clases UML. Contiene un puntero al modelo correspondiente (`MetaClass`), gestiona las vistas de asociaciones, generalizaciones y asociaciones-clase relacionadas con la metaclase, y redefine eventos de ratón para permitir su interacción directa en la escena. En el método `paint` se dibuja la representación gráfica de la clase, incluyendo su nombre y compartimentos.

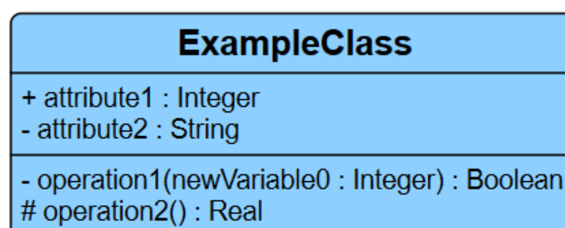

Figura 8.7 Representación visual de una clase UML.

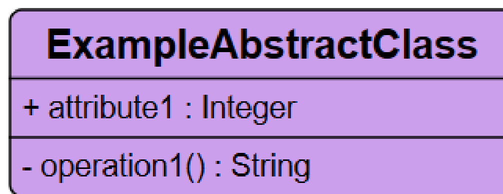

Figura 8.8 Representación visual de una clase abstracta UML.

EnumItemView

De forma análoga a las clases, la clase `EnumItemView` hereda de `BoxItemView` y se encarga de representar gráficamente los enumerados UML. Permite calcular tamaños mínimos en función del contenido textual, accede a su modelo interno (`MetaEnum`) y redefine eventos de ratón para soportar interacciones como selección o edición.

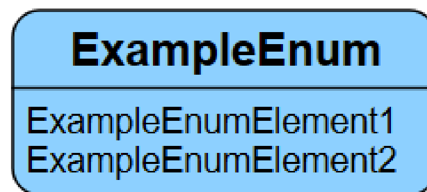

Figura 8.9 Representación visual de un enumerado UML.

AssociationItemView

La clase `AssociationItemView` representa gráficamente las asociaciones entre clases. Implementa un `paint` especializado para dibujar líneas, flechas y rombos en función del tipo de asociación, así como un método `updatePosition` que recalcula las posiciones de sus extremos cuando las clases conectadas se mueven en la escena. Además, incorpora lógica para gestionar asociaciones múltiples entre las mismas clases mediante desplazamientos (`offset`) y admite la conexión con asociaciones-clase.

En este ítem, podemos encontrar varios métodos a destacar. El método `drawDiamond` emplea conceptos básicos de geometría analítica y trigonometría para calcular las coordenadas de los vértices de un rombo (o diamante) que se dibuja al final de una línea de asociación. La idea fundamental consiste en obtener la dirección de la línea y, a partir de ella, construir

puntos desplazados angularmente que definan la figura.

```
QPointF AssociationItemView::drawDiamond(QLineF &line, QPainter *painter, bool
    filled){
    double angle = std::atan2(-line.dy(), line.dx()) - M_PI_2;
    qreal diamondSize = 8;
    QPointF head(line.pointAt(0.99));

    QPointF diamondP1 = head + QPointF(std::sin(angle + M_PI / 6) * diamondSize,
        std::cos(angle + M_PI / 6) * diamondSize);
    QPointF diamondP2 = head + QPointF(std::sin(angle - M_PI / 6) * diamondSize,
        std::cos(angle - M_PI / 6) * diamondSize);

    QPointF center = (diamondP1 + diamondP2) / 2;
    QPointF diamondP3 = 2 * center - head;

    QPolygonF diamondHead;
    diamondHead << head << diamondP1 << diamondP3 << diamondP2;

    if(filled) painter->setBrush(lineColor);

    painter->drawPolygon(diamondHead);
    return diamondP3;
}
```

En primer lugar, se calcula el ángulo de orientación de la línea mediante la función `atan2`, que devuelve el ángulo en radianes formado por el vector de la línea respecto al eje x. Posteriormente, se desplaza dicho ángulo en $-\frac{\pi}{2}$ para obtener una base perpendicular, lo cual permite ubicar los vértices laterales del diamante.

La determinación de los puntos se realiza con el uso de funciones trigonométricas:

$$x = \sin(\theta \pm \tfrac{\pi}{6}) \cdot s, \qquad y = \cos(\theta \pm \tfrac{\pi}{6}) \cdot s$$

donde θ es el ángulo de orientación calculado, y s corresponde al tamaño del diamante (`diamondSize`). Estos desplazamientos se aplican al punto `head`, que representa el extremo de la línea, generando así los vértices P_1 y P_2.

El vértice opuesto P_3 se obtiene calculando el punto simétrico de `head` respecto al centro

del segmento $\overline{P_1P_2}$:

$$P_3 = 2 \cdot \text{center} - \text{head}.$$

Y el método `getNearestEdgeIntersection`, que se encarga de buscar qué lado de la clase es el más adecuado para situar el inicio y el final de la línea de la asociación; y `updatePosition`, que se utiliza para actualizar los puntos de las asociaciones cuando se ha movido una de sus clases, o se ha cambiado una clase, etc.

AssociationClassItemView

La `AssociationClassItemView`, al igual que su modelo de datos, combina características de los `ItemView` respectivos de las asociaciones y de las clases. En vez de heredar de ellas, como se hacía en el metamodelo de datos, se guarda una instancia de un `ClassItemView` (que muestra la información relativa al modelo de clase de la clase asociación) y de un `AssociationItemView`. Este diseño asegura la coherencia entre la representación gráfica y el modelo de metadatos subyacente.

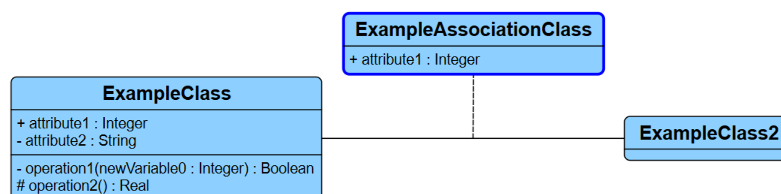

Figura 8.10 Representación visual de una clase de asociación UML.

GeneralizationItemView

La clase `GeneralizationItemView` representa las generalizaciones entre clases (relaciones de herencia). Su método `paint` dibuja la línea que conecta la subclase con la superclase, finalizando en la punta triangular característica de UML, la cual es calculada de una forma parecida a los diamantes de las agregaciones y las composiciones. Implementa también métodos como `updatePosition` ya mencionados en el ítem de la asociación.

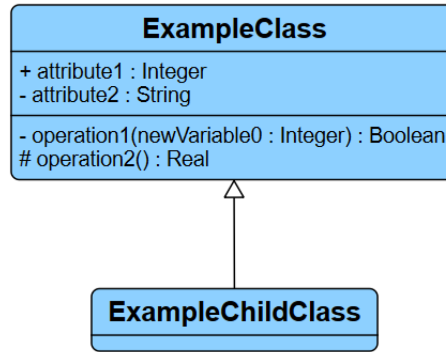

Figura 8.11 Representación visual de una generalización UML.

8.4. Implementación del patrón `Command`

Una característica esencial de la aplicación es la capacidad de deshacer y rehacer acciones realizadas por el usuario. Para lograr este objetivo, se ha implementado el patrón de diseño `Command`, que encapsula cada operación como un objeto independiente, permitiendo su ejecución, deshacer y rehacer de manera controlada. Las acciones reversibles son gestionadas a través de una pila de comandos, donde cada comando conoce cómo deshacer y rehacer su propia acción. Qt ofrece ya implementada una interfaz para facilitar la integración de este patrón, `QUndoCommand`. Esta interfaz define los métodos `undo()` y `redo()`, que deben ser implementados por cada comando específico. Además, Qt proporciona la clase `QUndoStack`, que actúa como una pila para almacenar los comandos ejecutados, facilitando la gestión del historial de acciones. Dicha pila puede ser conectada a la interfaz de usuario, por ejemplo como se ha hecho en este TFG, integrando las acciones ofrecidas por la pila en la barra de herramientas superior, junto a sus respectivos comandos.

En cuanto a las acciones implementadas podemos encontrar: añadir, eliminar y modificar clases, asociaciones, clases asociación y enumerados, y mover clases y enumerados.

Cada comando debe almacenar la información necesaria para poder rehacer y deshacer la acción sin depender de otros objetos. Normalmente, almacenarán el modelo de datos del elemento editado del diagrama, el `QGraphicsItem` correspondiente del elemento modificado, la escena gráfica para añadir o eliminar los elementos, y el metamodelo.

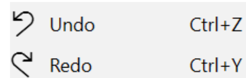

Figura 8.12 Acciones de la pila en la barra superior.

A continuación, se muestra un ejemplo de la definición del comando para añadir, editar y eliminar una clase. En `AddMetaClassCommand`, el puntero de la clase se añade al modelo en el método `redo` y se elimina del modelo en el `undo`. En `EditMetaClassCommand` se guarda la referencia antigua (antes de editarla) y la nueva (clase editada), y se va actualizando el valor del puntero contenido en el modelo, dependiendo del método. Y en `RemoveMetaClassCommand`, se almacena el puntero con la clase eliminada, para poder rehacer y deshacer la acción.

```
        class AddMetaClassCommand : public QUndoCommand{
public:
    AddMetaClassCommand(std::shared_ptr<MetaModel::MetaModel> model,
     ↪  std::shared_ptr<MetaModel::MetaClass> newClass, ClassItemView* newClassView,
     ↪  ModelGraphicsScene* scene);

    void undo() override;
    void redo() override;

private:
    std::shared_ptr<MetaModel::MetaModel> model;
    std::shared_ptr<MetaModel::MetaClass> newClass;
    ClassItemView* newClassView;
    ModelGraphicsScene* scene;
};
```

```
class EditMetaClassCommand : public QUndoCommand{
public:
    EditMetaClassCommand(std::shared_ptr<MetaModel::MetaClass> modelElement,
     ↪  std::shared_ptr<MetaModel::MetaClass> newElement, ClassItemView* classView,
     ↪  ModelGraphicsScene* scene);

    void undo() override;
    void redo() override;
```

```cpp

private:
    ModelGraphicsScene* scene;
    ClassItemView* classView;
    std::shared_ptr<MetaModel::MetaClass> modelMetaElement;
    std::shared_ptr<MetaModel::MetaClass> oldMetaElement;
    std::shared_ptr<MetaModel::MetaClass> newMetaElement;
};
```

```cpp
    class RemoveMetaClassCommand : public QUndoCommand{
public:
    RemoveMetaClassCommand(ClassItemView* classItemView, ModelGraphicsScene* scene,
     ↪  std::shared_ptr<MetaModel::MetaModel> model);

    void undo() override;
    void redo() override;

private:
    ClassItemView* classItemView;
    ModelGraphicsScene* scene;
    std::shared_ptr<MetaModel::MetaModel> model;

    std::map<std::string, std::shared_ptr<MetaModel::MetaAssociation>>
     ↪  associationsRemoved;
};
```

8.5. Implementación de un portapapeles gráfico

La integración de un portapapeles en este tipo de aplicaciones es una característica prácticamente esencial que permite a los usuarios copiar, cortar y pegar elementos dentro de la escena gráfica de manera intuitiva. Esta funcionalidad mejora significativamente la experiencia de usuario al facilitar la manipulación de los elementos del diagrama UML.

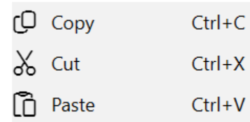

Figura 8.13 Acciones de portapapeles en la barra superior.

Para la implementación de este, se ha creado la clase `ItemViewClipboard`, que actúa como un contenedor temporal para los elementos gráficos seleccionados. Esta clase almacena referencias a los `QGraphicsItem` copiados o cortados, así como a sus modelos de metadatos asociados, garantizando que al pegar los elementos se mantenga la coherencia entre la representación visual y el modelo subyacente. El elemento cortado o copiado perderá todas sus asociaciones. Además, la clase `ModelGraphicsScene` se ha extendido para incluir un puntero al portapapeles, permitiendo que las operaciones de copiar, cortar y pegar se gestionen directamente desde la escena gráfica. Los métodos `setClipboard` y `setClipboardModel` permiten inicializar el portapapeles con la referencia al modelo conceptual, asegurando que los elementos pegados mantengan coherencia con la estructura de datos de alto nivel.

```
1   class ItemViewClipboard : public QObject{
2       Q_OBJECT
3
4   public:
5
6       ItemViewClipboard(ModelGraphicsScene* scene, std::shared_ptr<MetaModel::MetaModel>
        ↪  model);
7
8       void copy(BoxItemView* item);
9       void cut(BoxItemView* item);
10      void paste();
11
12      void setModel(std::shared_ptr<MetaModel::MetaModel> model);
13
14  private:
15      BoxItemView* itemView;
16      ModelGraphicsScene* scene;
17      std::shared_ptr<MetaModel::MetaModel> model;
18
19      std::string getClassCopyName(std::string name);
```

```
    std::string getEnumCopyName(std::string name);
};
```

8.6. Gestión de Temas y Estilos Visuales

Para la alternancia entre los temas disponibles en la aplicación, se ha implementado una funcionalidad que permite cambiar dinámicamente entre un tema claro y un tema oscuro. Esta característica mejora la experiencia del usuario al ofrecer opciones de visualización adaptadas a sus preferencias y condiciones de iluminación. Para ello se ha implementado la clase `ThemeManager`, que se encarga de gestionar los diferentes temas y aplicar los estilos correspondientes a los elementos de la interfaz de usuario. Para aplicar dichos estilos se han usado las hojas de estilo de Qt (QSS), que permiten definir estilos personalizados para los widgets de la aplicación y el uso de paletas de colores. La intención de dicha implementación era usar únicamente las hojas de estilo, pero debido a limitaciones de estas (no pueden gestionar todos los aspectos visuales de algunos widgets), se ha tenido que usar también paletas de colores para controlar los colores de ciertos elementos.

```
class ThemeManager{
private:
    static bool theme;
    static QPalette darkPalette;
    static QPalette lightPalette;

public:
    static void toogleTheme();
    static bool getTheme();
    static void getInitialTheme();

    const static QPalette& getLightPalette();
    const static QPalette& getDarkPalette();

    static int getSectionBackgroundColor();
    static int getWidgetBackgroundColor();
    static int getTextColor();
    static int getAssociationColor();
```

```cpp
19
20     static void createPalettes(const QPalette& systemPalette);
21 };
```

Esta clase ofrece los métodos necesarios para gestionar los temas y estilos visuales de la aplicación que no se pueden gestionar ni con las hojas de estilo como con las paletas de colores, especialmente para el renderizado de los elementos de la escena gráfica.

8.7. Guardado y carga de la disposición de la escena

Para mejorar la experiencia del usuario y permitirle retomar su trabajo en cualquier momento, se ha implementado una funcionalidad que permite guardar y cargar la disposición de los elementos en la escena gráfica. Esta característica es especialmente útil en aplicaciones de modelado, donde los usuarios pueden invertir tiempo significativo en organizar y posicionar los elementos del diagrama. La implementación de esta funcionalidad se ha realizado mediante el uso de archivos `.json`, que ofrecen una estructura ligera y fácil de manipular para almacenar la información necesaria sobre la disposición de los elementos. Además, se ha utilizado el formato `.clt` utilizado por la herramienta USE para guardar y cargar dicha disposición. Aunque esta última solo se ha implementado para la carga de la disposición, ya que USE guarda información adicional de la que no se dispone en este proyecto.

8.8. Pruebas Unitarias del Metamodelo

Las pruebas unitarias constituyen una de las prácticas fundamentales dentro del proceso de desarrollo de software, ya que permiten verificar de manera temprana la corrección de los componentes más pequeños e independientes de la aplicación [29]. En el caso particular del presente proyecto, se ha considerado de especial importancia garantizar la robustez del *metamodelo*, dado que este constituye la base conceptual y estructural sobre la cual se construyen y validan los modelos gráficos que la aplicación gestiona.

El metamodelo, al definir los elementos esenciales como las *metaclases*, *metaasociaciones* y *metatipos*, debe asegurar propiedades de consistencia y coherencia que, de no cumplirse, comprometerían la validez del modelo conceptual en su conjunto. Por ello, se diseñaron y

ejecutaron una serie de pruebas unitarias enfocadas en verificar tanto la creación como la manipulación de dichos elementos.

8.8.1. Objetivos de las pruebas

Los principales objetivos de las pruebas unitarias realizadas al metamodelo fueron los siguientes:

- Comprobar la correcta instanciación de metaclases, verificando que atributos y operaciones se inicialicen de forma adecuada.
- Validar la creación de metaasociaciones entre metaclases, asegurando que se respete la direccionalidad y multiplicidad definidas.
- Garantizar la correcta inserción, modificación y eliminación de elementos en el metamodelo.
- Verificar que el mapa interno de elementos del metamodelo (`modelItemViewElementsMap`) mantenga la integridad de claves y referencias.
- Asegurar que las operaciones de sincronización con la capa de representación gráfica no generen inconsistencias.

8.8.2. Metodología aplicada

Para la implementación de las pruebas unitarias se empleó el marco de trabajo `Qt Test` [30], que proporciona utilidades para la verificación de clases y métodos en proyectos desarrollados con C++ y Qt. Este marco permitió definir casos de prueba de forma estructurada, incluyendo `fixtures` de inicialización y limpieza de datos antes y después de cada ejecución. Además se ha utilizado la estructura AAA (*Arrange, Act, Assert*) para organizar cada prueba de manera clara y comprensible.

Cada prueba fue concebida bajo la premisa de aislar la funcionalidad a validar, de manera que se redujera al mínimo la interferencia con otros módulos de la aplicación.

Ejemplo de prueba unitaria para la clase `MetaAttribute`:

```
void MetaAttributeTest::metaAttribute_setType_notVoidType_updatesType(){
    // Arrange
```

```
    auto stringType = MetaModel::String::instance();
    auto newType = MetaModel::Integer::instance();

    // Act
    metaAttribute->setType(newType);

    // Assert
    QVERIFY(metaAttribute->getType().equals(*newType));
    QVERIFY(!metaAttribute->getType().equals(*stringType));
}
```

8.8.3. Resultados obtenidos

Las pruebas unitarias han permitido identificar algunos fallos relacionados con la gestión de claves en el mapa de elementos del metamodelo y con la propagación de cambios en las asociaciones. Una vez corregidos estos errores, el metamodelo alcanzó un grado de estabilidad suficiente que permitió continuar con la implementación de las capas superiores de la aplicación.

En conclusión, la aplicación de pruebas unitarias sobre el metamodelo se reveló como un proceso clave para garantizar la solidez de la base conceptual y para minimizar futuros errores durante la manipulación de los modelos gráficos y la generación de código.

Capítulo 9

Conclusión

Finalmente se presentan las conclusiones obtenidas a lo largo del desarrollo del proyecto, así como una reflexión sobre el proceso de diseño e implementación. También se discutirán las lecciones aprendidas y las posibles direcciones futuras para el trabajo en este ámbito.

9.1. Experiencia

Este proyecto ha supuesto un desafío significativo que ha permitido aplicar y consolidar conocimientos en diversas áreas, incluyendo el diseño de interfaces gráficas, la gestión de modelos de datos y la implementación de patrones de diseño. La experiencia adquirida en el uso de Qt y C++ ha sido especialmente valiosa, ya que ha implicado la resolución de problemas complejos relacionados con la interacción del usuario, la representación visual de datos y la gestión eficiente de memoria. También es cierto que el aprendizaje de Qt y de C++ ha requerido un esfuerzo considerable debido a su extensa documentación, la variedad de componentes disponibles, y la necesidad de comprender conceptos de menos alto nivel, como puede ser la gestión de memoria con punteros, variables por referencia, etc.

También hay algunos pequeños problemas que escapan de nuestro alcance, debido a cómo está hecho el framework de Qt, como puede ser el tema de las hojas de estilo, que no permiten modificar ciertos aspectos visuales de los elementos de la interfaz gráfica, y el renderizado de la escena no actualiza bien los elementos pintados (por ejemplo, manteniendo elementos

pintados en su posición original al moverse, no pintar las asociaciones o generalizaciones...). Pero esto pasa con poca frecuencia.

En resumen, este proyecto ha sido una experiencia enriquecedora que ha permitido el crecimiento profesional y personal de los involucrados, así como la creación de una herramienta útil y versátil para la gestión de modelos.

9.2. Líneas futuras

Esta aplicación puede llegar a tener un potencial muy grande para contribuir al mundo del software al intentar integrar funcionalidades que no estan presentes en herramientas de código abierto y mejorando la usabilidad de estas. Pero a pesar de los logros alcanzados, siempre hay margen para la mejora. La aplicación podría tener infinitos caminos para seguir su implementación en el futuro, dependiendo de lo que se quiera buscar mejorar. Algunas líneas futuras para su desarrollo, que yo considero importantes a tener en cuenta, podrían ser:

- Ampliar el soporte para cargar más tipos y formatos de diagramas y relaciones entre elementos.
- Implementar una interfaz que sea más reactiva, y reducir el numero de ventanas adicionales.
- Implementar un editor de texto para observar los codigos generados dentro de la aplicación.
- Añadir una sección de configuración, para poder personalizar la apariencia y el comportamiento de la herramienta.
- Integrar otro mecanismo para guardar el modelo en formato *Alloy*, otra herramienta de validación de modelos.
- Mejorar la interfaz de usuario para hacerla más accesible.
- Transladar el proyecto a otro lenguaje con una curva de aprendizaje más baja, como puede llegar a ser Kotlin [31] y su framework Compose Multiplatform, que nos ayudaría a crear una interfaz de aspecto mas moderna e intentar reutilizar código de USE.
- Realizar pruebas más exhaustivas para identificar y corregir posibles errores o com-

portamientos no deseados.

- Añadir un mecanismo de validación de modelos para asegurar que los diagramas creados cumplen con las reglas y restricciones del meta-modelo UML y sus restricciones OCL.
- Implementar la visualización de multiplicidades y roles en las asociaciones. No se ha podido aplicar debbido a laa complejidad para manejar numerosos elementos gráficos en la escena.
- Integrar la selección de visibilidad en los extremos de las asociaciones, para su posterior generación de código.
- Mostrar las relaciones reflexivas en la escena gráfica. Para esta funcionalidad no se ha conseguido encontrar una solución eficiente que permitiese controlar la posición de los elementos en la escena, por lo que se ha optado por mostrarlas en el cuadro de dialogo al editar la clase.

Bibliografía

[1] M. Fowler y K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 1999.

[2] R. S. Pressman et al., «A practitioner's approach,» *Software Engineering*, vol. 2, pág. 60, 2010.

[3] J. Lorés et al., *Interacción Persona-Ordenador*, J. Lorés, ed., págs. 72-74. dirección: `https://aipo.es/wp-content/uploads/2022/02/LibroAIPO.pdf`

[4] D. Lending y N. L. Chervany, «The use of CASE tools,» 1998.

[5] O. (M. Group), *UML*, Online. dirección: `https://www%20.omg.org/spec/UML/`

[6] *Ejemplos de diagramas de clase 2*. dirección: `https://www.webyempresas.com/ejemplos-de-diagramas-de-clases-uml/`

[7] M. Richters y M. Gogolla, «OCL: Syntax, Semantics, and Tools,» en *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, T. Clark y J. Warmer, eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, págs. 42-68, ISBN: 978-3-540-45669-8. DOI: `10.1007/3-540-45669-4_4` dirección: `https://doi.org/10.1007/3-540-45669-4_4`

[8] G. Ramos Jiménez, *Apuntes de teoría de autómatas y lenguajes formales/Gonzalo Ramos Jiménez*, spa. Málaga: Universidad de Málaga, 2003, ISBN: 8460776948.

[9] M. Gogolla, F. Büttner y M. Richters, «USE: A UML-based specification environment for validating UML and OCL,» *Science of Computer Programming*, vol. 69, n.º 1, págs. 27-34, 2007, Special issue on Experimental Software and Toolkits, ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2007.01.013` dirección: `https://www.sciencedirect.com/science/article/pii/S0167642307001608`

[10] J. Lorés et al., *Interacción Persona-Ordenador*, J. Lorés, ed., págs. 59-66. dirección: `https://aipo.es/wp-content/uploads/2022/02/LibroAIPO.pdf`

[11] F. C., «La evaluación heurística,» *Prototypr*, 2018. dirección: https://blog.prototypr.io/evaluacion-heuristica-9c8fce655759

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

[13] Wikipedia, *Enterprise architect*. dirección: https://en.wikipedia.org/wiki/Enterprise_Architect_%28software%29

[14] VisualParadigm, *Visual Paradigm*. dirección: https://www.visual-paradigm.com/

[15] VisualParadigm, *Visual Paradigm guide*. dirección: https://guides.visual-paradigm.com/unlock-your-creative-potential-with-visual-paradigm-community-edition-your-free-uml-tool/

[16] StarUML, *StarUML*. dirección: http://staruml.io/

[17] StarUML, *StarUML*. dirección: https://staruml.io/

[18] PlantUML, *PlantUML*. dirección: https://plantuml.com/

[19] B. Stroustrup, «An overview of C++,» págs. 7-18, 1986. DOI: 10.1145/323779.323736 dirección: https://dl.acm.org/doi/pdf/10.1145/323779.323736

[20] Qt, *Qt framework*. dirección: https://www.qt.io/product/framework

[21] Qt, *Qt Creator Documentation*. dirección: https://doc.qt.io/qtcreator/

[22] Figma, *Figma*. dirección: https://www.figma.com/es-es/

[23] T. J. Parr y R. W. Quong, «ANTLR: A predicated-LL(k) parser generator,» *Software: Practice and Experience*, vol. 25, n.º 7, págs. 789-810, 1995. DOI: https://doi.org/10.1002/spe.4380250705 eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705. dirección: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705

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

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

[26] M. T. Gallego, *Metodología Scrum*. dirección: https://openaccess.uoc.edu/server/api/core/bitstreams/64de036d-c56c-4317-ad60-fcb968469221/content

[27] Qt, *User Interface Compiler*. dirección: https://doc.qt.io/qt-6/uic.html

[28] Qt, *Signals and Slots*. dirección: https://doc.qt.io/qt-6/signalsandslots.html

[29] M. Ellims, J. Bridges y D. C. Ince, «The economics of unit testing,» *Empirical Software Engineering*, vol. 11, n.º 1, págs. 5-31, 2006.

[30] Qt, *Qt Test Framework*. dirección: `https://doc.qt.io/qt-6/qtest-overview.html`

[31] S. Samuel y S. Bocutiu, *Programming kotlin*. Packt Publishing Ltd, 2017.

Apéndice A

Guía de instalación

Este apéndice proporciona instrucciones detalladas para la instalación y configuración del proyecto, tanto si se desea compilar desde el código fuente como si se prefiere instalar la aplicación directamente.

A.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.

A.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`.

Apéndice B

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.

B.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 B.1 Ventana principal de la aplicación.

B.2. Funciones principales

B.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:

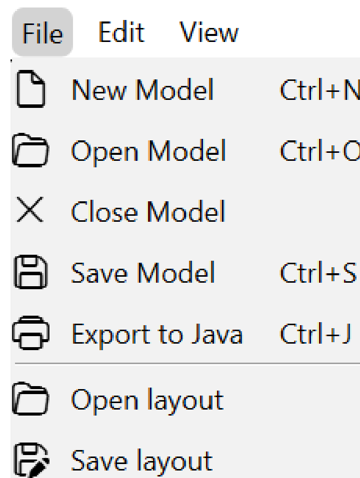

Figura B.2 Menú de gestión de archivos.

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

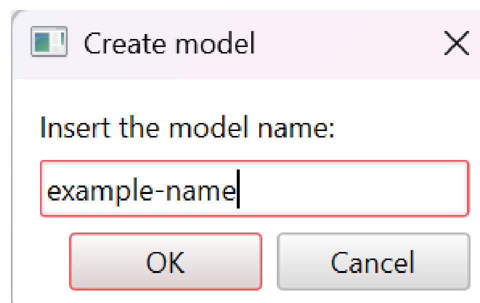

Figura B.3 Creación de un nuevo modelo.

- **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`.

B.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 B.4). Todos los elementos comparten que, para ser válidos, su nombre debe ser único.

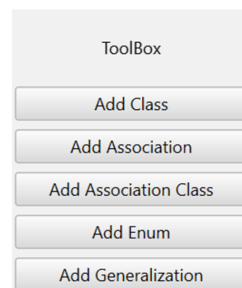

Figura B.4 Caja de herramientas de edición del modelo.

- **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 B.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 area central de trabajo.

Figura B.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 B.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.

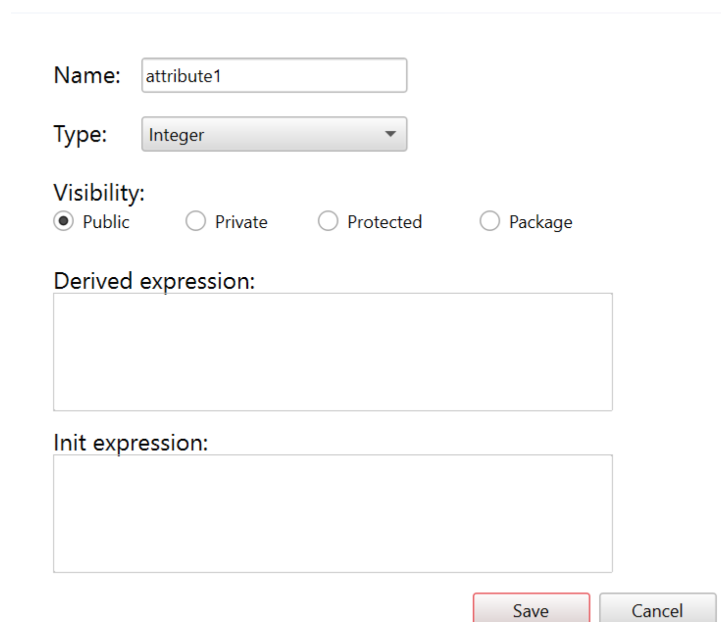

Figura B.6 Creación de un nuevo atributo.

- **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 B.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 boton *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 se podrá rellenar un campo de texto con la expresión y establecer su tipo).

Figura B.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 B.8). En dicha ventana podremos editar el nombre del enumerado y añadir sus literales. Una vez creado el enumerado, se añade al area central de trabajo. También se pueden editar los enumerados haciendo doble click sobre ellos en el area de trabajo.

Figura B.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 B.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 despues 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 módelo, 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 B.10).

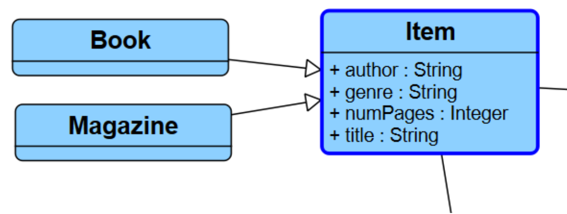

Figura B.10 Elemento seleccionado con el borde azulado.

B.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.

B.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.

B.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.

B.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.

B.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.

B.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, respectívamente.
- `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