Transformación de modelos de la vista estructural de un sistema

Unai Gastón Osés

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Directora del trabajo: María Antonia Zapata Abad
11 de junio de 2023

Índice general

Resumen

En este Trabajo de Fin de Grado exploramos las transformaciones de modelos, un componente clave de la Ingeniería de Software basada en modelos. Las transformaciones de modelos son necesarias por diferentes motivos. En primer lugar, la adaptación a cambios en los requisitos del sistema es un motivo importante para realizar transformaciones. A medida que evoluciona un proyecto, es común que surjan nuevos requisitos o que los existentes se modifiquen. Mediante las transformaciones de modelos, podemos ajustar los modelos existentes para reflejar estos cambios, garantizando así la coherencia entre el modelo y los requisitos del sistema. Otro motivo para realizar transformaciones en los modelos es la mejora de la calidad del modelo. Los modelos pueden contener elementos redundantes o ineficientes que pueden afectar la comprensión, mantenibilidad y eficacia del sistema. Mediante las transformaciones de modelos, podemos aplicar técnicas para eliminar redundancias, simplificar estructuras complejas y mejorar la legibilidad y mantenibilidad del modelo.

En nuestro enfoque, proponemos tratar los modelos como grafos y las transformaciones de modelos como transformaciones de grafos, lo que nos permite aplicar teorías y técnicas de la teoría de grafos. Comenzamos por introducir los conceptos fundamentales de modelo y metamodelo, sentando así las bases para comprender las transformaciones de modelos. A continuación, exploramos diferentes tipos de transformaciones de modelos y posteriormente presentamos diversas definiciones de grafos y de transformaciones de grafos, las cuales nos sirven de base teórica para estudiar las transformaciones de modelos. En particular, nos centramos en dos tipos de transformaciones: endógenas y exógenas.

En el ámbito de las transformaciones endógenas, nos centramos en la refactorización de modelos y demostramos cómo la teoría de transformación de grafos brinda soporte formal para esta actividad. Presentamos un ejemplo específico de un modelo de red de área local (LAN) y describimos el proceso de refactorización utilizando un ejemplo concreto llamado *Ascender Método*. Además, presentamos una secuencia de reglas adicionales que nos permitirán llevar a cabo la transformación completa.

En cuanto a las transformaciones exógenas, nos enfocamos en la transformación del diagrama de clases al esquema relacional, una transformación ampliamente explorada en la literatura. Establecemos un conjunto de reglas con un orden concreto que nos permiten llevar a cabo la transformación al esquema relacional. Como ejemplo particular, aplicamos estas reglas a un diagrama de clases que modeliza los profesores y estudiantes de los centros de una universidad.

Para llevar a cabo estas transformaciones hemos utilizado el software AGG. A lo largo de este trabajo, se presentan varias opciones que ofrece AGG, así como sus dos modos de ejecución, que nos permiten realizar y visualizar las transformaciones de modelos de manera más cómoda y visual.

Abstract

In this Final Degree Project we explore model transformations, a key component of model-driven software engineering. Model transformations are necessary for several reasons. First, adapting to changes in system requirements is an important reason to perform transformations. As a project evolves, it is common for new requirements to emerge or existing ones to change. Through model transformations, we can adjust existing models to reflect these changes, thus ensuring consistency between the model and the system requirements. Another reason for performing model transformations is to improve model quality. Models may contain redundant or inefficient elements that can affect system comprehensibility, maintainability and efficiency. Through model transformations, we can apply techniques to eliminate redundancies, simplify complex structures and improve model readability and maintainability.

In our approach, we propose to treat models as graphs and model transformations as graph transformations, which allows us to apply theories and techniques from graph theory. We begin by introducing the fundamental concepts of model and metamodel, thus laying the foundation for understanding model transformations. We then explore different types of model transformations and subsequently present various definitions of graphs and graph transformations, which serve as a theoretical basis for studying model transformations. In particular, we focus on two types of transformations: endogenous and exogenous.

In the area of endogenous transformations, we focus on model refactoring and demonstrate how graph transformation theory provides formal support for this activity. We present a specific example of a local area network (LAN) model and describe the refactoring process using a concrete example called *Pull Up Method*. In addition, we present a sequence of additional rules that will allow us to carry out the complete transformation.

As for exogenous transformations, we focus on the transformation from class diagram to relational schema, a transformation widely explored in the literature. We establish a set of rules with a concrete order that allow us to perform the transformation to the relational schema. As a particular example, we apply these rules to a class diagram modeling the teachers and students of the centers of a university.

To carry out these transformations we have used the AGG software. Throughout this work, several options offered by AGG are presented, as well as its two execution modes, which allow us to perform and visualize model transformations in a more comfortable and visual way.

Capítulo 1

Transformaciones de modelos

En el ámbito de la Ingeniería del Software, se han desarrollado varias propuestas que se agrupan bajo el término de Ingeniería Dirigida por Modelos (Model Driven Engineering - MDE), las cuales promueven el desarrollo de sistemas informáticos utilizando modelos y metamodelos. Esta aproximación plantea que las modificaciones necesarias en el sistema se capturan a través de transformaciones de modelos. En este contexto, comenzaremos introduciendo los conceptos de modelo y metamodelo. Luego, nos adentraremos en una explicación más detallada de diversos aspectos relacionados con la transformación de modelos. Las referencias que hemos utilizado principalmente para desarrollar este capítulo han sido [2], [3], [5], [6], [8], [10] y [11].

1.1. Modelo y metamodelo

Un **modelo** se define como una representación simplificada y abstracta de un sistema. Su objetivo principal es capturar los aspectos esenciales y relevantes del sistema en cuestión, al tiempo que simplifica y omite detalles innecesarios para facilitar su comprensión y análisis. Al utilizarse como una herramienta para comprender y analizar sistemas complejos, es fundamental que el modelo sea más fácil de manejar que el sistema original.

El uso de modelos en el desarrollo de software abarca diversas etapas del ciclo de vida. Al inicio, los modelos de requisitos sirven para identificar y definir las funcionalidades del sistema, actuando como una representación simplificada de las necesidades del cliente. En la etapa de diseño, los modelos de arquitectura y estructura permiten visualizar las interacciones entre los componentes y evaluar diferentes alternativas arquitectónicas. Por otro lado, los modelos de comportamiento capturan el flujo de información y las interacciones del sistema, facilitando la comprensión de su comportamiento en distintas situaciones y permitiendo simular y validar antes de llevar a cabo la implementación del sistema.

Es importante comprender que no existe una línea clara que delimite los diferentes conceptos y construcciones utilizados en la modelización de sistemas. Sin embargo, por conveniencia, se suelen dividir en varias perspectivas o vistas [12]. Una **vista** es simplemente un subconjunto de construcciones de modelización que representa un aspecto específico de un sistema. En este sentido, en la modelización de sistemas se suelen considerar dos vistas principales:: la vista estructural y la vista dinámica.

La **vista estructural** se enfoca en describir los elementos del sistema y las relaciones entre sí. Utiliza clasificadores como clases, casos de uso, componentes y nodos. Los clasificadores son los elementos a partir de los que posteriormente se podrán representar el comportamiento dinámico del sistema. En esta vista se pueden utilizar diagramas tales como el diagrama de clases, el diagrama de casos de uso o el diagrama de componentes.

Para lograr una modelización precisa, es necesario que en la vista estructural se definan los conceptos

clave del sistema, sus propiedades internas y sus relaciones mutuas. Uno de los diagramas más utilizados para representar la parte estructural del sistema es el diagrama de clases. En un diagrama de clases los conceptos de la aplicación se modelizan como clases. Se utilizan atributos para modelizar la información y operaciones para modelizar el comportamiento. Además, varias clases pueden compartir una estructura común mediante la generalización, donde una subclase añade estructura y comportamiento adicional a la estructura y comportamiento heredado de la superclase común.

En la vista estructural también se representan las relaciones entre objetos, como las dependencias, las cuales agrupan relaciones como cambios en los niveles de abstracción, concesión de autorización y uso entre elementos.

La figura 1.1. muestra un ejemplo de diagrama de clases usando el Lenguaje Unificado de Modelado (UML) [12]. Este modelo representa la estructura de una red de área local (LAN), donde se pueden observar vértices que conforman la LAN, los cuales pueden ser de tres tipos distintos: estación de trabajo, servidor de archivos y servidor de impresión. La clase *Nodo* (superclase) representa la generalización de estos tres tipos de vértices (subclases). También se ha representado el hecho de que los nodos pueden enviarse paquetes.

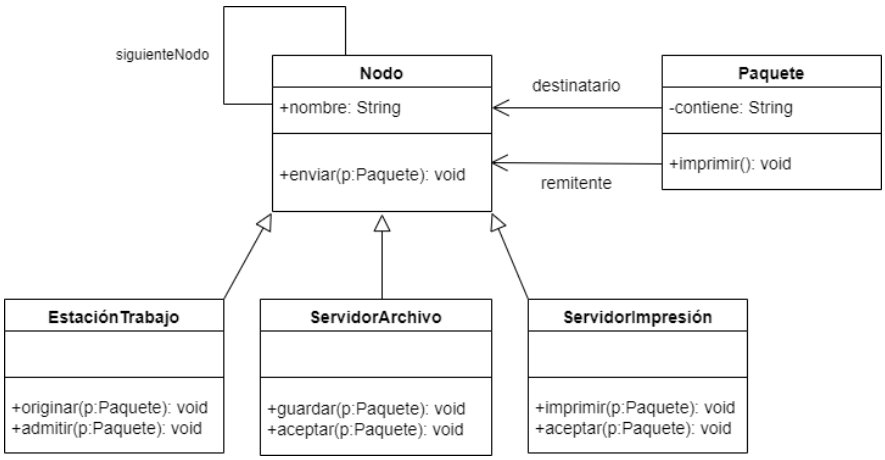

Figura 1.1: Diagrama de clases UML que modeliza la estructura de una LAN.

La **vista dinámica** describe el comportamiento de un sistema a lo largo del tiempo. El comportamiento se puede describir como una serie de cambios en instantáneas del sistema obtenidas de la vista estructural. Las vistas de comportamiento dinámico incluyen, por ejemplo, la vista de máquina de estados, la vista de actividades y la vista de interacción.

El comportamiento dinámico se puede modelizar de dos formas. Una es la historia de vida de un objeto a medida que interactúa con el resto de elementos. La otra es el patrón de comunicación de un conjunto de objetos conectados mientras interactúan con el objetivo de implementar un comportamiento.

Es importante destacar que en este trabajo solamente nos centramos en el estudio de las transformaciones de modelos que corresponden a la vista estructural de los sistemas.

Un **metamodelo** se define como una especificación explícita de una abstracción o simplificación utilizada para modelizar otros modelos. En otras palabras, un metamodelo es un modelo que describe cómo deben ser estructurados y construidos otros modelos dentro de un dominio específico.

Puesto que un metamodelo también es un modelo, surge el problema de que a su vez, de forma recurrente, necesitamos un meta-metamodelo para representarlo. En el contexto del MDE, la solución que se propone es una arquitectura de tres niveles de modo que, en el primer nivel, M1, estarían los modelos, en un segundo nivel, M2, estarían los metamodelos y en un tercer nivel, M3, estaría un meta-metamodelo que se define en términos de sí mismo.

El propósito principal de un metamodelo es proporcionar una representación formal y precisa de las reglas, restricciones y relaciones que gobiernan la construcción y el uso de modelos en un dominio particular (ver figura 1.2). Un metamodelo define los elementos esenciales que componen un modelo, cómo se relacionan entre sí y qué propiedades y comportamientos pueden tener. Al establecer estas reglas, el metamodelo garantiza la consistencia y la corrección de los modelos creados dentro del dominio específico.

Un sistema no tiene asociado un único modelo, sino que diferentes modelos de un mismo sistema pueden representar aspectos de diferentes vistas. Puesto que estos modelos estén extraídos del mismo sistema, seguirán manteniendo una relación entre sí.

Por ejemplo, supongamos que tenemos un modelo que define los elementos de una red local (LAN). También tenemos otro modelo que define los roles de acceso y permisos en la red. El modelo LAN posee estaciones de trabajo, servidores de archivos y servidores de impresión. Luego podemos asignar los roles de acceso y permisos correspondientes a cada elemento de la red. Aunque son conceptos diferentes, podemos relacionar los elementos de la red con los roles de acceso y permisos.

Un metamodelo sirve como base para realizar transformaciones automáticas de modelos. Al definir la estructura y las relaciones de los elementos del modelo, el metamodelo proporciona las pautas necesarias para realizar operaciones de transformación. Estas transformaciones automatizadas permiten mejorar la productividad y mantener la consistencia entre los diferentes modelos.

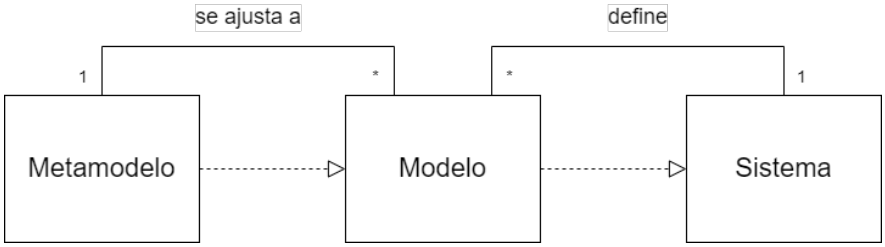

Figura 1.2: Relación entre modelo, metamodelo y sistema.

Para finalizar este capítulo, introducimos el concepto de trasformación de modelos, así como algunas de las razones que motivan la necesidad de hacer transformaciones.

Una **transformación de modelos** es el proceso de generar automáticamente un modelo objetivo a partir de uno o varios modelos fuente, siguiendo una definición de transformación. Esta definición consiste en un conjunto de reglas de transformación que describen cómo los elementos en los modelos fuente pueden ser convertidos en elementos en el modelo objetivo, que puede ser único o múltiple. Cada regla de transformación especifica cómo se realiza la transformación de uno o más elementos del modelo fuente al modelo objetivo (ver figura 1.3).

Las transformaciones de modelos son ampliamente utilizadas en desarrollo de software por varias razones. Estas incluyen la reutilización y mantenimiento de modelos, donde las transformaciones permiten adaptar modelos existentes a diferentes contextos o plataformas. También se utilizan para realizar análisis y verificaciones en modelos, lo que ayuda a detectar errores y mejorar la calidad del sistema. Por último, las transformaciones son útiles para integrar sistemas y datos al permitir la transformación de modelos entre diferentes lenguajes y tecnologías, facilitando la interoperabilidad entre ellos.

1.2. Tipos de transformaciones de modelos

En esta sección vamos explorar los diferentes tipos de transformaciones de modelos. Uno de los enfoques utilizados para clasificar estas transformaciones se basa en la distinción entre transformaciones exógenas y endógenas.

Figura 1.3: Conceptos básicos de la transformación de modelos.

Las **transformaciones exógenas** se refieren a procesos de conversión entre modelos que están expresados haciendo uso de distinto metamodelo. Estas transformaciones permiten traducir la información y estructura de un modelo en un metamodelo de origen a un modelo equivalente en un metamodelo de destino.

Destacamos algunos ejemplos comunes de transformaciones exógenas:

- *Síntesis* de una especificación de nivel superior, más abstracta, en un modelo de nivel inferior, más concreto. Por ejemplo, se puede traducir un modelo de análisis o diseño en un modelo de programa Java. La *ingeniería inversa*, por otro lado, extrae una especificación de nivel superior a partir de una de nivel inferior.
- *Migración* de un programa escrito en un lenguaje a otro, manteniendo el mismo nivel de abstracción, es decir, sin cambiar el nivel de detalle o la representación del sistema.

Las **transformaciones endógenas** son transformaciones entre modelos expresados en el mismo metamodelo. Estas transformaciones implican la manipulación y modificación de un modelo existente utilizando las mismas representaciones y estructuras del metamodelo de origen.

A continuación, veamos algunos ejemplos de transformaciones endógenas:

- *Optimización*, una transformación que busca mejorar ciertas cualidades operativas del modelo, como el rendimiento, sin alterar su semántica.
- *Refactorización*, transformación que implica cambios en la estructura interna del modelo para mejorar sus características sin cambiar su comportamiento.
- *Simplificación* y *normalización*, que se utilizan para reducir la complejidad sintáctica del modelo mediante la traducción de construcciones sintácticas avanzadas en construcciones más primitivas del lenguaje.
- *Adaptación de componentes*, que implica la modificación y adaptación del código de componentes de modelo existentes, ya sea de manera estática o dinámica, para satisfacer las necesidades del usuario.

Es importante mencionar que las transformaciones endógenas pueden clasificarse aún más en función del número de modelos involucrados. En algunos casos, los cambios se realizan en el mismo modelo (*in-place*), mientras que en otros se crean elementos de modelo en base a propiedades de otro modelo (*out-place*). Cabe señalar que las transformaciones exógenas siempre son *out-place*.

A continuación, vamos a seguir explorando la clasificación de las transformaciones en dos categorías: transformaciones horizontales y transformaciones verticales.

Las **transformaciones horizontales** son aquellas que se centran en el mismo nivel de abstracción del modelo. Estas transformaciones se enfocan en cambios y mejoras dentro del mismo dominio y nivel de detalle. Ejemplos de transformación horizontal son la refactorización y la migración.

Por otro lado, las **transformaciones verticales** se refieren a cambios que ocurren entre diferentes niveles de abstracción del modelo. Estas transformaciones implican la creación de nuevos modelos o la modificación de modelos existentes a un nivel de abstracción más alto o más bajo. Ejemplos típicos de transformación vertical son la simplificación y la generación de código.

	horizontal	vertical
endógena	*Refactorización*	*Simplificación*
exógena	*Migración de lenguaje*	*Generación de código*

Figura 1.4: Dimensiones ortogonales de las transformaciones de modelos con ejemplos.

Para finalizar con la clasificación de transformaciones, podemos distinguir entre estas tres categorías: Model to Text (M2T), Model to Model (M2M) y Text to Model (T2M).

La transformación **Model to Text (M2T)** es una transformación que genera texto legible por humanos a partir de un modelo. En este enfoque, el modelo de entrada se procesa y se extraen elementos relevantes que se utilizan para generar el texto final. El objetivo principal de M2T es producir documentación, código fuente u otros tipos de textos a partir de modelos.

Model to Model (M2M) se refiere a la transformación de un modelo de origen en un modelo de destino. En este tipo de transformación, el modelo de entrada se analiza y se generan uno o más modelos de salida que representan información específica o en un nivel de abstracción diferente. El propósito de M2M es permitir la interoperabilidad entre diferentes modelos y facilitar la reutilización de modelos.

Por último, **Text to Model (T2M)** es una transformación que captura información estructurada a partir de texto no estructurado. En este tipo de transformación, el texto de entrada se procesa y se extraen elementos significativos que se utilizan para construir un modelo. El objetivo de T2M es permitir la representación formal y la manipulación de información contenida en documentos de texto. El concepto de transformación de T2M implica el proceso inverso a la transformación M2T, es decir, la creación de un modelo a partir de texto o código fuente existente.

Adicionalmente, exploraremos el concepto de espacio técnico.

Un **Espacio técnico** es un marco de gestión de modelos que contiene conceptos, herramientas, mecanismos, técnicas, lenguajes y formalismos asociados a una tecnología específica. En otras palabras, representa el entorno en el que se desarrolla una determinada tecnología.

El término Espacio Técnico se utiliza con la intención de denotar tecnologías a un nivel más abstracto para permitir el razonamiento sobre sus similitudes, diferencias y posibilidades de integración. A menudo, está asociado a una comunidad de usuarios con conocimientos compartidos, apoyo educativo, literatura común e incluso talleres y conferencias. Es tanto una zona de experiencia establecida como una investigación en curso, así como un repositorio de recursos abstractos y concretos.

Cuando se realiza una transformación de modelos, los modelos de origen y destino pueden pertenecer al mismo espacio técnico o a espacios técnicos diferentes. En el último caso, se requieren herramientas y técnicas para definir transformaciones que conecten los espacios técnicos. Los espacios técnicos pueden clasificarse según su nivel de abstracción. Si se puede expresar más semántica en un determinado espacio

técnico, se dice que está en un nivel más alto de abstracción.

Por último, se presentan dos ejemplos de espacios técnicos: MDA, que puede considerarse un espacio de alto nivel (abstracto), y XML, que puede considerarse un espacio de nivel más bajo (concreto).

Model Driven Architecture (MDA) es un enfoque propuesto por *Object Management Group* (OMG). Según MDA, el proceso de desarrollo de software se compone de varios modelos diferentes, cada uno representando una vista particular del sistema que se está construyendo. Los modelos se escriben en el lenguaje de su metamodelo. En el contexto de MDA, existen varios metamodelos estándar, siendo el más popular el metamodelo UML. En línea con lo que hemos comentado previamente, los modelos se encuentran en el nivel M1, y la definición de los lenguajes (metamodelos) utilizados para crear estos modelos se encuentra en el nivel M2. Por último, también existe un lenguaje especial utilizado para definir los metamodelos, llamado *Meta-Object Facility* (MOF), que se califica como un metametamodelo. El MOF se define a sí mismo y constituye el nivel M3 (ver figura 1.5).

Un objetivo particular de MDA es separar los modelos que son descripciones neutrales del sistema (PIM o Modelos Independientes de Plataforma) de los modelos que están vinculados a una tecnología específica (PSM o Modelos Específicos de Plataforma). El objetivo es definir procesos estándar para obtener PSMs a partir de PIMs, idealmente mediante algún tipo de generación automática.

Extensible Markup Language (XML) es un marco para la definición de lenguajes de marcado estandarizados por el W3C. Se acepta ampliamente como un estándar para la representación e intercambio de datos estructurados y semiestructurados. El documento XML es el concepto central en el espacio técnico de XML. Los documentos se escriben en una sintaxis limitada por restricciones que permiten determinar si un documento está bien formado y es válido. Las restricciones para determinar si un documento está bien formado están definidas por las reglas de gramática XML, mientras que las restricciones de validez se definen en un documento separado llamado esquema de documento, que se escribe en un lenguaje de esquema dado (Definición de Tipo de Documento (DTD), lenguaje de esquema XML, etc.) (ver figura 1.5).

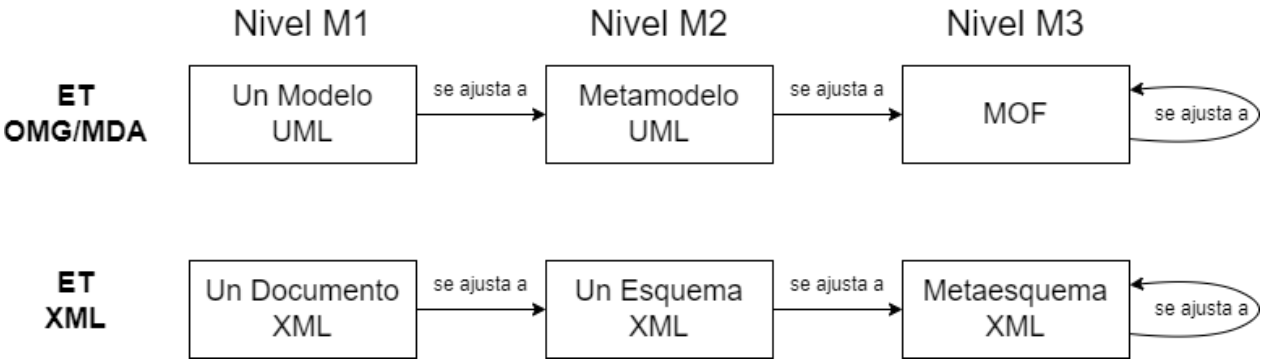

Figura 1.5: La organización del modelo en tres niveles en varios espacios técnicos. Distinguimos tres niveles denominados M3 (nivel de metametamodelo), M2 (nivel de metamodelo) y M1 (nivel de modelo).

A lo largo de este trabajo, nos enfocaremos en las transformaciones M2M dentro del espacio técnico de las transformaciones de grafos. En los próximos apartados, exploraremos en detalle este espacio técnico.

1.3. Los modelos son grafos

Existe un acuerdo generalizado de que los modelos se pueden representar de forma natural por medio de una estructura basada en grafos. Tal como hemos visto previamente, además de los modelos también es necesario definir un metamodelo que especifique lo que significa ser un modelo bien formado. Para recoger este hecho, tal como veremos en este apartado, se utilizan grafos tipo para representar metamodelos y

grafos tipados para representar modelos. De este modo, un modelo estará bien definido si su representación mediante un grafo está tipado de acuerdo al grafo tipo (ver figura 1.6).

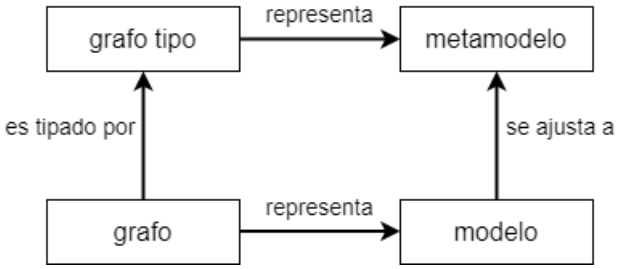

Figura 1.6: Relación entre un modelo y su representación por medio de un grafo.

A continuación, se presentarán las definiciones formales de los conceptos de grafo y grafo tipo, acompañados de ejemplos ilustrativos. Específicamente, en lo que respecta a la representación de modelos mediante grafos, se hace hincapié en la relevancia de los grafos dirigidos.

Definición 1: Grafo dirigido y etiquetado

Un grafo dirigido y etiquetado $G = (V_G, A_G, s_G, t_G, l_G)$ *está formado por un conjunto de vertices* V_G *y un conjunto de arcos* A_G *tal que* $V_G \cap A_G = \emptyset$*. Por otro lado, las funciones* $s_G : A_G \longrightarrow V_G$ *y* $t_G : A_G \longrightarrow V_G$*, asocian a cada arco vértices de origen y destino. Finalmente, la función de etiquetado* $l_G : V_G \cup A_G \longrightarrow \mathscr{L}$*, asigna una etiqueta a cada vértice y arco.*

Una extensión directa y útil de la definición anterior sería agregar información adicional a los vértices y arcos mediante atributos. En este caso, hablamos de grafos con atributos. Cada vértice o arco puede contener cero o más atributos. Un atributo suele ser un par de nombre-valor que permite asociar un valor específico a cada nombre de atributo. Estos valores pueden ser muy simples, como números o cadenas de texto, o más complejos, como expresiones en Java.

En el siguiente ejemplo se ilustrará cómo se puede representar un modelo mediante un grafo. Previamente, es necesario mencionar que para realizar los ejemplos en este trabajo se ha utilizado la herramienta de transformación de grafos AGG[1] . AGG es una herramienta implementada en Java que permite trabajar con grafos y agregar información adicional utilizando elementos de Java. En particular, es importante destacar que los grafos implementados en AGG son grafos con atributos, lo que significa que cada vértice y arista puede tener atributos asociados, los cuales pueden ser expresados utilizando el lenguaje Java.

La figura 1.7 nos muestra cómo se puede representar, usando la herramienta AGG, el diagrama de clases de la figura 1.1 por medio de un grafo. Más adelante, cuando introduzcamos el concepto de grafo tipo, veremos el significado de los distintos tipos de nodos y arcos que aparecen en este grafo. También veremos como esta representación permite aplicar técnicas de transformación de grafos para transformar y modificar la estructura de la red de manera más eficiente y sencilla.

Una vez visto el ejemplo de grafo en la figura 1.7, es necesario mencionar dos conceptos fundamentales que debemos explicar para asegurarnos de que nuestro grafo represente correctamente el modelo: el morfismo de grafos y el grafo tipado.

Para que un grafo pueda servir como representación de un modelo, es necesario asegurarse de que el grafo esté bien formado y sea coherente. Para ello, se especifica un conjunto de tipos válidos para los vértices y las arcos. Además, se asegura que las conexiones entre vértices y arcos sean consistentes con esos tipos, lo cual

[1] `https://www.user.tu-berlin.de/o.runge/AGG/WWW/down_V21_java8/index.html`

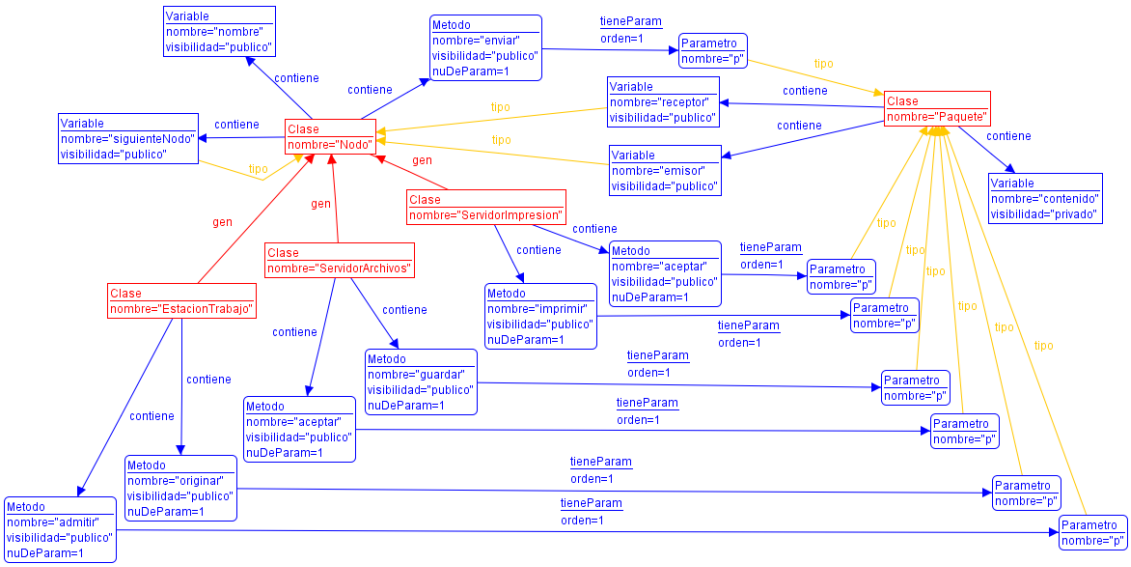

Figura 1.7: Grafo que representa el diagrama de clases UML de la figura 1.1

se logra mediante el uso de morfismos de grafos y grafos tipados.

Definición 2: Morfismo de grafos

Sean G y H grafos dirigidos y etiquetados. Un morfismo (parcial) de grafos $m : G \longrightarrow H$ consiste en dos funciones parciales $m_V : V_G \longrightarrow V_H$ y $m_A : A_G \longrightarrow A_H$ las cuales preservan orígenes y destinos de arcos, es decir, $s_H \circ m_A = m_V \circ s_G$ y $t_H \circ m_A = m_V \circ t_G$. Además, estas dos funciones también preservan las etiquetas de vértices y arcos, es decir, $l_H \circ m_V = l_G$ y $l_H \circ m_A = l_G$.

Un morfismo (parcial) de grafos $m : G \longrightarrow H$ es inyectivo (sobreyectivo) si ambas m_V y m_A son inyectivas (sobreyectivas). Es isomorfo si m es inyectivo y sobreyectivo. En tal caso escribimos $G \cong H$.

Nota: Las funciones m_V y m_A deben ser parciales para permitir la eliminación de vértices y arcos. Todos los vértices en $V_G \setminus dom(m_V)$ y todos los arcos en $A_G \setminus dom(m_A)$ se consideran eliminados por m.

Definición 3: Grafo tipado

Sea TG un grafo dirigido y etiquetado (denominado grafo tipo). Un grafo tipado (sobre TG) es un par (G,t) tal que G es un grafo dirigido y etiquetado y $t : G \longrightarrow TG$ es un morfismo total de grafos. Un morfismo (parcial) de grafos tipados $(G,t_G) \longrightarrow (H,t_H)$ es un morfismo (parcial) de grafos $m : G \longrightarrow H$ el cual también preserva el tipado, es decir, $t_H \circ m = t_G$.

Nota: La definición anterior de grafo tipado requiere un morfismo total de grafos $t : G \longrightarrow TG$ para garantizar que cada vértice y arco del grafo G tiene un tipo correspondiente en TG.

Los conceptos de grafo tipo y grafo tipado se han implementado en la herramienta AGG de modo que, en primer lugar se especifica el grafo tipo y posteriormente se pueden especificar grafos haciendo uso de los tipos de nodos y vértices especificados en el grafo tipo. En concreto, la herramienta AGG, con objeto de establecer restricciones en los grafos tipados, permite definir los grafos tipo haciendo uso de características adicionales, entre las que destacamos las dos siguientes:

- El grafo tipo puede ser atribuido para limitar los nombres y tipos de atributos de vértices y aristas en

grafos tipados concretos.

- El grafo tipo puede contener cardinalidades en nodos y aristas para establecer un límite inferior y superior en el número de vértices y aristas de un cierto tipo permitido en grafos tipados concretos.

En concreto, vamos a especificar el grafo tipo que representa un metamodelo simplificado de los diagramas de clases de UML. Es decir, se establece un conjunto de tipos válidos para los vértices y los arcos y, solo aquellos grafos que cumplan con esta especificación, se considerarán como grafos bien formados.

Este grafo tipo se muestra en la figura 1.8, donde cada vértice y arco tiene una etiqueta que muestra el tipo de elemento que representa. De esta manera, se garantiza que cualquier grafo que se ajuste a esta especificación será coherente y representará correctamente un diagrama de clases.

El grafo tipo expresa las siguientes restricciones sobre grafos concretos:

- **Restricciones entre vértices y arcos:** las clases pueden relacionarse por generalización (*gen* representa la generalización directa y *tgen* su variante transitiva). Las clases contienen Métodos y Variables. Los Métodos se envían Mensajes entre sí y tienen una serie de Parámetros. Los Métodos acceden o actualizan Variables. Las Variables y los Parámetros son tipados por las Clases. El tipo de retorno de un Método es también una Clase.

- **Restricciones de multiplicidad en los arcos:** por ejemplo, cada Variable o Método está contenido exactamente en una Clase. Una Clase contiene cero o más Variables y Métodos. Un Método contiene cero o un tipo de retorno.

- **Restricciones de multiplicidad en los vértices:** en el grafo tipo considerado todos los vértices tienen una multiplicidad de cero o varios valores (*) que no impone ninguna restricción.

- **Restricciones de atributos:** el número de Parámetros de un Método, así como el nombre y la visibilidad de Métodos y Variables, se representa mediante atributos de vértice. El orden de un Parámetro en la declaración de un Método se representa mediante un atributo de arco.

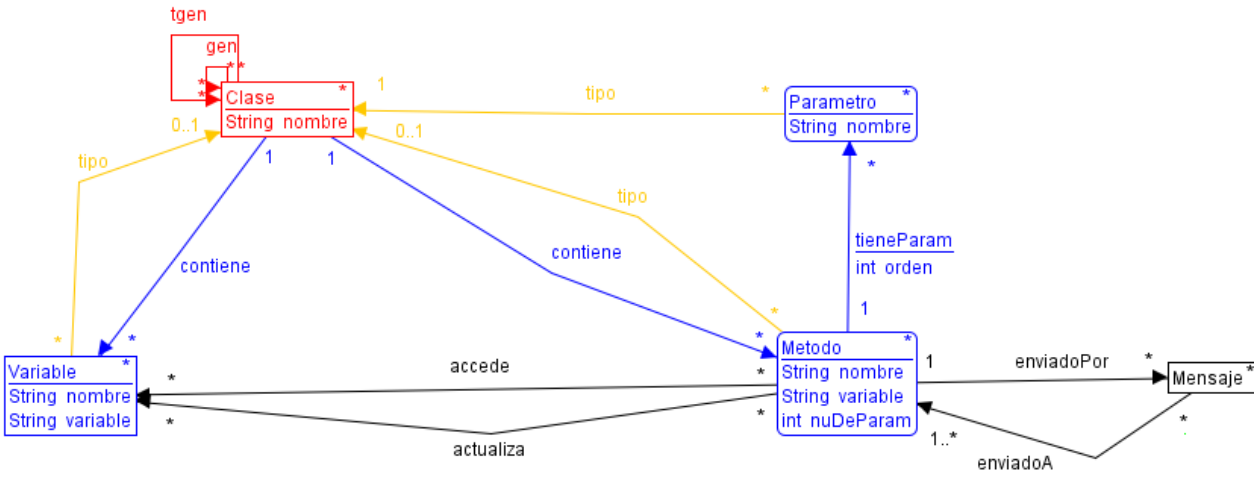

Figura 1.8: Grafo tipo que representa un metamodelo simplificado para diagramas de clases. El grafo de la figura 1.7 es una instancia concreta de este grafo tipo.

Una vez vistas las definiciones de grafos, vamos a introducir la idea de que las transformaciones de modelos pueden ser modelizadas mediante transformaciones de grafos. El objetivo es proporcionar una visión

general de la relación entre las transformaciones de modelos y las transformaciones de grafos, así como una explicación detallada de cómo estas transformaciones pueden ser utilizadas para representar y automatizar transformaciones de modelos.

Las transformaciones de modelos pueden ser definidas mediante reglas de producción y transformaciones de grafos. La definición de regla de producción se basa en la especificación de la estructura de los grafos que deben ser transformados, y las transformaciones de grafos son un mecanismo formal para aplicar reglas de producción a grafos concretos.

Definición 4: Regla de producción y transformación de grafos

Sean L y R dos grafos. Una regla de producción es un morfismo parcial de grafos $p : L \longrightarrow R$*. Una transformación de grafos* $G \Rightarrow_t H$ *es un par* $t = (p,m)$ *formado por una regla de producción* $p : L \longrightarrow R$ *y un morfismo total inyectivo de grafos (llamado coincidencia)* $m : L \longrightarrow G$*.*

Usando un concepto de teoría de categorías llamado pushout, se puede calcular automáticamente los morfismos $m' : R \to H$ *y* $p' : G \to H$ *que hacen que el diagrama* (p,m) *conmute. El grafo H obtenido a través de este proceso es el resultado de aplicar la transformación de grafo t a G.*

De modo informal, la aplicación de una regla de producción $p : L \longrightarrow R$ en el contexto de un grafo concreto G se puede realizar siguiendo los siguientes pasos:

1. Encontrar una coincidencia m del lado izquierdo L de la regla de producción en el grafo G.
2. Crear un grafo de contexto eliminando la parte del grafo G que se asigna a L pero no a R.
3. Unir el grafo de contexto con aquellos vértices y arcos de R que no tienen una contraparte en L.

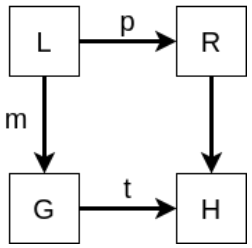

Figura 1.9: Transformación de grafos $G \Rightarrow_t H$ que consiste en una regla de producción $p : L \longrightarrow R$ y una correspondencia $m : L \to G$.

En los sistemas de transformación de grafos con un gran número de reglas de producción, a menudo es necesario restringir la aplicación de las producciones. Para ello, se puede utilizar la noción de Condiciones de Aplicación Negativas (NAC). Esto hace que la transformación de grafos sea considerablemente más expresiva. Intuitivamente, una NAC es un grafo que define una estructura de grafo prohibida (por ejemplo, la ausencia de algunos vértices o arcos). El mecanismo de transformación de grafos se puede ampliar fácilmente para tratar las condiciones de aplicación, comprobando todas las NACs asociadas a la regla de producción en el contexto del grafo de entrada concreto G.

Definición 5: Condición de aplicación negativa

Sea $p : L \longrightarrow R$ *una regla de producción. Una condición de aplicación negativa para p es un morfismo total de grafos* $nac : L \longrightarrow \hat{L}$*. Una transformación de grafos* $G \Rightarrow_{(p,m)} H$ *cumple una condición de aplicación negativa si no hay morfismo de grafos* $\hat{m} : \hat{L} \longrightarrow G$ *tal que* $\hat{m} \circ nac = m$

En la práctica, a una sola regla de producción se le pueden adjuntar varias NAC , es decir, cada regla de producción p tiene asociado un conjunto N de NACs.

Definición 6: Aplicabilidad de una transformación de grafos

Sea $\hat{p} = (p, N)$ una regla de producción $p : L \longrightarrow R$ junto a un conjunto N de condiciones de aplicación negativas. Una transformación de grafos $G \Rightarrow_{(\hat{p},m)} H$ es aplicable si $G \Rightarrow_{(p,m)} H$ satisface cada condición de aplicación negativa de N.

Por último, para ilustrar las definiciones previamente mencionadas, presentamos un ejemplo de transformación de modelos denominada *Renombrar Método*.

La transformación *Renombrar Método* sirve para cambiar el nombre de un método en un modelo. Esta transformación se puede implementar como una regla de producción en AGG, lo que significa que se puede describir formalmente en términos de una transformación de grafos.

Es importante tener en cuenta que cualquier aplicación de la regla de producción *Renombrar Método* debe respetar las condiciones de aplicación negativas (NAC) impuestas. Una de las condiciones (*AusenciaMétodo*) nos indica que no podemos llevar a cabo la transformación si la clase que contiene el método que queremos renombrar, llamado x, ya contiene un método con el nuevo nombre que queremos dar a x, en este caso, y. La otra condición de aplicación negativa (*AusenciaMétodoJerarquía*) nos impide renombrar un método asociado a una clase si su superclase ya posee un método con el mismo nombre que queremos cambiar. De este modo, nos aseguramos de que la transformación no de lugar a una clase con dos métodos con el mismo nombre. Existe una tercera NAC la cual se utiliza para evitar que el método que estamos ascendiendo a la superclase ya exista en alguna de las subclases.

Como podemos observar en la figura 1.10, al lado izquierdo se muestra la NAC *AusenciaMetodo*, luego se muestra el lado izquierdo de la regla de producción, el cual es el grafo inicial y por último se muestra el lado derecho de la regla de producción, resultado de la transformación. Los vértices y aristas que se preservan tienen el mismo número tanto en LHS como en RHS. Las otras dos condiciones de aplicación negativas (NAC) impuestas se observan en la figura 1.11.

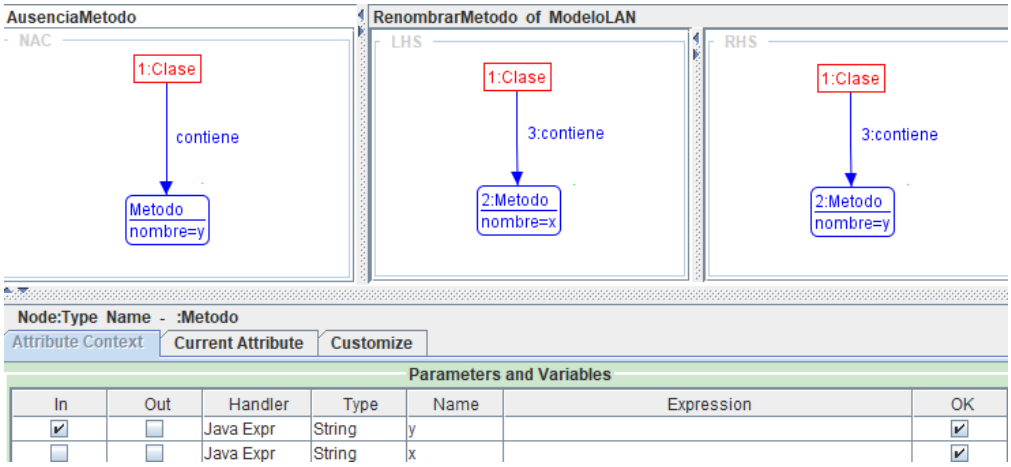

Figura 1.10: Regla de producción *Renombrar Método* con condición de aplicación negativa (AusenciaMétodo) representada en AGG.

Retomando el ejemplo del diagrama de clases del modelo LAN, expresado como un grafo en la figura 1.7, se examina cómo se aplica la regla de producción a la transformación del grafo. En este caso, se pretende renombrar el método *admitir* a *aceptar*. No obstante, no será factible llevar a cabo esta transformación si la

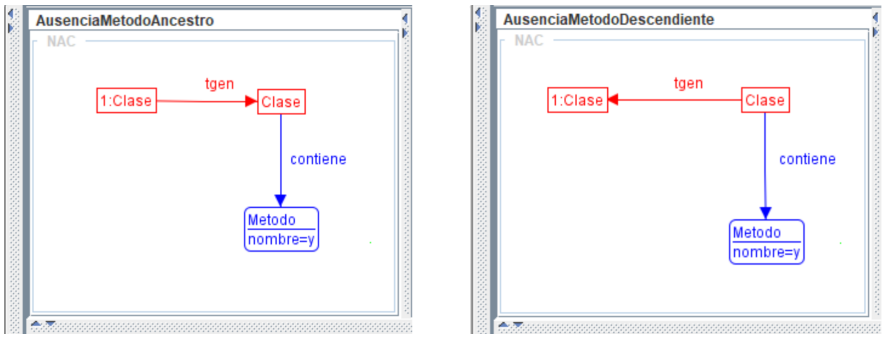

Figura 1.11: Condiciones de aplicación negativas adicionales representadas en AGG.

clase *EstaciónTrabajo* ya tuviera un método con ese nombre, tal como lo indica la condición de aplicación negativa (NAC). Las otras dos NAC no se tienen en cuenta porque previamente habría que haber realizado el cierre transitivo de las relaciones de generalización (*gen*). Las reglas para realizar este cierre transitivo se verán en el siguiente capítulo.

La figura 1.12 muestra el resultado del grafo después de implementar la transformación. Es importante resaltar que la transformación se lleva a cabo utilizando el modo **Step** de AGG. Para ello, es necesario realizar previamente un emparejamiento (*Match*) de la LHS con los elementos del modelo que deseamos transformar y dar valor al atributo y, ya que, tal como se observa en la figura 1.10, es un atributo de entrada (*In*). Observamos que el nombre del método se ha actualizado exitosamente a *aceptar*.

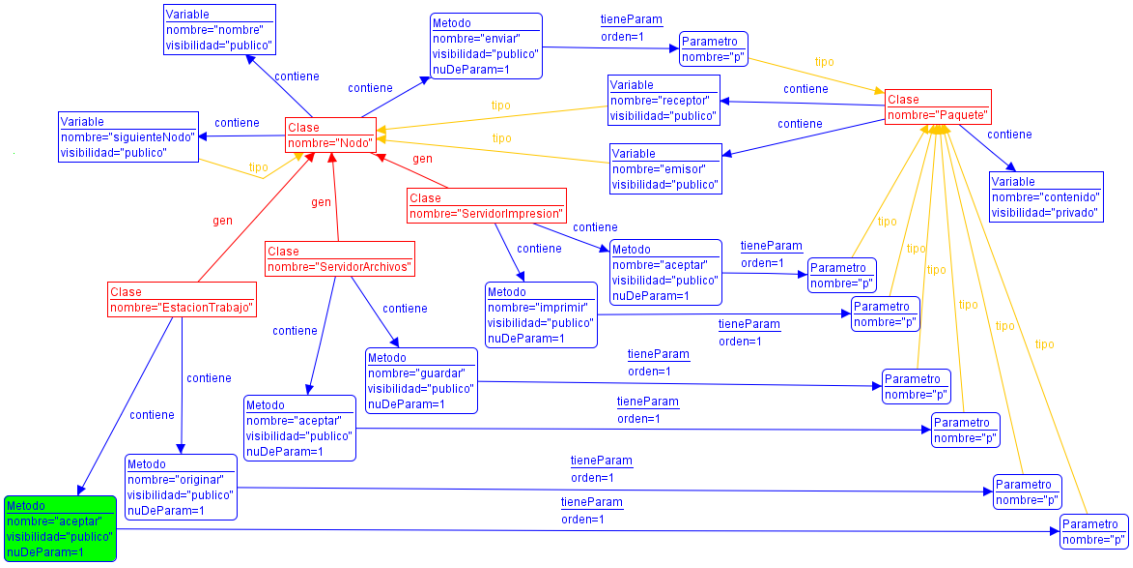

Figura 1.12: Resultado de aplicar la regla de producción *Renombrar Método* al grafo de la figura 1.7.

Por último, queremos comentar que AGG también ofrece la posibilidad de deshabilitar o habilitar las NAC asociadas a una regla de producción. Por ejemplo, en el caso de la transformación *Renombrar Método*, si en una clase queremos sobrecargar un método (dos métodos con el mismo nombre pero con distintos tipos o cantidad de parámetros) entonces deshabilitaríamos la NAC que no permite que una clase tenga dos métodos con el mismo nombre. Del mismo modo, en caso de querer usar polimorfismo (redefinir un método de una clase en un descendiente) entonces deshabilitaríamos las otras dos NAC.

Capítulo 2

Refactorización de modelos: Transformaciones endógenas

En este capítulo, abordamos un ejemplo de un tipo particular de transformaciones endógenas que sirven para la refactorización de modelos. Iniciamos explicando el concepto de refactorización y posteriormente nos adentramos en un ejemplo concreto: la transformación denominada *Ascender Método*. Ilustramos esta transformación en el contexto del modelo LAN y establecemos una estructura de control para garantizar la realización completa de la refactorización. Para respaldar nuestros argumentos, nos basamos en la referencia [11].

La **refactorización** de modelos es un proceso que tiene como objetivo mejorar la calidad de los modelos de software existentes al reorganizar su estructura interna mediante la definición de una transformación específica que se aplicará al modelo. Estas transformaciones son endógenas, lo que significa que se realizan entre modelos expresados en el mismo metamodelo. Por lo general, se utilizan cuando se detectan problemas en el diseño o cuando se necesita mejorar la modularidad o la legibilidad del modelo. Entre las transformaciones más comunes que se utilizan en la refactorización de modelos se incluyen, por ejemplo, la eliminación de elementos duplicados, la simplificación del modelo complejo, la reorganización de jerarquías de clases y la extracción de funcionalidades comunes a una superclase.

A diferencia de la optimización, que se enfoca en mejorar la eficiencia del modelo, la refactorización se enfoca en mejorar la calidad del modelo y hacer que el modelo sea más fácil de entender y mantener. Al aplicar la refactorización es necesario mejorar la calidad del modelo sin introducir errores o problemas en su funcionamiento. Por lo tanto, la refactorización se ha convertido en una práctica común y altamente recomendada para mejorar la calidad y mantenibilidad de los modelos de software existentes.

2.1. Ejemplo de refactorización: Ascender Método

Como ejemplo de refactorización vamos a presentar la transformación **Ascender Método** que posteriormente aplicaremos al modelo LAN. Esta técnica se enfoca en reorganizar la estructura de una jerarquía de clases, moviendo un método de una subclase a una superclase.

Esta técnica es particularmente útil cuando hay una funcionalidad que es común a varias subclases. Al mover la implementación común a la superclase, se evita la duplicación de código.

Al igual que con las reglas de producción, las refactorizaciones también pueden tener condiciones de aplicación negativas. Es importante tener en cuenta que la transformación *Ascender Método* debe aplicarse con cuidado, ya que puede afectar la estructura y la funcionalidad del modelo. Por esta razón, se deben añadir condiciones de aplicación negativas (NAC) que especifiquen cuándo no se permite aplicar la refactorización.

La transformación *Ascender Método* se puede implementar en AGG. El lado izquierdo (LHS) de la regla

de producción se muestra en la parte central de la figura 2.1, mientras que el lado derecho (RHS) se muestra en la parte derecha. Los vértices y aristas que se conservan tienen el mismo número en LHS y RHS. En este ejemplo, se elimina una arista del tipo *contiene* en LHS y se agrega otra arista del mismo tipo en RHS. Todos los demás vértices y aristas se conservan.

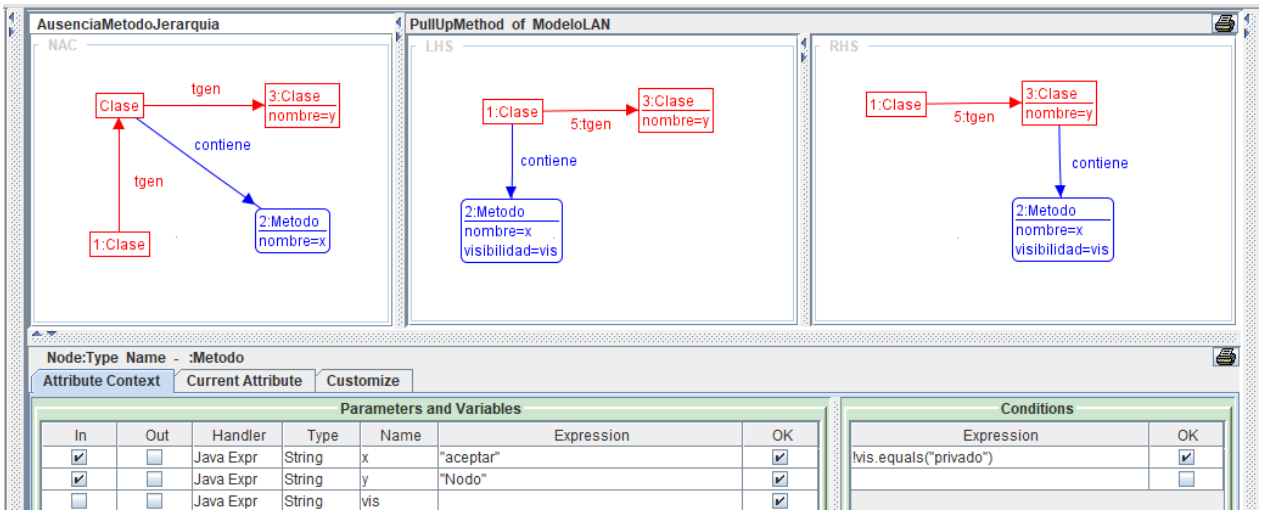

Figura 2.1: Regla de producción de grafos que representa la refactorización *Ascender Método* con condición de aplicación negativa (NAC).

La regla de producción *Ascender Método* se puede hacer más precisa agregando restricciones adicionales que especifiquen cuándo se permite aplicar la refactorización. En la figura 2.1, se muestran dos de estas restricciones. La primera restricción, mostrada en el panel superior izquierdo de la figura 2.1, es una condición de aplicación negativa (NAC) llamada *AusenciaMetodo*. Esta condición especifica que el método no se puede trasladar si un método con el mismo nombre (referido por la variable *x*) ya existe en una clase que es un ancestro de la clase origen en la cadena de herencia entre la clase de origen y la clase de destino del método que se va a trasladar.

La figura 2.1 también muestra una segunda restricción sobre la regla de producción. Esta se puede expresar gráficamente como una NAC pero la expresamos de este modo para visualizar otra de las posibilidades que nos ofrece AGG. Específicamente, queremos expresar que solo se pueden trasladar métodos con una visibilidad no privada. Esto se puede lograr agregando la condición **!vis.equals("privado")**, que se muestra como una expresión Java en el panel inferior derecho.

Esta regla de producción se puede aplicar en el contexto del grafo de la figura 1.7 haciendo coincidir los nodos y las aristas (numeradas del 1 al 5) en el lado izquierdo LHS de la figura 2.1 con los nodos y aristas con números correspondientes en el grafo (mediante la opción *Match* de AGG). Al aplicar esta transformación (mediante la opción *step* de AGG) en el modelo LAN se consigue seleccionar uno de los métodos *aceptar* de las subclases de *Nodo* y mover ese método a la superclase. El resultado de realizar esta transformación se puede observar en la figura 2.2.

En la figura 2.2 se puede observar que, tras la aplicación de la regla *Ascender Método*, podría ser necesario eliminar también los métodos *aceptar* de las clases *EstaciónTrabajo* y *ServidorImpresión*. En el siguiente apartado abordamos la forma de llevar a cabo este proceso.

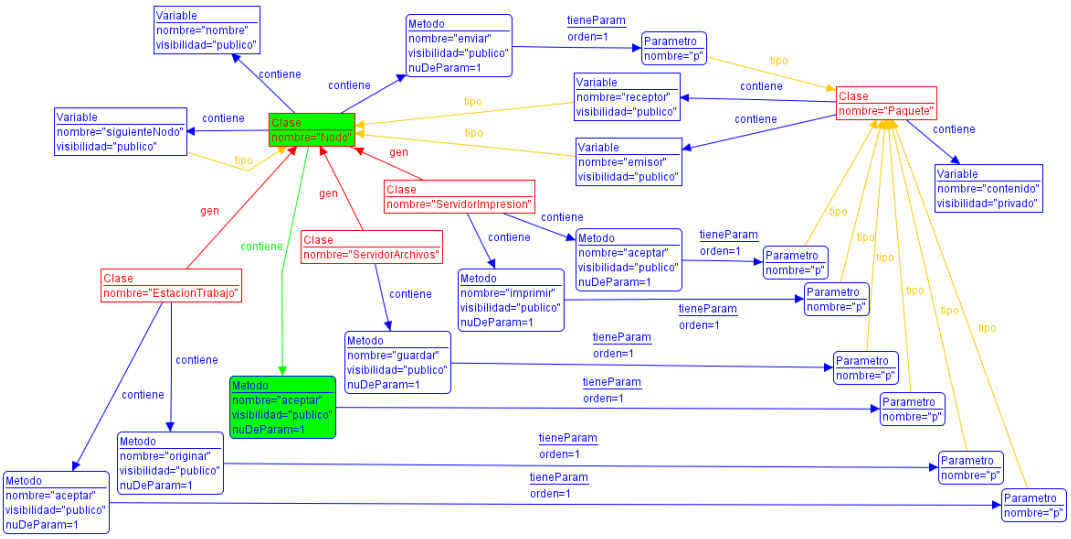

Figura 2.2: Resultado de aplicar la regla de producción *Ascender Método*

2.2. Transformación de grafos programada

Para abordar un sistema de transformación de grafos que contenga un gran número de reglas de producción de grafos, se requieren mecanismos adicionales para reducir la complejidad. Uno de los mecanismos consiste en la transformación de grafos programada. En la **transformación de grafos programada**, es posible combinar un subconjunto de reglas de producción utilizando estructuras de secuencia, ramificación y bucles para controlar su orden de aplicación.

En este sentido, la posibilidad que ofrece la herramienta AGG es definir una *Secuencia de Reglas* en la que se puede indicar el orden de ejecución de un subconjunto de reglas, especificando también las veces que se tienen que ejecutar. Vamos a ver esta posibilidad en relación a la regla de producción *Ascender Método* introducida anteriormente.

Para llevar a cabo la refactorización de un método de forma completa, necesitamos no sólo ascender el método a una superclase, sino también eliminar los métodos que tengan el mismo nombre de las clases descendientes en las que aparezca. Por lo tanto, necesitamos definir otras reglas adicionales y establecer la secuencia en que se deben ejecutar. En concreto, la *Secuencia de Reglas* que hemos definido (ver figura 2.3) consiste primero en calcular el cierre transitivo de las relaciones de generalización, después ascender un método a una superclase y finalmente borrar los métodos de la jerarquía que tengan el nombre del método que hemos ascendido. La ejecución de la secuencia de reglas se ejecuta mediante la opción **Start** de AGG.

En concreto la secuencia de reglas que hemos definido es la siguiente. La secuencia comienza con dos reglas que realizan el cierre transitivo, transformando las relaciones de generalización directas (*gen*) en relaciones de generalización transitivas (*tgen*). A continuación se ejecuta la transformación que asciende un método a una superclase. El método que se quiere ascender, así como la superclase, se indicarían a través de las variables de entrada de la regla. Después se ejecuta la regla *BorrarParámetros*, la cual se ejecuta tantas veces como sea necesario para eliminar todos los parámetros asociados a los métodos que se llaman igual al método ascendido (excepto el método de la superclase). A continuación, se ejecutan las reglas *BorrarEnviadoA* y *BorrarMensaje*, las cuales permiten eliminar el mensaje asociado a los métodos. Cabe destacar que en nuestro ejemplo concreto de modelo no existen mensajes, por lo tanto, estas reglas no se ejecutarán. Las siguientes reglas que se ejecutan son *BorrarAccede* y *BorrarActualiza*, las cuales se encargan de eliminar las asociaciones accede y actualiza. Finalmente, se aplica la regla *BorrarEnDescendente* para eliminar de manera efectiva los métodos con el mismo nombre que aquel que ha sido movido a la superclase.

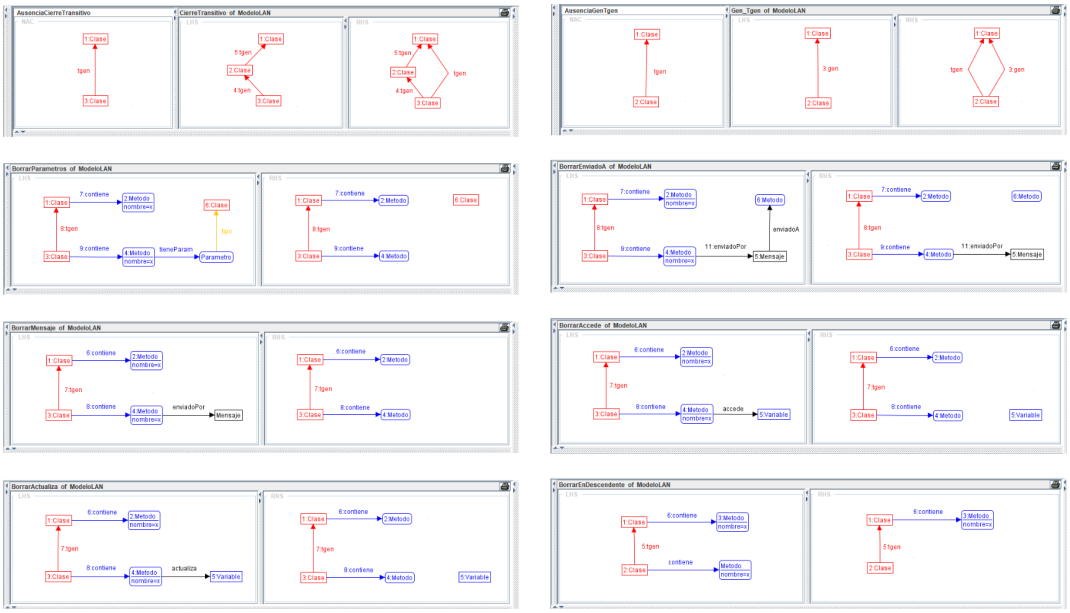

Figura 2.3: Reglas que conforman la *Secuencia de Reglas*.

La figura 2.4 muestra el grafo resultante tras aplicar la secuencia de reglas para ascender el método *aceptar* a la clase *Nodo*, eliminando este método de todas las subclases.

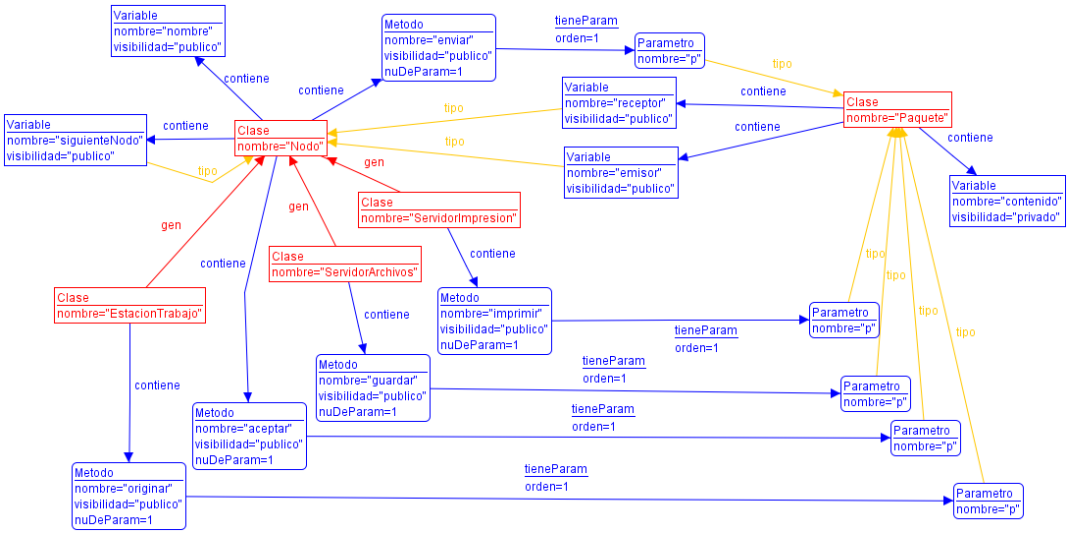

Figura 2.4: Resultado de aplicar la secuencia de reglas al grafo de la figura 1.12

Capítulo 3

Del diagrama de clases al esquema relacional: Transformaciones exógenas

En este capítulo presentaremos un ejemplo de transformación exógena y vertical que es uno de los casos de estudio más habituales en la literatura: la traducción de diagrama de clases en esquemas relacionales. Comenzaremos por explicar los metamodelos que vamos a utilizar y después presentaremos las reglas de transformación específicas de la traducción. En este proceso, emplearemos el modo de ejecución **Start** de AGG para ilustrar cómo se pueden aplicar estas reglas. Además, exploraremos el concepto de integridad referencial. También examinaremos las diferentes posibilidades y enfoques que existen para traducir jerarquías de clases en una base de datos relacional. Para respaldar la exposición de estos conceptos, nos hemos apoyado en las siguientes referencias bibliográficas: [1], [4], [7], [9], [12] y [13]. En particular, los metamodelos y la traducción que vamos a presentar se corresponden con la propuesta que aparece en [7].

3.1. Metamodelos del diagrama de clases y del esquema relacional

El metamodelo que vamos a considerar para el diagrama de clases, al que llamaremos metamodelo **Clase**, es una versión simplificada de los diagramas de clases de UML. Este metamodelo es muy similar al que se ha usado en los capítulos anteriores. La principal diferencia es que, en este caso, como sólo interesa modelizar las estructuras de datos, en el metamodelo no aparece el concepto de método y sus elementos relacionados.

El metamodelo de **Clase** se muestra en la parte izquierda de la figura 3.1. Su metaclase principal es *Clase*, que contiene un conjunto de *Atributos* y tiene una referencia que apunta a una superclase, en caso de que exista, para modelizar árboles de herencia. Un *Atributo* tiene asociado un *Tipo de Dato Primitivo* y puede ser primario. Los atributos primarios se utilizan para identificar objetos. Cabe destacar que, tal como veremos más adelante, las subclases no tendrán atributos primarios adicionales. Por último, la metaclase *Asociación* permite representar relaciones binarias entre objetos y apunta a sus clases origen y destino. Todas las metaclases tienen un nombre.

Para llevar a cabo la traducción al esquema relacional, a los diagramas de clases se les exigen algunas restricciones adicionales. En concreto, cada clase, que no tenga una superclase, tendrá siempre un único atributo primario. Por su parte, las subclases no tendrán atributos primarios, puesto que su identificación se llevará a cabo a través del atributo primario que heredan de la superclase. Por último, sólo consideraremos un nivel de jerarquía de clases y subclases. Como hemos visto en el capítulo anterior, si no se diera esta circunstancia se podría calcular previamente el cierre transitivo de la relación.

La figura 3.2 muestra un ejemplo de diagrama de clases sobre los profesores y estudiantes de los centros de una universidad. Este modelo incluye dos clases, *Persona* y *Centro*, las cuales están conectadas por una asociación llamada *perteneceA*. La clase *Centro* tiene dos atributos, *Nombre* y *Código*, siendo este último

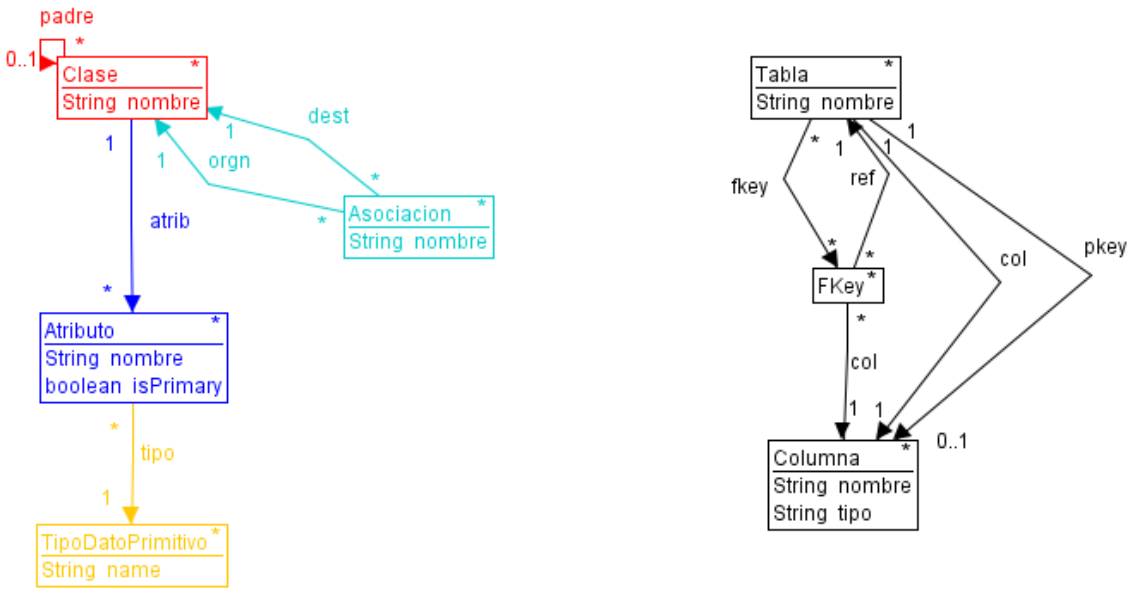

Figura 3.1: Metamodelo de **Clase** y **Tabla**.

el atributo primario. Por otro lado, la clase *Persona* también tiene dos atributos, *Nombre* y *NIP*, siendo *NIP* el atributo primario. Además, esta clase tiene dos subclases, *Estudiante*, con un atributo adicional llamado *Grado*, y *Profesor*, con un atributo adicional llamado *Área*. Como se puede observar este ejemplo cumple las restricciones que hemos indicado.

Con respecto al modelo relacional, indicar que este modelo introdujo la idea de almacenar elementos de la base de datos en tablas bidimensionales. La idea de separar cosas estrechamente relacionadas de cosas más distantes dividiéndolas en tablas fue uno de los principales factores que distinguieron al modelo relacional de los modelos jerárquico y de red. En concreto, una base de datos relacional consta de tablas bidimensionales de filas y columnas, donde la celda en la intersección de una fila y columna contiene un valor no divisible en subvalores. Además, cada tabla puede tener una **clave primaria**, que identifica de manera única cada fila de la tabla. Un esquema relacional representa la estructura de la base de datos relacional y determina los posibles valores que serán almacenados en ella.

En el modelo relacional, la integridad referencial es un principio que garantiza la coherencia y consistencia de los datos. Se refiere a la capacidad de mantener las relaciones entre las tablas de una base de datos de manera precisa y válida. De esta manera, cualquier operación que modifique la base de datos debe cumplir las reglas correspondientes sin necesidad de realizar verificaciones adicionales dentro de la aplicación. En el contexto relacional, la integridad de los datos se gestiona mediante restricciones de clave primaria y clave foránea. Una **clave foránea** es una restricción que establece una relación entre un grupo de origen y un grupo de destino, donde cada instancia del grupo de origen debe corresponder a una instancia del grupo de destino. Además, el grupo de destino debe representar un identificador.

El metamodelo que vamos a considerar para el esquema relacional, al que llamaremos metamodelo **Tabla**, contiene los elementos esenciales del modelo relacional (ver parte derecha de la figura 3.1). En concreto, la metaclase principal de este metamodelo es *Tabla*, que contiene un conjunto de *Columnas* y un conjunto de claves foráneas (*fkey*). La *Tabla* se relaciona con la metaclase *Columna* mediante las referencias *pkey* (clave primaria) y *col* (columnas). Una tabla puede tener como máximo una columna que funcione como clave primaria. Una clave foránea está representada por la *Columna* a la que se refiere con *fcol* (que será una columna de la tabla donde está la clave foránea) y hace referencia (*ref*) a una *Tabla* (realmente a la clave primaria de dicha tabla). Las tablas y columnas tienen nombres y las columnas también tienen asociado

un tipo.

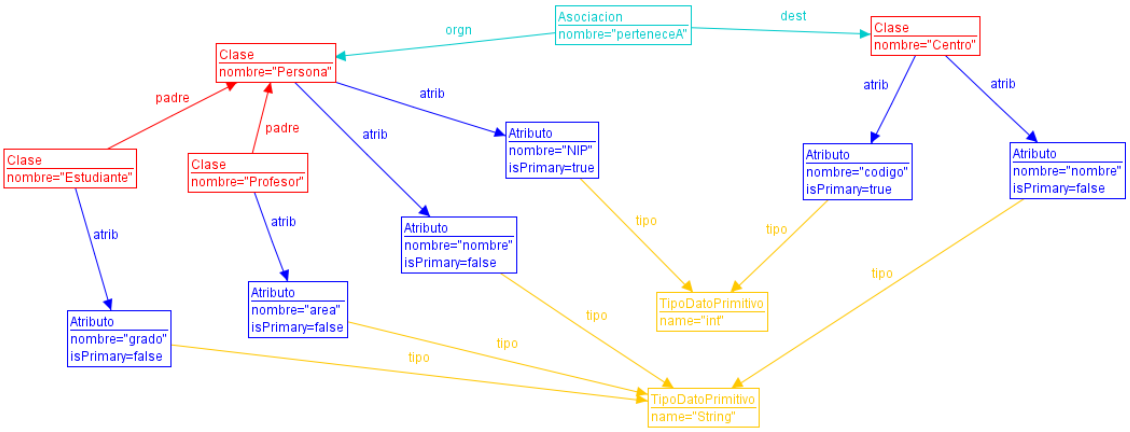

Figura 3.2: Ejemplo de diagrama de clases.

3.2. Traducción del diagrama de clases al esquema relacional

Para la traducción que presentamos en este capítulo utilizaremos la transformación vertical exógena introducida en la referencia [5], también conocida como mapeo objeto-relacional. Durante el proceso de transformación exógena y vertical, nuestro objetivo principal es lograr la conversión del diagrama de clases en un esquema relacional. El diagrama de clases proporciona una visión de alto nivel de las entidades, sus atributos y las relaciones entre ellas. Sin embargo, para implementar el sistema en una base de datos relacional, necesitamos descender a un nivel más detallado y concreto.

Una parte fundamental de la traducción es la selección del método para traducir la jerarquía de clases en una base de datos relacional. La jerarquía de clases es un concepto clave en el diseño orientado a objetos y permite establecer relaciones entre clases, donde una clase puede heredar atributos y comportamientos de otra clase. Al trasladar este concepto a un esquema relacional, es necesario tomar decisiones sobre cómo representar y gestionar esta relación jerárquica entre clases. A continuación, exploraremos tres soluciones fundamentales para implementar la jerarquía de clases en una base de datos relacional:

- **Utilizar una tabla para toda la jerarquía de clases:** se mapea toda la jerarquía de clases en una única tabla, donde se almacenan todos los atributos de todas las clases de la jerarquía. Las ventajas de este enfoque son que es simple ya que se admite el polimorfismo, es decir, la capacidad de los objetos de cambiar su tipo, en las diferentes clases. Además, es fácil asignar identificadores a los objetos, ya que todos se almacenan en una sola tabla. También es muy fácil generar informes, ya que todos los datos necesarios sobre una clase se encuentran en una sola tabla. Las desventajas son que cada vez que se agrega un nuevo atributo en cualquier lugar de la jerarquía de clases, se debe agregar un nuevo atributo a la tabla. Esto aumenta el acoplamiento dentro de la jerarquía de clases, ya que si se comete un error al agregar un solo atributo, podría afectar a todas las clases dentro de la jerarquía y no solo a las subclases de la clase que recibió el nuevo atributo. Además, se desperdicia mucho espacio en la base de datos.

- **Utilizar una tabla por cada clase concreta:** cada tabla incluye tanto los atributos propios como los atributos heredados de la clase que representa. La principal ventaja de este enfoque es que es relativamente fácil generar informes, ya que todos los datos necesarios sobre una clase concreta se almacenan

en una única tabla. También es sencillo asignar identificadores de objetos siempre y cuando el polimorfismo no sea un problema. Sin embargo, existen varias desventajas en este enfoque. En primer lugar, cuando se modifica una clase, es necesario modificar la tabla de cualquiera de sus subclases concretas. En segundo lugar, cada vez que un objeto cambia de rol, se debe copiar los datos en la tabla apropiada y asignarle un nuevo identificador, implicando mucho trabajo. En tercer lugar, es difícil admitir múltiples roles y mantener la integridad de los datos al utilizar este enfoque.

- **Utilizar una tabla por cada clase:** en este enfoque, se crea una tabla por cada clase, donde los atributos incluidos son el identificador y los atributos específicos de esa clase. La principal ventaja de este enfoque es que se ajusta mejor a los conceptos del diseño orientado a objetos. Admite muy bien el polimorfismo, ya que simplemente se tienen registros en las tablas correspondientes para cada rol que pueda tener un objeto. Además, es muy fácil modificar las superclases y agregar nuevas subclases, ya que solo se necesita modificar o agregar una tabla. Sin embargo, existen varias desventajas en este enfoque. En primer lugar, se termina teniendo muchas tablas en la base de datos, una por cada clase. Esto puede aumentar la complejidad y dificultar la administración de la base de datos. En segundo lugar, leer y escribir datos utilizando esta técnica lleva más tiempo, ya que es necesario acceder a múltiples tablas. En tercer lugar, realizar informes en la base de datos puede ser difícil si no se agregan vistas en las tablas de datos para facilitar el proceso.

En nuestro ejemplo, al traducir la jerarquía de clases en una base de datos relacional, se ha optado por utilizar una tabla única que abarque toda la jerarquía de clases. Esta tabla contendrá tanto los atributos de la superclase como los atributos de las subclases. Esta decisión de diseño es adecuada para jerarquías de herencia pequeñas con poca variación entre las clases, mientras que para las jerarquías grandes con una gran variación se podría optar por alguna de las otras dos opciones.

La traducción de modelos de clase a modelos relacionales se describe, de modo informal, de la siguiente manera.

- Para cada instancia de *Clase* que no sea una subclase, se debe crear una instancia de *Tabla* con el mismo nombre. Una jerarquía de clases completa se traduce en una tabla, es decir, una subclase no se asigna a una tabla separada, sino que corresponde a la tabla de su superclase.

- Para cada instancia de *Atributo* con un tipo de dato, se debe crear una instancia de *Columna*. Sus nombres y tipos deben coincidir. Si el Atributo es primario, la *Tabla* hace referencia a esta *Columna* mediante una referencia *pkey*. En cualquier caso, la *Tabla* debe hacer referencia a la *Columna* con una referencia *col*.

- Para cada instancia de *Asociación*, se debe crear una tabla y dos instancias de Clave Foránea (*FKey*). Para ello, la tabla creada contiene dos columnas, con referencias *col*. Además, cada una de estas columnas es una clave foránea (*fkey*) que apunta a la tabla correspondiente a una de las clases relacionadas por la asociación, indicado mediante una referencia *ref*. Esta traducción se puede hacer para traducir cualquier tipo de asociación, incluidas las asociaciones N-N.

Para diseñar en AGG una traducción de diagramas de clases a esquemas relacionales, primero identificamos las correspondencias entre las metaclases. La traducción debe asignar una *Clase* a una *Tabla*, una *Asociación* a una clase y a dos *Clave Foránea* y, por último, un *Atributo* y un *Tipo de Dato* a una *Columna*. Tal como se puede observar en la figura 3.2, para realizar un seguimiento de la traducción, hacemos estas correspondencias explícitas mediante el uso de nodos de correspondencia y sus referencias en los modelos fuente y objetivo. Dependiendo de qué tipos de nodos correspondan entre sí, los nodos de correspondencia están tipificados por *C* (clase-tabla y asociación-tabla) y *A* (atributo-columna y asociación-clave foránea).

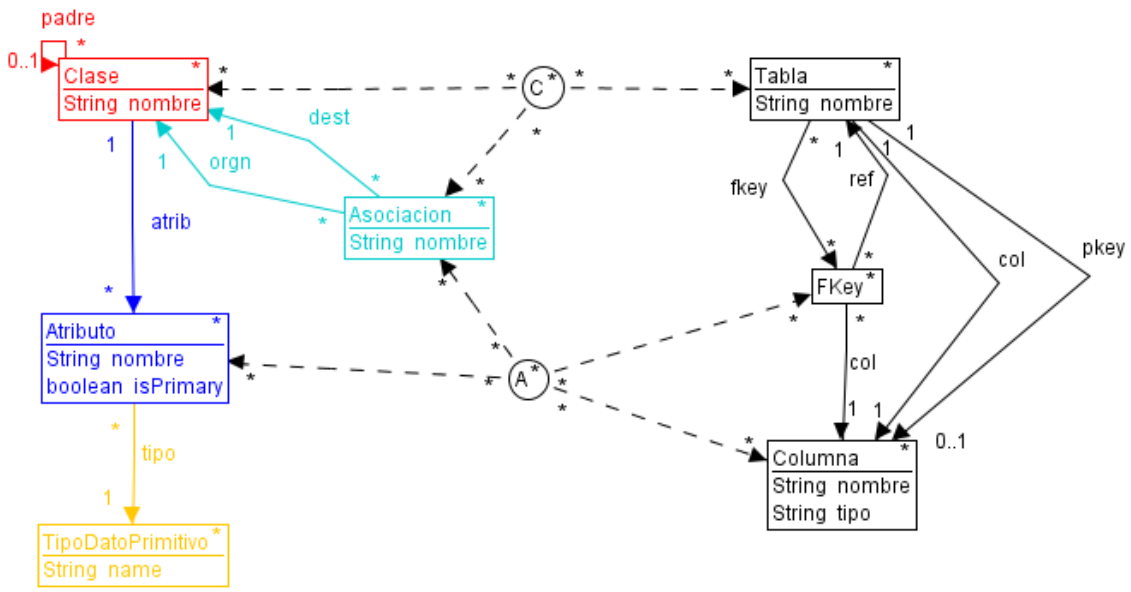

Figura 3.3: Metamodelo de **Clase** y **Tabla** y sus relaciones.

A continuación, describimos e ilustramos las reglas que se van a aplicar al diagrama de clases para que se realice la traducción. Previamente hay que hacer notar que cada elemento del diagrama de clases solo puede ser traducido una vez. Para conseguir esto, por un lado, vamos a llevar registro de los elementos que van siendo traducidos haciendo uso de los nodos de correspondencia. Y, por otro lado, todas las reglas tienen una regla de aplicación negativa que evita que se aplique si el elemento correspondiente ya ha sido traducido. Este tipo de solución sólo es válido cuando cada elemento del modelo origen se traduce en un elemento del modelo destino (y un elemento del destino puede provenir de varios elementos del modelo origen). Por ejemplo, esta solución no sería adecuada para la traducción inversa, es decir, de esquemas relacionales a diagramas de clases, ya que una tabla puede corresponder a más de una clase. En este caso, la asignación inversa no sería unívoca y se produciría una ambigüedad en la correspondencia entre el esquema relacional y los diagramas de clases.

A continuación, describimos e ilustramos las reglas que se van a aplicar al diagrama de clase para que se realice la traducción.

La regla **claseEnTabla** es utilizada para traducir una clase que no tiene superclase en una tabla con el mismo nombre. Cuando se aplica esta regla, se crea una tabla en el esquema relacional con el nombre de la clase. Esta tabla servirá para almacenar la información correspondiente a los objetos de esa clase en el diagrama de clases.

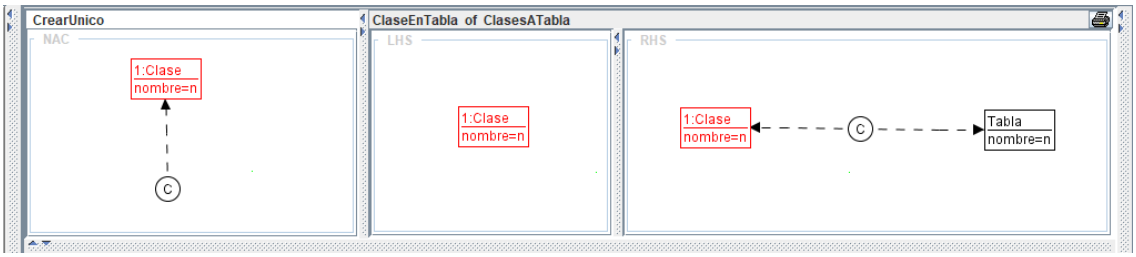

Figura 3.4: Regla **claseEnTabla**.

La regla **atributoEnColumna** se utiliza para traducir un atributo de una clase si la clase contenedora ya

ha sido traducida previamente. Cuando se aplica esta regla, se crea una nueva columna en la tabla correspondiente a la clase traducida, utilizando el mismo nombre que el atributo y asignando un tipo de dato adecuado.

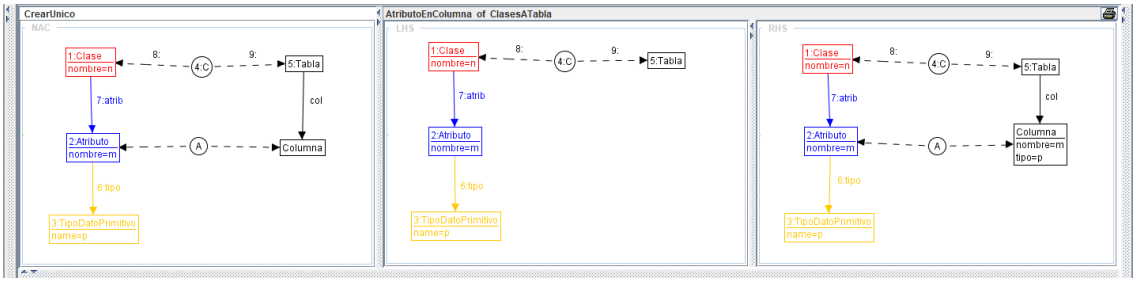

Figura 3.5: Regla **atributoEnColumna**.

La regla **clavePrimaria** se aplica para determinar si una columna en la tabla relacional debe ser designada como clave primaria, basándose en si su atributo correspondiente en el diagrama de clases es considerado como primario.

Cuando se aplica esta regla, se verifica si el atributo de la clase está marcado como primario. Si es así, se establece la columna correspondiente en la tabla como la clave primaria. Esto implica que los valores en dicha columna deben ser únicos y no nulos, lo que garantiza la identificación única de cada registro en la tabla.

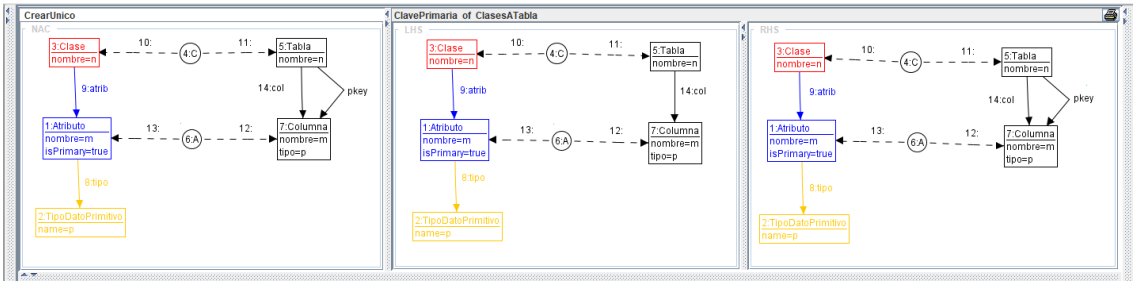

Figura 3.6: Regla **clavePrimaria**.

La regla **subclaseEnTabla** es utilizada para traducir una subclase de una clase que ya ha sido traducida, de manera que se aplana todo el árbol de herencia en una sola tabla en el esquema relacional.

Cuando se aplica esta regla, se toma una subclase y se la asigna a la misma tabla que su superclase. Como hemos explicado anteriormente, este enfoque de aplanar todo el árbol de herencia en una sola tabla simplifica la estructura relacional y evita la necesidad de crear tablas separadas para cada subclase.

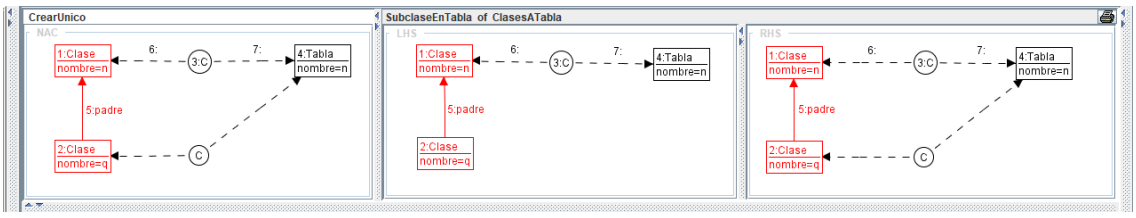

Figura 3.7: Regla **subclaseEnTabla**.

La regla **asociacionEnClaveForanea** se utiliza para traducir una asociación en una tabla con dos claves foráneas en el esquema relacional. En concreto, las claves foráneas de la tabla se crean incluyendo dos columnas que se corresponden con las claves primarias de las clases relacionadas por la asociación. Los tipos de estas columnas se definen de acuerdo con los tipos de datos utilizados por las claves primarias de las tablas en las que se han traducido las clases relacionadas. Esto se debe a que las claves foráneas deben contener los valores de las claves primarias de las tablas referenciadas (*ref*), estableciendo así la relación entre las tres tablas.

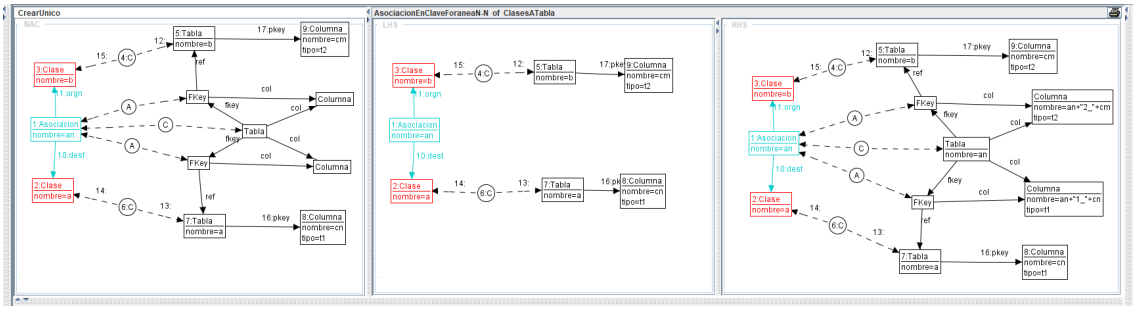

Figura 3.8: Regla **asociacionEnClaveForanea**.

3.3. Ejecución de la traducción

Para llevar a cabo esta transformación, volveremos a utilizar la opción **Step** del software AGG. Sin embargo, a diferencia de lo que hemos visto en el capítulo anterior, en este caso no se define una secuencia de reglas, sino que la ejecución involucra a todas las reglas de producción aplicables, de manera no determinista, hasta que no se pueda aplicar ninguna regla más. Lo que sí se puede indicar es una cierta prioridad en el orden de ejecución de las reglas. Para ello, el conjunto de reglas definidas se divide en n >0 capas, de modo que cada regla tiene asociado un número de capa. Al seleccionar la opción **Step**, las reglas se aplican durante el mayor tiempo posible en el orden de sus capas. Solo cuando las reglas de la capa i ya no son aplicables, el control pasa a la capa i+1.

En el caso particular de la transformación del diagrama de clases al esquema relacional la asignación de capas que se ha definido es la siguiente:

- La regla **claseEnTabla** está en la capa 0.
- La regla **subclaseEnTabla** está en la capa 1.
- La regla **atributoEnColumna** está en la capa 2.
- La regla **clavePrimaria** está en la capa 3.
- La regla **asociacionEnClaveForanea** está en la capa 3.

El criterio que se ha seguido para la definición de las capas es que cada regla de traducción utiliza elementos que han sido previamente creados en niveles anteriores, asegurando así un flujo ordenado y coherente en la transformación del modelo. De este modo, cada regla crea nuevos elementos que serán utilizados en niveles superiores, garantizando la correcta estructura y cohesión del esquema final. Este enfoque de capas y dependencias entre reglas asegura que el resultado de la traducción cumpla con todas las restricciones y requisitos establecidos, permitiendo una migración precisa y completa del diagrama de clases al esquema relacional. En la figura 3.9 se muestra el resultado de aplicar la traducción al diagrama de clases de la figura 3.2.

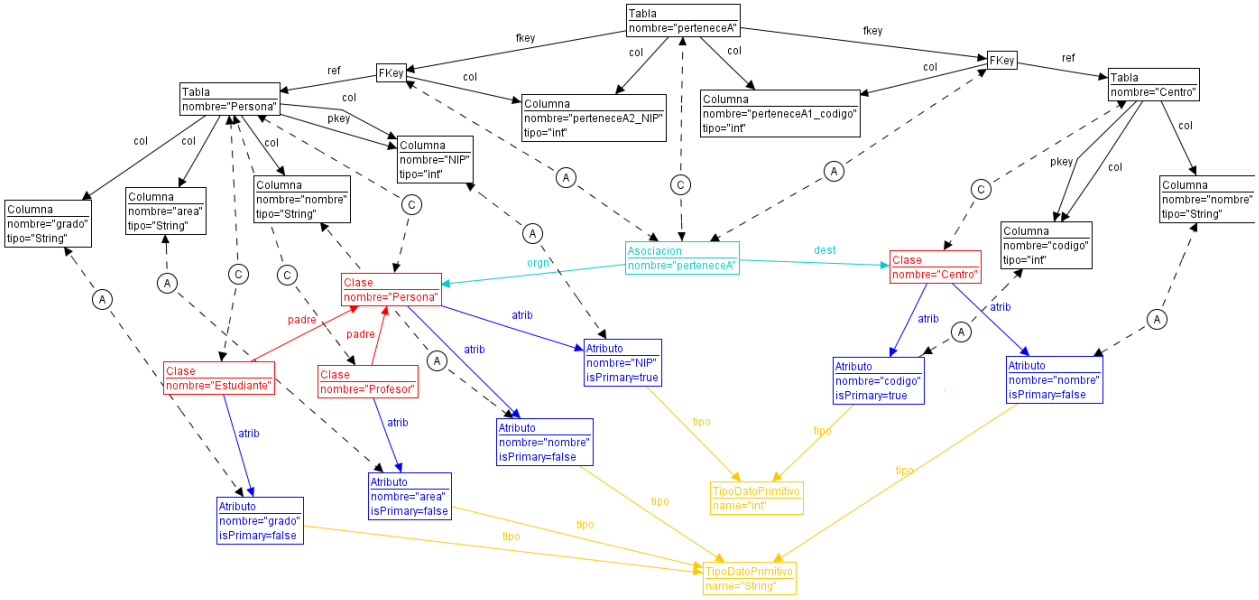

Figura 3.9: Diagrama de clases y esquema relacional tras aplicar las reglas en su orden correspondiente.

Bibliografía

[1] Ambler, S. W. (2005). Mapping objects to relational databases: O/R mapping in detail, 2006. URL `http://agiledata.org/essays/mappingObjects.html`.

[2] Bézivin, J., & Gerbé, O. (2001, November). Towards a precise definition of the OMG/MDA framework. In *Proceedings 16th annual international conference on automated software engineering (ASE 2001)* (pp. 273-280). IEEE.

[3] Bézivin, J., & Kurtev, I. (2005, November). Model-based technology integration with the technical space concept. In *Metainformatics Symposium* (Vol. 20, pp. 44-49).

[4] Cleve, A. (2010, September). Program analysis and transformation for data-intensive system evolution. In *2010 IEEE International Conference on Software Maintenance* (p. 16). IEEE.

[5] Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM systems journal, 45(3)*, 621-645.

[6] Da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures, 43*, 139-155.

[7] Heckel, R., & Taentzer, G. (2020). Graph Transformation for Software Engineers. *Springer International Publishing, doi*, 10, 978-3.

[8] Kurtev, I., Bézivin, J., & Aksit, M. (2002). Technological spaces: An initial appraisal. *CoopIS, DOA, 2002*.

[9] Masson, T., Ravet, R., Ruiz, F. J. B., Serbout, S., Ruiz, D. S., & Cleve, A. (2020, January). Defining Referential Integrity Constraints in Graph-oriented Datastores. In *MODELSWARD* (pp. 409-416).

[10] Mens, T. (2006). On the use of graph transformations for model refactoring. *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, 219-257.

[11] Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic notes in theoretical computer science, 152*, 125-142.

[12] Rumbaugh, J., Jacobson, I., & Booch, G. (1999). The Unified Modeling Language (UML) reference manual. USA, Massachusetts: Addison Wesley Longman Inc.

[13] Smith, J. M., & Smith, D. C. (1977). Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems (TODS), 2(2)*, 105-133.