Especificación de un metamodelo para apoyar y extender la propuesta *TD–MBUID*

Daniel Fernando Orozco Morales

Maestría en Ingeniería

William Joseph Giraldo Orozco Ph.D.

Director

Helmuth Trefttz Gómez Ph.D.

Asesor EAFIT

Universidad EAFIT

Escuela de Ingeniería

Departamento de Informática y Sistemas

Medellín – Mayo de 2014

Dedicatoria

A Dios, que ilumina mi camino con fortaleza y su Espíritu. A mi esposa, Mónica, por su incondicional amor, soporte y presencia; a mi madre, Luz Marina, por ser valuarte de mi vida y haberme dado todo.

Agradecimientos

Sólo palabras de admiración y gratitud puedo expresar a mi director de trabajo de grado doctor William Joseph Giraldo Orozco por su dedicación y paciencia, conocimiento y entusiasmo; mentor: muchas gracias por sus incontables horas, correcciones, instrucciones y lecciones impartidas y por siempre guiarme en la dirección correcta.

A EAFIT, en atención al doctor Helmuth Trefftz, coasesor de este trabajo de grado, por compartir su conocimiento e interés en el logro de esta investigación y, como no agradecer, al cuerpo de profesores de EAFIT que con sus capacidades, habilidades y entrega permitieron hacer realidad la primera cohorte de esta maestría en Armenia.

También agradezco a mis compañeros de maestría, pues con ellos compartí esta aventura de conocimiento.

Resumen

Este trabajo de grado propone una nueva herramienta CASE, basada en trabajos anteriores, para la generación automática de interfaces gráficas de usuario.

La investigación realizada se centró, básicamente, sobre el trabajo de Giraldo[26]. En este trabajo, Giraldo expone toda una integración de Procesos y Notaciones, y en lo que respecta a la presente investigación se retomó y continuó el tema de la integración de Notaciones, por lo que el resultado de este trabajo de grado es una herramienta CASE propuesta que integra las notaciones CIAN y UML logrando combinar el diseño, el prototipado (la generación) y la evaluación de las interfaces gráficas de usuario.

La manera en cómo se desarrolló este trabajo de grado y su resultado fue una dinámica sistemática. Inicialmente, se estudiaron los metamodelos iniciales de Chico y CIAN, luego, se integraron estos dos metamodelos recibiendo nuevas funcionalidades, entre ellas: El diagrama de transformación y la gestión propia de los diagramas al interior del metamodelo resultante. Una vez cumplido este primer objetivo, se procedió con la definición de la sintaxis concreta; en otras palabras: la construcción del DSL gráfico que soportara la generación de distintos tipos de diagramas y sus respectivos elementos y relaciones.

Teniendo ya la herramienta construida, el esfuerzo siguiente se centró en los algoritmos de transformación, tanto para transformar de modelo a modelo como para transformar de modelo a código. Finalmente, se procedió a validar todo lo anteriormente realizado a través del planteamiento de un caso de estudio.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Debido a los rápidos cambios de hoy en las organizaciones y sus negocios, muchos departamentos de sistemas tienen el problema de adaptar rápidamente las interfaces de usuario de las aplicaciones interactivas a estos cambios. Entre estos cambios se pueden anotar: reasignación de personal, redefiniciones de tareas y/o procesos, soporte de nuevas plataformas de computación, migración de plataformas de computación existentes a nuevas plataformas, usuarios con mayores demandas, incremento en la necesidad de interfaces de usuario mas usables, fusión de tareas y procesos, redefinición de la estructura organizacional, evolución/cambios en el negocio; estos cambios modifican el contexto de uso afectando en ocasiones la interacción de los usuarios y el rol que desempeñan en sus organizaciones.

El proceso de desarrollo de interfaces de usuario en las organizaciones, generalmente, no refleja la implicación de cualquier cambio en el ciclo de vida del proceso de desarrollo. Las organizaciones reaccionan de diferente manera al proceso de desarrollo de sus interfaces de usuario[41]. Algunas usan el enfoque ascendente (bottom-up), por ejemplo, cuando comienzan por la recuperación de pantallas existentes de entrada/salida, rediseñan y completan el núcleo funcional y la nueva interfaz de usuario es validada por el cliente; otras organizaciones prefieren modificar el modelo de dominio y el modelo de tareas para mapear diseño adicional en la pantalla (esto es enfoque top-down), por citar un par de ejemplos.

La diversidad de los enfoques adoptados en las organizaciones para atender esta serie de situaciones motivan a pensar en soluciones más flexibles para que se adapten con mayor facilidad y menor rigidez al manejo e información solicitada y manipulada en el desarrollo de las interfaces de usuario.

Esta motivación está en la línea de los sistemas de diseño basado en modelos, que están tomando cada vez una mayor importancia en el campo del desarrollo de Interfaces de Usuario (IU), formulando un modelo conceptual, apoyado en los datos y tareas del negocio, antes de facilitar una pantalla final a un usuario de la organización.

1.1. Definiendo el contexto

Este trabajo de grado está enmarcado en el dominio de la *Ingeniería de Software* bajo el marco de la *Ingeniería Dirigida por Modelos (MDE)* usando el framework CIAF *(Collaborative Interactive Application Framework)* y apoyándose en el contexto del *Desarrollo de Interfaces de Usuario (HCI)* haciendo uso de la aproximación metodológica *Task & Data–Model Based User Interface Development (TD–MBUID)*, propuesta desarrollada por el doctor William Joseph Giraldo Orozco [26].

Lo propuesto en este trabajo es realizar, refinar y extender un metamodelo sobre el framework *CIAF*, de un mecanismo de mapeo sobre y de los diferentes modelos que el framework *CIAF* manipula y almacena, centrándose específicamente sobre la extensión al metamodelo que corresponde en la adición del modelo de transformación.

Este metamodelo debe servir como base conceptual para el apoyo de la notación del lenguaje y soportar la metodología a la vez de ser el *core* para la especificación–creación de la herramienta.

Un metamodelo describe la posible estructura de los modelos – de una manera abstracta, define las construcciones de un lenguaje de modelado y sus relaciones, así como las limitaciones y reglas de modelado [86]. Un metamodelo incluye los conceptos, reglas, diagramas y anotaciones de un lenguaje dado [34]. Un metamodelo es un modelo explícito de las construcciones y las normas necesarias para construir modelos específicos dentro de un dominio de interés [23]. Un metamodelo es un modelo que sirve de base para la construcción de otro modelo. Aunque ambos son modelos, uno se expresa en términos del otro, en otras palabras, un modelo es una instancia conforme al otro modelo [28]. Un metamodelo es simplemente el modelo de un modelo, y si ese modelo es en sí mismo un metamodelo, el metamodelo es en realidad un meta–metamodelo [15]. Un metamodelo es un "modelo de un modelo", en el que los elementos del meta-modelo describen el significado (semántica) de los diagramas concretos generados a partir de la estructura del metamodelo [39].

En el marco *MDE* se presentan los conceptos de lenguaje, modelado y herramienta. Son estos conceptos los que soportan *CIAF* y lo que Giraldo [26] denomina como *'triada'* de *CIAF*. Este conjunto de conceptos deben estar representados en el metamodelo que se propone conformar en este trabajo de grado. En cuanto al concepto de lenguaje, el metamodelo debe representar la sintaxis abstracta lo que le proporciona al lenguaje la capacidad de representar sus conceptos. Igualmente, para representar gráficamente estos conceptos, el metamodelo debe estar capacitado para exponer la sintaxis concreta del lenguaje (la notación gráfica del mismo). Se concibe que la metodología debe estar apoyada por su notación y sus herramientas.

En este conjunto denominado *'triada'* ninguno de los tres conceptos puede estar separado del conjunto. Esta *'triada'* para poder tener una entidad necesita tener un marco conceptual que considere estos tres conceptos de manera independiente, pero también de manera integrada a través de sus relaciones. En este sentido se entiende que cuando se va a diseñar un lenguaje también debe pensarse en la herramienta sobre la cual el ingeniero trabaja. Es decir, cuando el desarrollador usa un lenguaje lo hace es a través de una herramienta.

El tema de este trabajo de grado es realizar el metamodelo que actúa como la base conceptual para la definición del lenguaje y la herramienta.

Esta implementación se aborda con la perspectiva de *MBUID (Model–Based User Interface Development)* que se soporta en la metodología *TD–MBUID* [26]. En esta perspectiva se abordan los niveles de abstracción del enfoque MBUID: Abstracto; Concreto y Final. Es a través de estos niveles que se pretende realizar un mapeo de qué elementos se identifican y se transforman en otros de un nivel a otro nivel de abstracción.

En este trabajo de grado se pretende realizar una infraestructura de metamodelo que aporte un sistema de transformación además de la sintaxis abstracta y sintaxis concreta para el diagrama de transformación.

1.2. Situación Problémica

Realizando una revisión con respecto de la propuesta *Task & Data–Model Based User Interface Development (TD–MBUID)*[26], al nivel del lenguaje que la soporta, se ha identificado que:

- Las transformaciones en esta herramienta están codificadas para un modelo en particular. Si el usuario modifica algunos parámetros del modelo, como nombres de paquetes, por ejemplo, las transformaciones no se ejecutan.
- El *Mapping* entre el modelo de interacción y el modelo de diseño no está definido como un elemento de modelado y se realiza a partir de *shortcuts*.
- No soporta el modelado de datos de dominio ni las interfaces de usuario de negocio.

1.3. Objetivos

1.3.1. Objetivo General

Desarrollar un metamodelo que apoye la herramienta denominada *CIAT.TDMBUID* [26], que soporte funcionalidades inexistentes en las situaciones problemáticas definidas anteriormente.

1.3.2. Objetivos Especifícos

1.3.2.1. Conformar la nueva sintaxis abstracta y sintaxis concreta del lenguaje y la herramienta *CIAT.TDMBUID* a partir de los desarrollos previos, teniendo en cuenta la incorporación del diagrama de transformación.

1.3.2.2. Conformar el marco conceptual del lenguaje y la herramienta *CIATTDMBUID*.

1.3.2.3. Conformar el metamodelo de la herramienta *CIAT.TDMBUID*.

1.3.2.4. Desarrollar nueva herramienta *CIAT.TDMBUID* a partir del metamodelo propuesto.

1.3.2.5. Validar el metamodelo a partir de la herramienta mediante caso de estudio.

Capítulo 2

Estado del Arte y Marco Conceptual

El tema del trabajo de grado está enmarcado en el dominio de la Ingeniería de Software bajo el marco de la *Ingeniería Dirigida por Modelos (MDE)* usando el framework *CIAF (Collaborative Interactive Application Framework)* y apoyándose en el contexto del *Desarrollo de Interfaces de Usuario (HCI)* y haciendo uso de la aproximación metodológica *Task & Data – Model Based User Interface Development (TD–MBUID)*, propuesta Giraldo en [26].

2.1. *Estado del Arte*

2.1.1. *MDE (Model–Driven Engineering)*

El marco de *MDE (Model–Driven Engineering)* [23] ayuda a descubrir los elementos de un sistema a partir de la creación de modelos enfocados sobre los conceptos de dominio y no tanto sobre los conceptos de informática. Uno de los objetivos del enfoque *MDE* es especificar y explicitar los términos del negocio en modelos durante todo el proceso de desarrollo de software.

A partir de este objetivo, este enfoque está centrado en que los desarrolladores se enfoquen solo en modelos sin considerar aspectos de implementación. Se pueden trabajar independientemente los detalles de las plataformas específicas. Esto hace que el sistema desarrollado se acerque más a las necesidades del usuario final, el cual tendrá mayor funcionalidad en un menor tiempo. Realizar modificaciones sobre los modelos será mucho más rápido y seguro que hacerlo sobre el código final.

El modelo constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema y los desarrolladores de software. Por lo tanto es de fundamental importancia que exprese la esencia del problema en forma clara y precisa [57].

Kleppe[35] afirma, que si se trabaja con modelos, se pueden obtener importantes beneficios tanto en la productividad (prestando mayor atención, por parte de los desarrolladores, a la solución del problema a partir del dominio y no tanto en los detalles técnicos), como en la portabilidad (dependiendo de las herramientas de transformación usadas) e independencia de la plataforma de sistema de software, en la interoperabilidad (haciendo referencia a los *'brigdes'* puentes – mapeo de elementos entre distintas plataformas y a la independencia de los fabricantes), en el mantenimiento y la documentación (esta se mantiene a un alto nivel de abstracción partiendo desde los modelos conceptual e independiente de plataforma).

En el contexto de *MDSD (Model–Driven Software Development)* [86], corriente especializada del enfoque *MDE*, un metamodelo describe la posible estructura de los modelos – en una manera abstracta, define los constructores de un lenguaje de modelado y sus relaciones, como también las restricciones –*constraints*– y las reglas de modelado [86], como también los conceptos y notaciones de diagramas de un lenguaje dado [34]. Gronback define un metamodelo como un modelo que provee las bases para construir otro modelo [28]. Un metamodelo define la estructura de un modelo así como su semántica. La semántica que puede ser expresada es específica para un dominio, por lo tanto, el metamodelo es una representación de ese dominio [4]. Para Beydeda[5] un metamodelo es definido por los modelos intrínsecos de un dominio descritos en otro modelo de mayor nivel de abstracción. En [39] un metamodelo es un "modelo de un modelo", en el que los elementos del metamodelo describen el significado (semántica) de los diagramas concretos generados a partir de la estructura del metamodelo.

2.1.2. *DSL (Domain Specific Language)*

Todo este trabajo con modelos debe siempre obtener un resultado. Bien sea un nuevo modelo, metamodelo, o código del aplicativo para el cual se ha modelado un dominio. Pero ese resultado, más allá de convertirse en un modelo de mayor (o menor) nivel de abstracción, debe aportar condiciones únicas para el dominio modelado. Este aporte de condiciones particulares sobre el dominio está dado por el *DSL (Domain Specific Language)*. El DSL, según describe Fowler[20], es un lenguage de un propósito determinado, cuya representación puede ser gráfica o textual, adaptado a problemas concretos de un dominio. Para Völter[86], un DSL sirve para el propósito de realizar los aspectos claves de un dominio formalmente expresables y modelables. Y Gronback[28] define DSL como un lenguaje diseñado para ser útil para un conjunto específico de tareas. Ghosh[24] indica que un DSL es un lenguaje de programación que está dirigido a un problema específico.

Un DSL gestiona complejidad a través de abstracción organizada, en software existente y/o en literatura técnica, en la que participan expertos de un dominio. El DSL es un lenguaje de modelado especializado que se orienta hacia un dominio estrecho (específico). Entre estos dominios se pueden mencionar *i)* Verticales: dominio de negocio; telecomunicaciones; finanzas; banca; seguros; etc., y *ii)* Horizontales, subdivididos en Transacciones y Técnicos: persistencia; interfaces de usuario; telecomunicaciones; etc. Fowler[20] manifiesta las cualidades deseables de un DSL:

- Expresividad limitada.
- Integridad conceptual.
- Manejabilidad.
- Promoción de la escritura de código conciso y legible.
- Regularidad.
- Fiablidad.
- Consistencia con notaciones aceptadas convencionalmente.

Un DSL puede ser interno (embebido) o externo, en este caso, se representa en un lenguaje diferente al lenguaje de programación principal. Un DSL se traduce, generalmente, en llamadas a bibliotecas de subrutinas. Un DSL, para Ghosh[24], es un artefacto que constituye una parte importante de un proceso de mapeo. Esto es, buscar la forma de expresar un problema en el vocabulario (contexto) de un dominio particular y, posteriormente, en el modelo de dominio de

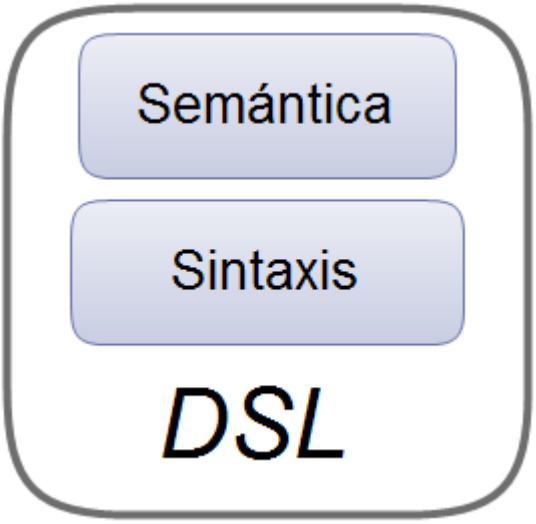

Figura 2.1: Concepción de DSL. Presentación propia del autor

la solución. El modelo de aplicación de este concepto es la esencia de lo que se denomina *Lenguaje de Dominio Específico (DSL).* Un DSL es una manera ideal para modelar el tipo de reglas presentes en una aplicación de negocios: reglas que deben ser explícitas, legibles y declarativas. Este Lenguaje de Dominio Específico *(DSL)* está definido, para Völter[86], por un metamodelo. Un metamodelo describe los conceptos que pueden ser usados para modelar el modelo. La formalización para un lenguaje toma forma en la construcción del metamodelo. Gronback, en [28], manifiesta que la estructura de un DSL es capturada en su metamodelo, comúnmente referida como su sintaxis abstracta.

Björner[17] expresa que por sintaxis podemos referimos a las formas en que las palabras están dispuestas a mostrar el significado (es decir, semántica) dentro y entre las oraciones, y las reglas para formar oraciones sintácticamente correctas. Al respecto, Kelly[34], manifiesta que la sintaxis abstracta de un lenguaje de modelado se especifíca normalmente en un metamodelo. La sintaxis de un lenguaje de modelado es más que sólo palabras reservadas. Se presenta con frecuencia que también incluye las reglas gramaticales que se deben seguir al especificar los modelos. Giraldo, en su propuesta [26], manifiesta que la sintaxis especifíca la estructura conceptual de un lenguaje por medio de sus construcciones (elementos de modelado de que se compone el lenguaje), sus propiedades y las conexiones entre sí.

Para Beydeda la sintaxis de un formalismo está dada por lo concreto y la sintaxis abstracta [5]. Gasević[23], afirma que toda lógica formal tiene una sintaxis bien definida que determina cómo las oraciones se construyen en el lenguaje, una semántica que determina el significado de las oraciones, y un procedimiento de inferencia que determina las oraciones que se pueden derivar de las otras oraciones. Boca[7] refiere que la sintaxis de un lenguaje puede ser descrita sin tener en cuenta la semántica prevista de los programas, mientras que la semántica no puede ser descrita sin referencia a la sintaxis del lenguaje. Para Kelly[34] la sintaxis especifíca la estructura conceptual de un lenguaje: las construcciones de un lenguaje de modelado, sus propiedades y las conexiones entre sí. La sintaxis, según Giraldo[26], define qué elementos de modelado existen en el lenguaje y cómo estos elementos de modelado se construyen en términos de otros.

Cuando se hace referencia al significado de los conceptos dentro del modelado y que tienen algún significado, se habla de *semántica*. Cuando se agrega un elemento en un modelo o se conectan elementos juntos, dentro de un modelo, se crea significado [34].

Un DSL, desde la perspectiva de Bärisch, define una sintaxis concreta o notación formal para expresar modelos concretos. La sintaxis abstracta define los conceptos básicos para la construcción del modelo, se determina por la semántica estática del metamodelo[4]. Según Völter[86], la sintaxis abstracta simplemente especifíca como luce (de apariencia) la estructura del lenguaje; la sintaxis concreta de un lenguaje especifíca lo que un analizador *(parser)* para el lenguaje acepta. Podría afirmarse, entonces, que la sintaxis concreta es la realización de una sintaxis abstracta. Beydeda[5] afirma, que la sintaxis concreta especifíca la representación legible de los elementos notacionales abstractos. Por su parte, Pastor[62], indica que la notación puede expresar que cada concepto (primitivas conceptuales o patrones conceptuales) se definen en el lenguaje de especificación en un esfuerzo para hacer el modelado gráfico de una tarea visual e intuitivo. Cuando el uso de una notación gráfica no es aplicable a todas las primitivas, la sintaxis del lenguaje de especificación puede ser utilizado contextualmente, es decir, mediante contextos estas primitivas tienen que ser correctamente asociadas a los elementos gráficos correspondientes. Según Ghosh[24] un DSL contiene la sintaxis y la semántica que los conceptos del modelo en el mismo nivel de abstracción como el dominio del problema tiene. Gronback[28], también afirma que el núcleo de un DSL es su sintaxis abstracta, que se utiliza en el desarrollo de casi todos los artefactos que siguen, incluyendo sintaxis gráfica concreta, transformaciones de modelo–a–modelo, y las transformaciones de modelo–a–texto. Típicamente, el primer elemento de un DSL a desarrollar es su sintaxis abstracta; la sintaxis abstracta para un DSL, por lo general, debe ser presentada para su uso por humanos, por lo que una o varias sintaxis concretas deben ser desarrolladas.

Giraldo, refiere en [26] que la sintaxis abstracta define qué elementos de modelado existen en el lenguaje y cómo estos elementos de modelado se construyen en términos de otros para describir los modelos del sistema que se desarrolla. La sintaxis abstracta indica la estructura y las reglas gramaticales del lenguaje.

También Giraldo[26], indica que la sintaxis concreta es la representación textual o gráfica del lenguaje que se diseña y es la que permite implementar modelos en dicho lenguaje. De esta forma, la sintaxis concreta facilita o dificulta el entendimiento de los diagramas y la organización de los modelos que describen un sistema. Kelly[34] expone que la sintaxis concreta hace referencia a los símbolos de la notación y la forma de representación que el lenguaje utiliza. Giraldo[26] afirma que la especificación de la sintaxis concreta debe ser uno de los pasos preliminares en la especificación de cualquier lenguaje para definir sus particularidades y el *mapping* con respecto de la sintaxis abstracta. Esta especificación tiene como partida el análisis visual de cómo se presentarán al usuario desarrollador los elementos de modelado del lenguaje. Este modelo define la estructura de componentes gráficos usados para representar los conceptos.

2.1.3. *MBUID (Model– Based User Interface Development)*

Lo que se presenta en este trabajo de grado es una herramienta que soporta la generación automática de interfaces gráficas de usuario a partir de la concepción de diferentes modelos y enfoques.

El enfoque *MBUID (Model– Based User Interface Development)* está definido, por Molina[50], como una aplicación del paradigma *MDE*; este enfoque facilita la especificación y construcción de sistemas considerando la diferenciación entre distintos niveles de abstracción:
Tres características de este tipo de entornos son definidas en [26]:

a) soporte para la generación automática de interfaces de usuario;

b) uso de métodos declarativos para la especificación de interfaces de usuario y

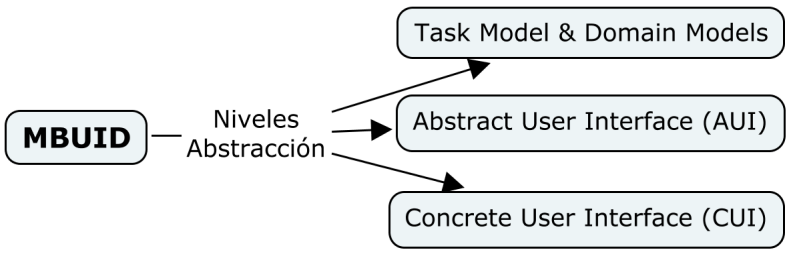

Figura 2.2: Niveles Abstracción del enfoque *MBUID.* Presentación propia del autor.

c) la adopción de metodología para soportar el desarrollo de interfaces de usuario.

En cuanto al dominio de este trabajo de grado se aborda un enfoque *MDE*; por lo que en el nivel de abstracción *Task Model & Domain Model* se puede establecer una correspondencia directa hacia el *Modelo Independiente de la Computación (CIM)*. En el nivel de abstracción *Task Model & Domain Model*, de la figura 2.2, se pueden realizar el modelo de tareas y el modelo de concepto de dominio.

En el primero, modelo de tareas, basado en la notación formal del *ConcurTaskTree CTT* que permite crear árboles de tareas con varios niveles de abstracción, [63], en donde se define la tarea como la forma en que el usuario puede lograr un objetivo en una aplicación de dominio específico. La tarea, según Limbourg[42], es clasificada en cuatro tipos:

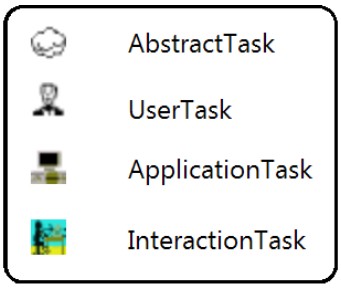

Figura 2.3: Clasificación tipos de tareas según Limbourg[42]. Notación *ConcurTaskTree CTT*[63].

Las tareas, para Giraldo[26], permiten mostrar la participación de los roles en relación con los datos que manipulan y permiten mostrar las subtareas que pueden componer una tarea.

Limbourg[42] clasifica las tareas de acuerdo a lo siguiente. La *tarea de usuario* se refiere a la acción cognitiva como tomar una decisión o adquirir información. Esta tarea es útil para predecir la ejecución en *runtime*. La *tarea interactiva (Interactive Task)* es la interacción del usuario con el sistema (por ejemplo, seleccionar un valor). La *tarea de sistema* es la acción que es realizada por el mismo sistema y la *tarea abstracta* es un constructor intermediario que agrupa tareas de diferentes tipos.

Las tareas tienen atributos: Frecuencia e Importancia. La *frecuencia* es la valoración de la frecuencia relativa de ejecución de una tarea; esta es evaluada en una escala de 1 a 5, donde 5 es la de mayor frecuencia. El atributo de *importancia* evalúa la importancia relativa de una tarea con respecto de los objetivos principales del usuario. Este atributo se mide, también, con una escala de 1 a 5 dónde 5 es la de mayor importancia.

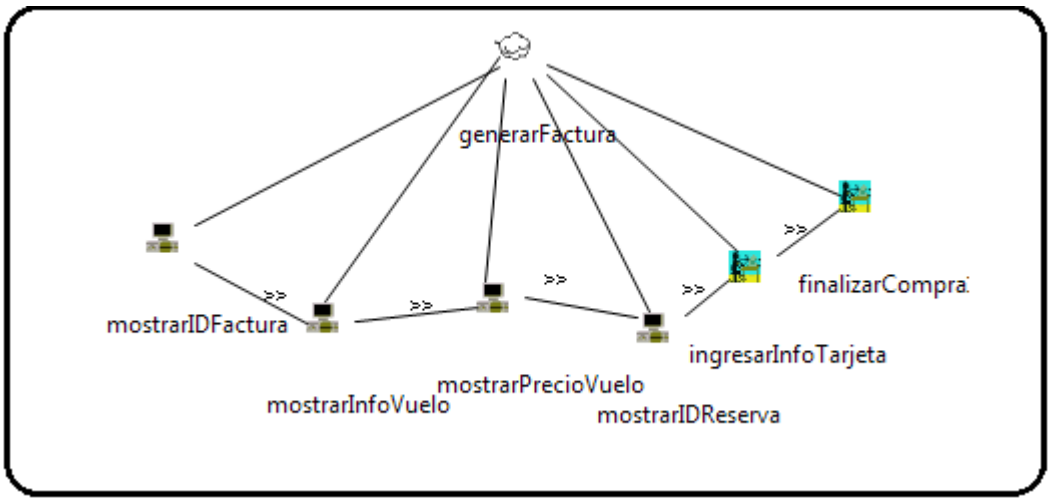

Figura 2.4: Modelo de Tareas. Presentación Propia del autor

Para Vanderdonckt, el *modelo de tareas (Task Model)* es un modelo que describe la tarea interactiva como es vista por el usuario final interactuando con el sistema [84]. Mientras que, para Giraldo[26], el *modelo de tareas (Task Model)* describe un ordenamiento lógico y temporal de las tareas que son realizadas por los usuarios en la interacción con un sistema. Los elementos individuales en un *modelo de tareas (Task Model)* representan las acciones específicas que el usuario puede llevar a cabo. Para Limbourg[42] un *modelo de tareas (Task Model)* se compone de tareas y las relaciones entre esas tareas.

En el mismo nivel de abstracción del *modelo de tareas (Task Model)* se ubica el *modelo de dominio (Domain Model)*. Para Limbourg, en [42], el modelo de dominio describe conceptos del mundo real y las interacciones, es decir, la interpretación de los usuarios y sus operaciones con esos conceptos. Vanderdonckt, en [84], comparte la misma definición: es una descripción de las clases de objetos manipulados por un usuario mientras interactúa con un sistema. Giraldo, a su vez, en [26], manifiesta que el *modelo de dominio (Domain model)* captura la semántica del contexto del sistema y define los requisitos de información para el desarrollo de la interfaz de usuario. Especifíca los datos que las personas utilizan, relacionados con las entidades del mundo real y las interacciones tal como son entendidas por los usuarios en relación con las acciones que se realizan sobre estas entidades. Los objetos del dominio son considerados como instancias de clases que representan los conceptos que son manipulados y que son totalmente independientes de la forma en la que se muestran en la pantalla y cómo se almacenan en el computador. Se usan los *diagramas de clases UML* como base para la representación [58].

El enfoque *MBUID (Model– Based User Interface Development)* define básicamente tres niveles de abstracción:

1. *Abstract User Interface (AUI)*
2. *Concrete User Interface (CUI)*
3. *Final User Interface (FUI)*

El nivel de Abstracción del enfoque MBUID, *Abstract User Interface (AUI)* , Giraldo, en [26], lo define como la representación de una expresión canónica de la renderización (representación) y la manipulación de los conceptos del dominio y las funciones, de manera que sea lo más independiente posible de la modalidad (auditiva, visual, táctil, etc.) y de la plataforma informática.

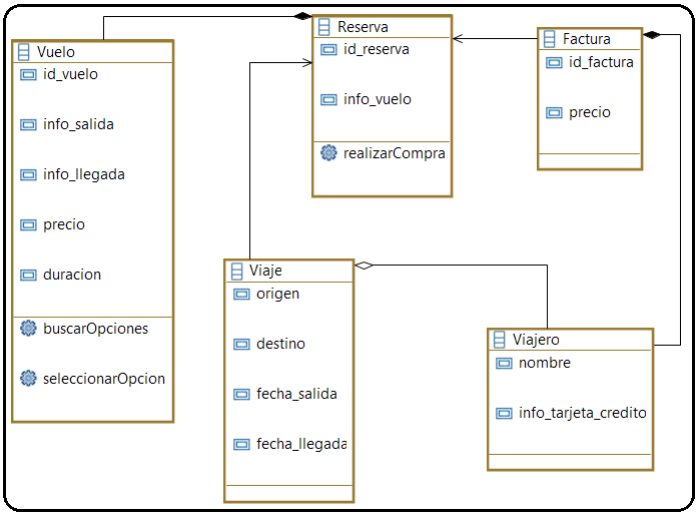

Figura 2.5: Modelo de Dominio. Presentación propia del autor.

Molina, en [50], indica que un AUI es la descripción abstracta del contenido y la estructura de la interfaz de usuario. Vanderdonckt, por su parte, en [84], realiza una correspondencia directa entre AUI y el modelo independiente de la plataforma (PIM). En el AUI, se definen Contenedores Abstractos *(Abstract Container – AC)* y Componentes Individuales Abstractos *(Abstract Individual Component – AIC)*, agrupados en subtareas de acuerdo a varios criterios, un esquema de navegación entre el AC y la selección del AIC para cada concepto abstracto para que sean independientes de cualquier modalidad. En este nivel, la interfaz de usuario se compone principalmente de definiciones de entrada/salida, junto con las acciones que se deben realizar sobre esta información.

Limbourg define, en [42], AIU en términos de Objeto de Interacción Abstracto *(Abstract Interaction Object – AIO)* y de Relaciones de Objeto de Interacción Abstracto. Cada AIO puede ser usado para representar rápidamente la interacción necesaria con el sistema. El modelo AUI [42] está compuesto por *AIO (Abstract Interaction Object)* y *Abstract User Interface RelationShip*. Los AIO están compuestos por Abstract Individual Component (AIC) y describen la interacción de los objetos de una manera que es independiente de la modalidad y que será representado en el mundo físico. La interacción de un AIC puede representarse por medio de *facets* (descripción de una función particular de un AIC):

Limbourg, en [42], define cuatro tipos de facets:

- *input:* representa la entrada de datos en el sistema;
- *output:* representa la información proporcionada por el sistema al usuario;
- *navegation:* representa la transición de un componente de la interfaz a otro;
- *control:* describe los vínculos (relaciones) entre un AIC y la(s) función(es) del sistema.

Adicionalmente, Giraldo [26], define:

- *action:* representa un evento de interacción que desencadena un disparo que cambia el estado de los componentes de la interfaz; y
- *group:* que representa un conjunto de AIOs que definen un componente más complejo.

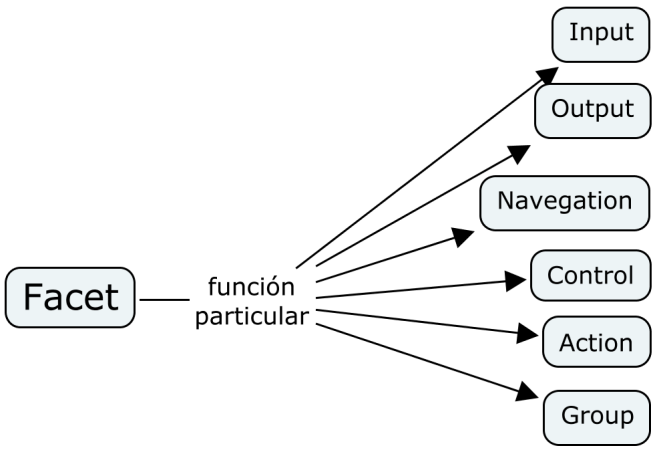

Figura 2.6: Facets. Presentación propia del autor.

En el conjunto de AIOs, también se representa el *Abstract Container (AC)*. Limbourg[42], define este conjunto como la agrupación lógica de otros contenedores abstractos o de AICs. Estos ACs soportan la ejecución de tareas lógicas/semánticamente conectadas.

En este modelo, AUI, también se tienen las *Abstract User Interfaces Relationship*. Estas relaciones son el panorama inicial para determinar el modelo de trazabilidad de la herramienta propuesta. En [26, 42], se definen las siguientes relaciones abstractas:

- *decomposition:* especifica la descomposición de una estructura jerárquica de contenedores abstractos (AC) y los componentes individuales abstractos (AIC);
- *abstractAdjacency:* esta relación indica que dos AIOs son lógicamente adyacentes;
- *spatio– temporal:* especificación precisa del layout en tiempo o espacio;
- *Dialog Control:* permite especificar un flujo de control de la interacción entre objetos abstractos;
- *Mutual emphasis:* especifica la diferenciación de dos componentes en el nivel concreto.

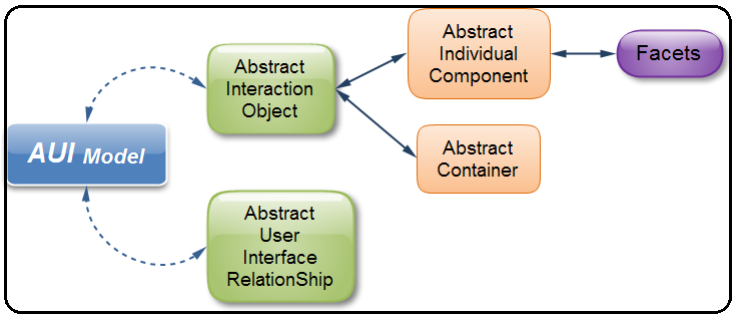

Figura 2.7: *Modelo AUI*. Presentación propia del autor.

El nivel de Abstracción, del enfoque MBUID, *Concrete User Interface (CUI)* para Limbourg, es el modelo de interfaces de usuario el que permite la especificación de la apariencia y el comportamiento de la interfaz de usuario con elementos que pueden ser percibidos por el usuario [42]. Este modelo está constituido por *Objetos de Interacción Concreta (CIO Concrete Interaction*

Object) que es la entidad que el usuario percibe/manipula independientemente del la representación (rendering). Estos CIO definen la apariencia (appaerance) de la interfaz de usuario. La clasificación de los CIOs, según la modalidad gráfica, se da en:

1. *Graphical Containers (GC)* como window, tabbed, table, menubar, dialog box, box;
2. *Graphical Individual Components (GIC)* que comprenden image components, text components, progressionbar, label, text input, combobox.

Por otro lado, el modelo CUI también se compone de relaciones, estas se denominan *Concrete User Interfaces Relationship*. Estas relaciones componen el comportamiento (behavior) que es considerado, por Limbourg[42], como el mecanismo de respuesta a eventos que resulta en un cambio de estado del sistema. Este cambio puede darse por un evento, que es un acontecimiento (ocurrencia) en tiempo de ejecución que activa una acción; una condición, que es la expresión de un estado que ha de ser verdad antes (pre-condición) o después (post condición) de realizarse una acción. Esta expresión se da en términos de *OR, AND, XOR* e *IMPLIES*; acción, es un proceso que resulta en un cambio de estado en el sistema, esta puede ser de tres tipos: a) method call, b) transformation system (cambio de propiedades en la interfaz de usuario) y c) transition (source y target).

Para Giraldo[26], el modelo CUI permite especificar el aspecto y comportamiento de la interfaz de usuario por medio de elementos que pueden ser percibidos por los usuarios. La traducción de AUI en CUI se aborda mediante la utilización de reglas de transformación disponibles en el lenguaje usiXML[41],[83]. Un Objeto de Interacción Concreto (CIO) es un componente de la interfaz de usuario que puede ser manipulado o percibido por el usuario. UsiXML[83], es la base tecnológica de *TD– MBUID* y proporciona dos tipos de interfaz concreta: a) contenedores gráficos y b) componentes individuales gráficos.

Así como en el AUI existen las relaciones, en el CUI también se presentan, pero a nivel gráfico:

1. *transición gráfica (Graphical Transition):* especifica vínculos entre los diferentes contenedores que componen la interfaz gráfica. Los tipos de transición están clasificados en: abrir, cerrar, minimizar, maximizar, restaurar,
2. *alineación (Alignment):* puede ser especificada entre cualquier componente individual perteneciente a la misma ventana;
3. *Adyacencia (Adjacency):* indica que dos componentes son topológicamente adyacentes;
4. *control de diálogo (Dialog Control):* especifica un flujo de control de la interacción entre objetos concretos utilizando los mismos operadores que para las tareas.

Para Vanderdonckt[84], CUI corresponde al modelo de plataforma especifica (PSM). Este paso es soportado por varias herramientas que ayudan a diseñadores y desarrolladores a editar, construir o dibujar una interfaz de usuario. Se concreta una interfaz abstracta, para un determinado contexto de uso, en CIO para definir el diseño de widgets y la navegación de interfaces. Se abstrae una *Interfaz de Usuario Final (FUI)* en una definición de interfaz de usuario que es independiente de cualquier plataforma de computación. Aunque un CUI hace explícito el look & feel definitivo de un FUI, sigue siendo un modelo a escala que sólo se ejecuta en un entorno particular.

El conjunto de estos modelos abstracto y concreto, se cierra con la representación final. Este cierre se da a través de un proceso de *reificación* que es la cosificación o, mejor, la materialización, y que Limbourg[43] define como la transformación de una descripción (o un conjunto de descripciones) en una descripción (o conjunto de descripciones) cuyo nivel de abstracción es menor que el del origen.

Esta reificación da como resultado una interfaz de usuario final – *(Final User Interface – FUI)* –, Vanderdonckt [84] realiza una correspondencia de este modelo (FUI) en el enfoque MDE directamente al código de la interfaz de usuario. Este nivel se logra cuando el código de la interfaz de usuario se produce a partir de los anteriores niveles: Task Model & Domain Model; AUI Model y CUI Model para la propuesta *TD– MBUID* y CIM, PIM y PSM en *MDE*. Este código de interfaz de usuario puede ser interpretado (por ejemplo en un navegador web) o ejecutado (compilación del código).

Para realizar un seguimiento de nivel a nivel, de los objetos que se trabajan y que dan como resultado una FUI, es requerido abordar la trazabilidad de los objetos por estos diferentes niveles.

A este seguimiento se le conoce como *Modelo de Trazabilidad* o *mapping*. Para Vanderdonckt [84], el *modelo de mapeo (mapping model)* está definido como un modelo que contiene una serie de relaciones establecidas (mapeadas) entre modelos y/o elementos de modelos. Molina[50] lo define como el proceso de convertir una representación abstracta de un sistema en una representación final (considerando los elementos de la plataforma en la que se implementa). En este modelo (mapping model) se especifica la correspondencia entre lo abstracto y la sintaxis concreta de las notaciones (CTT para el caso del *Task Model* y Class Diagram de UML para el caso del *domain model)*. El modelo de mapeo interconecta los diferentes elementos modelados por el usuario y construidos por medio de los editores. El modelo de mapeo, en la herramienta *CIAT* se define en el archivo gmfmap (del entorno EMF, que se explicará en el siguiente capítulo). También aquí, en este archivo, se incluye el conjunto de constraints (restricciones) de semántica.

Limbourg[43], plantea la problemática del modelo de mapeo debido al manejo desigual entre los modelos descriptivos y los modelos generativos; indica que: los modelos y las relaciones se definen solo para un proyecto o herramienta que no permiten los conceptos de reuso y compartir con otro contexto, esto impide que otros equipos de desarrollo usen estas especificaciones; las definiciones siguen siendo, principalmente, físicas y no lógicas.

En esta propuesta, se requerirá realizar una definición estructural del mapeo, sobre los modelos y las relaciones que se consideren dentro de los elementos de un mismo modelo como las relaciones entre modelos. Esta definición estructural deberá ser incluida en el archivo gmfmap de la herramienta CIAT.

Para cumplir con este objetivo, en particular, se propone realizar un recorrido para definir y encontrar las relaciones entre el modelo de dominio (domain model) y los modelos de interfaces. Toda relación posee un origen (source) y un destino (target). La intención del modelo de trazabilidad es poder conocer un elemento concreto que reifica (materializa) un objeto abstracto o, qué objeto abstracto es una abstracción de qué objeto concreto; igualmente, el modelo de trazabilidad establecería las relaciones directamente entre las tareas del *Task Model* y los elementos del dominio *(domain model)*.

Toda esta propuesta de desarrollo de una nueva herramienta está basada en un marco de desarrollo. Este marco de desarrollo es *CIAF (Collaborative Interactive Application Framework)* [85]. Este es un marco de desarrollo conceptual, metodológico y tecnológico para el desarrollo de sistemas interactivos que puede ser usado directamente, extendido o personalizado. En CIAF se integran tres notaciones:

- *CIAN* que cobija el aspecto de colaboración y HCI;
- *UML* que especifica la funcionalidad de sistemas informáticos y
- *UsiXML* que permite describir la interfaz de usuario para múltiples plataformas.

CIAF es un marco que se centra más en la especificación a nivel de proceso y no se centra en las pruebas de usabilidad. Este marco especifica solo las tareas de desarrollo de la interfaz de usuario; el usuario debe incluir las tareas de su metodología de desarrollo donde se encuentren componentes relacionados con la funcionalidad.

2.2. *Marco Conceptual*

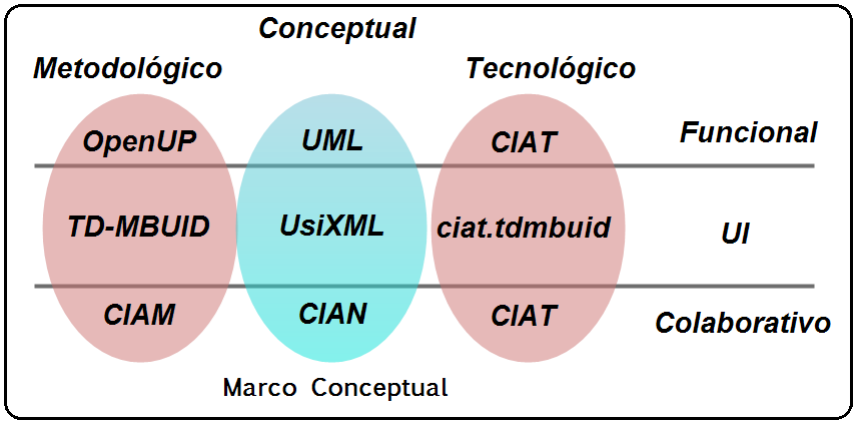

Figura 2.8: Marco conceptual de este trabajo

2.2.1. *CIAM (Collaborative Interactive Applications Methodology)*

CIAM [52] *(Collaborative Interactive Applications Methodology)* es el enfoque metodológico seleccionado para abordar el diseño conceptual de aplicaciones que soportan los grupos de trabajo. Esta se basa en el uso de notaciones específicas para el diseño de aplicaciones interactivas. Esta metodología pretende establecer una conexión entre los modelos de requerimientos a alto nivel con los modelos de interacción de más bajo nivel con el propósito de derivar la interfaz de usuario más directamente[26].

Esta metodología implica la adopción de distintos puntos de vista a la hora de abordar la creación de modelos conceptuales para este tipo de sistemas colaborativos pues, dar soporte al trabajo en grupo, resulta complejo dado el origen de las áreas de conocimiento de donde provienen los distintos integrantes del grupo. Las primeras etapas abordan un modelado centrado en el grupo, pasando en fases posteriores a un modelo más centrado en el proceso (cooperativo, colaborativo y de coordinación), aproximándose, a medida que se desciende en el nivel de abstracción, hacia un modelado centrado en el usuario, en el que se modelan las tareas interactivas, esto es, el diálogo que se da entre un usuario individual y la aplicación[22][52].

Esta propuesta metodológica es soportada por una herramienta *CASE* denominada *CIAT (Collaborative Interactive Application Tool)* [25], y permite la edición de modelos en cada una de las fases de la metodología, como el mapeo *(mapping)* y relación entre dichos modelos y diagramas de UML. *CIAT* es una herramienta basada en Eclipse[77] que ayuda a los desarrolladores a especificar modelos en CIAN.

2.2.2. *CIAN (Collaborative Interactive Applications Notation)*

Esta metodología, *CIAM*, propone una notación específica denominada *CIAN (Collaborative Interactive Applications Notation)* [49], la cual promueve el modelado de la colaboración para soportar el modelado de sistemas de apoyo al trabajo en grupo, igualmente, esta notación permite especificar las peculiaridades de los sistemas colaborativos interactivos, y se complementa con la notación UML[58], que permite especificar la funcionalidad del sistema. Esta notación se centra en el modelado de la colaboración y la interacción con el usuario. CIAN, entonces, permite identificar el conjunto de elementos gráficos para el modelado bajo la metodología CIAM.

CIAN provee un conjunto especializado de elementos de modelado para representar las tareas individuales y en grupo, a distintos niveles de abstracción y de granularidad. También soporta adecuadamente el modelado de la colaboración humana, pero no promueve el modelado de la funcionalidad del sistema[26]. La notación CIAN permite la creación de una serie de modelos. Cada uno de ellos representa un aspecto o perspectiva distinta de modelado y debe plasmar, por tanto, un subconjunto específico de conceptos. Este subconjunto que compone el marco posee cuatro vistas: vista organizacional, vista de inter-acción, vista de datos y vista de interacción [51].

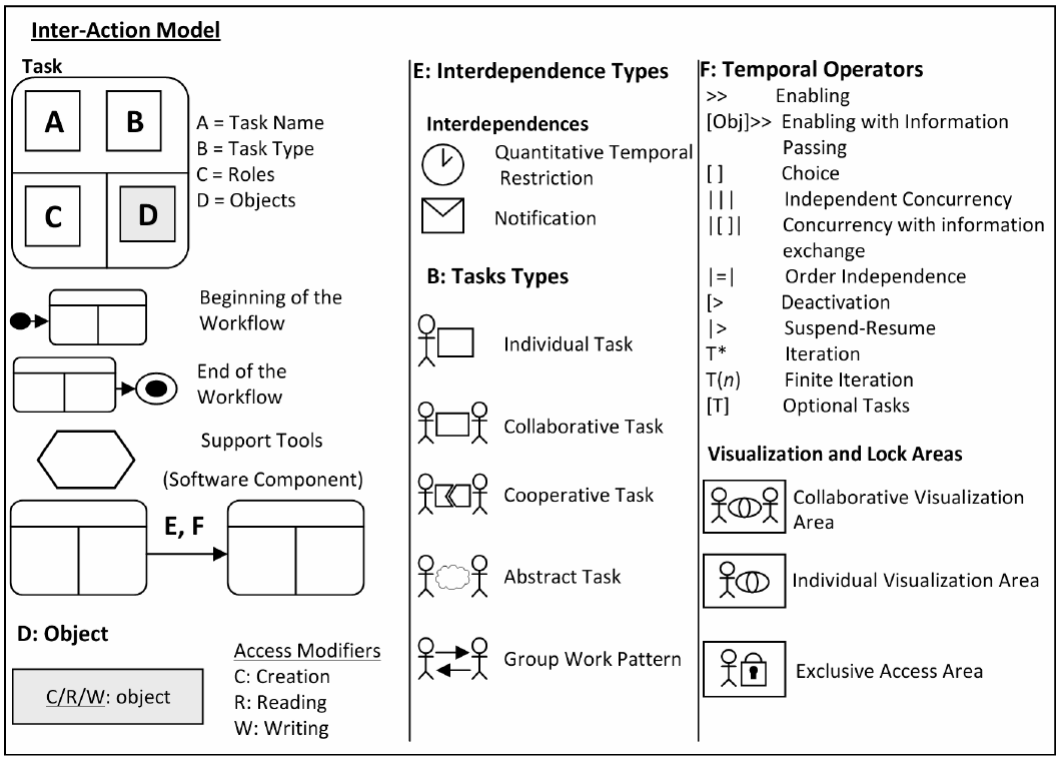

Figura 2.9: Elementos de CIAN. Tomado de [26]

2.2.3. *usiXML (User Interface eXtensible Markup Language)*

usiXML[40][83] se creó como una extensión de *XML* para describir la interfaz de usuario para múltiples contextos de uso, tales como gráficos, interfaces de usuario de voz, realidad virtual, e interfaces de usuario multimodales. Es un lenguaje específicamente basado en el marco de *referencia Cameleon*[10], que adopta cuatro etapas de desarrollo: tareas y conceptos, interfaz de usuario abstracta *(AUI)*, interfaz de usuario concreta *(CUI)* e interfaz de usuario final *(FUI)*.

El lenguaje *usiXML*[83] tiene por objeto describir interfaces de usuario con diferentes niveles de detalle y abstracción, en función del contexto de uso. usiXML provee soporte para el desarrollo de familias de interfaces independientes del dispositivo, de la plataforma, de la modalidad y del contexto. usiXML encapsula diferentes modelos y cuenta con soporte explícito para relacionar los distintos modelos, transformarlos y seleccionar partes de un modelo. Estos modelos son: modelo de tareas, de dominio, de presentación (interfaz abstracta, concreta y final), de contexto de uso, de mapping y de transformación [43]. El contexto de uso, a su vez, se descompone en modelo del usuario, de la plataforma y del entorno.

En la definición del lenguaje UsiXML [42] se proponen, para la sintaxis concreta, dos tipos de representación: una gráfica y una textual. La sintaxis gráfica, que se compone de cajas y flechas, se utiliza principalmente como un medio de representación de las reglas de transformación. Mientras que, la sintaxis textual es un lenguaje basado en XML (llamado usiXML), que se utiliza como un formato de intercambio entre diversas aplicaciones para explotar los modelos de especificación de interfaz de usuario en todas las etapas de desarrollo.

Sintaxis abstracta: se ha adoptado por completo la sintaxis abstracta de usiXML para soportar todos sus modelos y transformaciones. Aunque se conserva la estructura general de la sintaxis, se han realizado modificaciones, que se detallarán en el siguiente capítulo, para permitir su implementación en EMF[77], para garantizar la integración con otras notaciones, como *CIAN*, y para facilitar la implementación de reglas de transformación.

Sintaxis concreta: usiXML tiene una sintaxis concreta textual, basada en XML[9], que puede ser usada como un medio de comunicación entre herramientas. A nivel gráfico, adopta la representación de CTT[63] para las tareas y la representación de clases UML[58] para los datos.

2.2.4. Interfaz de Usuario TD-MBUID

TD-MBUID (Task & Data Model Based User Interface Development) es la propuesta de desarrollo de la interfaz de usuario basada en los modelos de datos (dominio) y de tareas (notación CTT [63]) [26]; de ahí su nombre.

El enfoque de desarrollo *MBUID (Model Based User Interface Development)* se centra en la especificación de alto nivel para generar la interfaz de forma automática. El entorno *MBUID* presenta tres características[26]:

1. el soporte para la generación automática de interfaces de usuario,
2. el uso de métodos declarativos para la especificación de las interfaces y
3. la adopción de una metodología para soportar el desarrollo de la interfaz.

El entorno MBUID promueve el desarrollo iterativo de modelos declarativos por medio del uso de editores gráficos y de lenguajes de alto nivel. Estos modelos incluyen la especificación del dominio, el contexto, la presentación, la actividad, el usuario y el diálogo. Una vez creados estos modelos, se incorporan una serie de guías, reglas y aspectos que están asociados a la presentación de la interfaz. Posteriormente se lleva a cabo una integración de estas especificaciones con el código de la aplicación que soportará la funcionalidad. De esta forma se tiene un método de ingeniería que reduce los errores, que permite conocer y entender a los usuarios para reducir su carga cognitiva y, al mismo tiempo, se mantiene la consistencia y la claridad en las especificaciones.

Capítulo 3

Entorno tecnológico usado

Este trabajo de grado tiene como entorno tecnológico *Eclipse Modeling Framework – EMF*; *Eclipse Graphical Modeling Framework – GMF* y *Graphical Editing Framework – GEF*.

3.1. *Eclipse Modeling Framework – EMF*

Eclipse Modeling Framework está definido como un framework de modelado y generación de código para herramientas de construcción y otras aplicaciones basadas en un modelo de datos estructurado. A partir de una especificación del modelo descrito en XMI, o importado (cómo se describe en 3.1.1.2), EMF proporciona herramientas y soporte de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases de adaptadores que permiten la visualización y edición de comandos basados en el modelo, y un editor básico [77]. En *EMF* se unifican *tres importantes tecnologías*: Java; XML y UML. Steinberg[15] define que el modelado en *EMF* es la base de granularidad fina para la interoperabilidad y el intercambio de datos entre las herramientas y aplicaciones en Eclipse. *Eclipse Modeling Framework* [77] es el proyecto de modelado central del proyecto *Eclipse Modeling Project (EMP)*[78]. *EMF* soporta la generación de código para construir herramientas y editores basados en árboles sobre un lenguaje de metamodelos llamado *ECORE*[79]. *EMF* soporta refinación, consulta, validación y transformación de modelos a través de sus subproyectos integrados y herramientas [18]. Gronback indica que este proyecto de modelado *(EMF)* está organizado lógicamente en proyectos que proporcionan las siguientes capacidades: el desarrollo de sintaxis abstracta, el desarrollo sintaxis concreta, la transformación de modelo–a–modelo, y la transformación de modelo–a–texto [28]. *EMF* es considerado como uno de los más exitosos enfoques de *MDE – Model-Driven Engineering* [33]. *EMF* se puede utilizar para describir y construir un modelo. Sobre la base de esta definición, el código Java puede ser generado y refinado. Este modelo aplicado se puede usar como la base para cualquier desarrollo de aplicaciones Java[54].

3.1.1. Definiendo el modelo de dominio *(.ECORE)* en *EMF*

El modelo de dominio describe los objetos de dominio de una aplicación. Un dominio se define como un área de interés a un esfuerzo de desarrollo particular [34]. En [17, 7] se entiende por *'Dominio'* un universo de discurso, un área de la actividad humana o de un área de la ciencia – lo suficientemente bien delineado para justificar que le da un nombre: *nombre de dominio*, y lo suficientemente bien distinguido de los universos "vecinos" o áreas para evitar la duplicacidad innecesaria y la confusión. Entonces, un *modelo de dominio* es una descripción cuidadosa de un número de entidades de dominio, funciones de dominio, eventos de dominio y comportamiento del mismo – formulados y detallados que es capaz de responder a las preguntas más relevantes

sobre ese dominio en particular [17]. En [47] un modelo de dominio no es un diagrama en particular, es la idea que el diagrama pretende transmitir. Se trata de una abstracción rigurosamente organizada y selectiva del conocimiento en ese particular.

Para Giraldo, en [26], el modelo de dominio es la información de las entidades de datos del sistema que identifican los conceptos principales utilizados en la ejecución de las actividades. El desarrollo del modelo de dominio se centra en el modelado de la información que caracteriza el negocio (la organización), no solo la que es útil para definir entidades de información que son transformadas en entidades del sistema informático. El modelo de dominio captura la semántica del contexto del sistema y define los requisitos del sistema de información para el desarrollo de la interfaz de usuario. El modelo de dominio especifíca los datos que las personas utilizan, relacionados con las entidades del mundo real y las interacciones tal como son entendidas por los usuarios en relación con las acciones que son posibles realizar sobre estas entidades. Los objetos del dominio son considerados como instancias de clases que representan los conceptos que son manipulados y que son totalmente independientes de la forma en que se muestran en la pantalla y cómo se almacenan en la máquina. Se usan los diagramas de clases UML[58] como base para la representación del modelo de dominio, ya que es el estándar *de facto* en el modelado de los diagramas de clase.

Para Bärisch, en [4], el dominio se representa a través de un metamodelo que es la definición de la estructura de un modelo así como su semántica. La semántica que se expresa es específica para un dominio, de manera tal, que, el metamodelo es la representación de ese dominio. En [86], el término *'dominio'* describe un contexto delimitado de interés o conocimiento. Los dominios pueden estar compuestos de pequeños *'subdominios'*. El modelo de dominio es el corazón del software [11].

Para Limbourg, en [42], un modelo de dominio captura los conceptos de la semántica del dominio de aplicación. Un modelo de dominio describe los conceptos del mundo real, y sus interacciones como se entienden por los usuarios y las operaciones que son posibles en estos conceptos. Un modelo de dominio es el resultado de un análisis Orientado a Objetos. Consiste en un esquema de diagrama de clases con atributos y métodos de cada objeto. Los conceptos de modelo de dominio son clases, atributos, métodos, objetos y relaciones de dominio. Sin conceptos de dominio una descripción de interfaz de usuario sería un cascarón vacío [42].

Ruscio[72] afirma, que el modelo de dominio es el metamodelo basado en *ECORE* (por ejemplo, la sintaxis abstracta) del lenguaje para el cual la representación y la edición deben ser proveídas. Además este modelo contiene todos los conceptos y relaciones que deben ser implementadas en el editor.

Este metamodelo, que sobre la plataforma tecnológica está representado por un archivo *.ECORE*, define la sintaxis abstracta que será manejada por el editor. Este archivo *.ECORE* define la semántica del lenguaje (sintaxis abstracta del modelo) así como la especificación de los elementos de la notación (sintaxis concreta del modelo) [50].

Gronback[28] afirma, que la base de todos los artefactos es el modelo de dominio.

Para crear el modelo de dominio, representado por el archivo *.ECORE* en EMF [81], existen cinco posibilidades:

3.1.1.1. *Ecore Editor*

EMF proporciona un editor basado en árbol y un editor de diagrama de clases *Ecore* con el propósito de que las propiedades de los modelos *Ecore* sean creadas y editadas [15].

3.1.1.2. Importar desde *UML*

El Asistente de Proyecto EMF proporciona esta opción únicamente para Rational Rose (archivos *.MDL*). La razón porque Rational Rose tiene esta condición especial es porque es la herramienta que utiliza para "arrancar" la implementación misma de EMF [15]. EMF soporta la importación de un modelo Rational Rose o un modelo UML2 para generar un modelo Ecore correspondiente [18].

Este proceso de importación se da en tres pasos: primero, crear el proyecto EMF, segundo, seleccionar el tipo de modelo a importar, en este caso ver figura 3.1 y, tercero, realizar el procedimiento de importación.

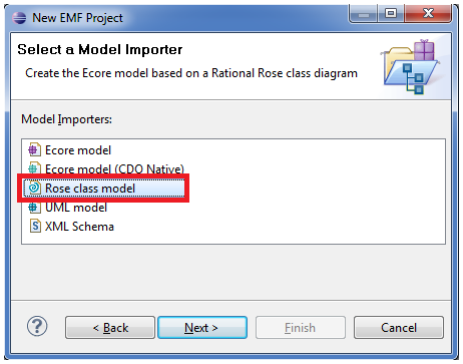

Figura 3.1: Selección de Tipo de Modelo a Importar. Presentación propia del autor.

Esta es la opción usada en este trabajo de grado. La siguiente figura representa la importación del modelo *ciat.tdmbuid*.

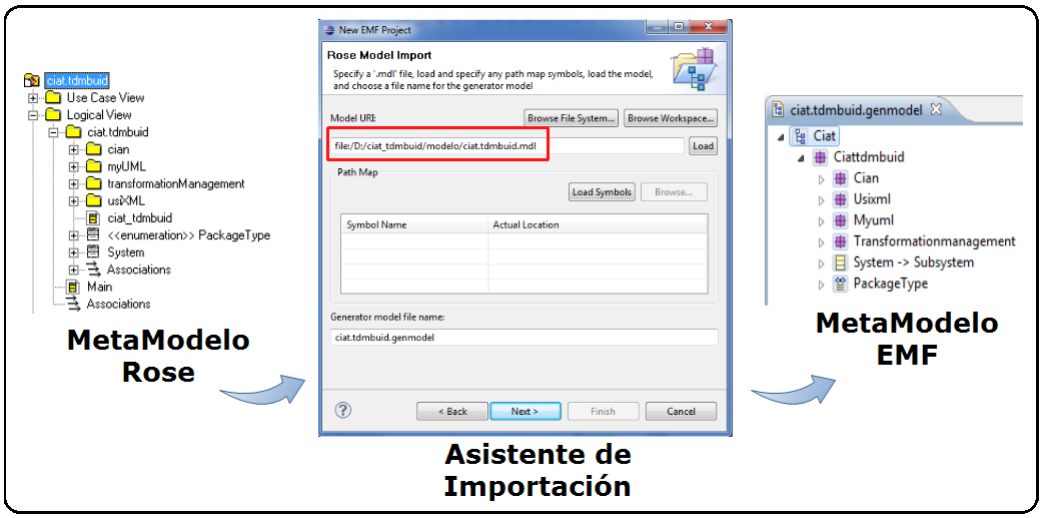

Figura 3.2: Importación de Metamodelo. Presentación propia del Autor

Una vez se ha importado el metamodelo, el resultado de este paso es un modelo con todas las características requeridas por *Eclipse Modeling Framework – EMF*

3.1.1.3. *Java Annotations*

EMF Ecore soporta la construcción de modelos basados en interfaces Java. Esencialmente, las interfaces de Java, incluidos sus métodos, se deberán marcar con una anotación *'@model'* con el fin de ser identificados como elementos del modelo. Además, la anotación *'@model'* puede ser seguida de propiedades para proporcionar información adicional y generación de código directo [18].

3.1.1.4. *XML Schema Definition (XSD)*

XML Schema es un Consorcio World Wide Web (W3C) estándar para la definición de documentos XML [9]. EMF soporta la importación y la transformación de un archivo *XSD* a un modelo Ecore mediante la asignación de elementos *XSD* a los correspondientes elementos Ecore [15].

3.1.1.5. XMI Serialization

(XML Metadata Interchange – XMI) EMF utiliza XMI como un estándar y formato de serialización por defecto de los modelos basados en Ecore, por lo que con un editor de texto, los archivos XMI se pueden crear para representar modelos Ecore. Proporcionar XMI como estándar subyacente para grabar modelos Ecore permite la interoperabilidad y unifica los métodos anteriores: Java, UML y XML Schema [15].

3.2. *Eclipse Graphical Modeling Framework – GMF*

El proyecto *Eclipse Graphical Modeling Framework – GMF*[82] proporciona un componente generativo de herramientas basado en modelos y porporciona también una infraestructura en tiempo de ejecución para desarrollar editores de diagramas para dominios específicos basados en una secuencia de creación de modelos y su respectiva transformación [82, 36]. El objetivo de *GMF* es simplificar el desarrollo de la edición gráfica (con *Eclipse Framework (GEF)* [80] mediante el aprovechamiento de los editores de diagramas de modelos *EMF*. *GMF* genera un editor gráfico completamente funcional (en tiempo de ejecución, basándose en el *GMF-Runtime*) a partir de los siguientes modelos [36]: i) *modelo de domino* 3.1.1; ii) *modelo de definición gráfica* 3.4; iii) *modelo de definición de herramienta* 3.5 y, iv) *modelo de mapeo* 3.6. Más precisamente, GMF utiliza el modelo Ecore EMF para capturar la sintaxis abstracta de un dominio aprovechando otro recurso, como GEF, para desarrollar modelos que representan la forma generativa de sintaxis concreta de ese dominio [18]. *Eclipse Graphical Modeling Framework – GMF* proporciona sintaxis gráfica concreta [28] al metamodelo generado a partir de *EMF*. A partir de esto, GMF permite obtener un modelo de interrelación entre los elementos de la sintaxis abstracta del dominio y los elementos de la sintaxis concreta, representados en los modelos de definición gráfica y de definición de herramienta, denominado modelo de mapeo.

Seehusen[74] expone que en el nivel de la configuración, el modelo de configuración consta de 3 archivos:

1. 1 archivo que se utiliza para especificar las formas de las construcciones gráficas del lenguaje, este es el archivo *.gmfgraph*.

2. 1 archivo que especifíca las construcciones del lenguaje que deben aparecer en la paleta de herramientas del editor del lenguaje, este es el archivo *.gmftool*.

3. 1 archivo *.gmfmap* que correlaciona (mapea) los elementos de la paleta de herramientas (definidos en *.gmftool*) a las construcciones del lenguaje gráfico (definidos en *.gmfgraph*) y a los elementos del metamodelo.

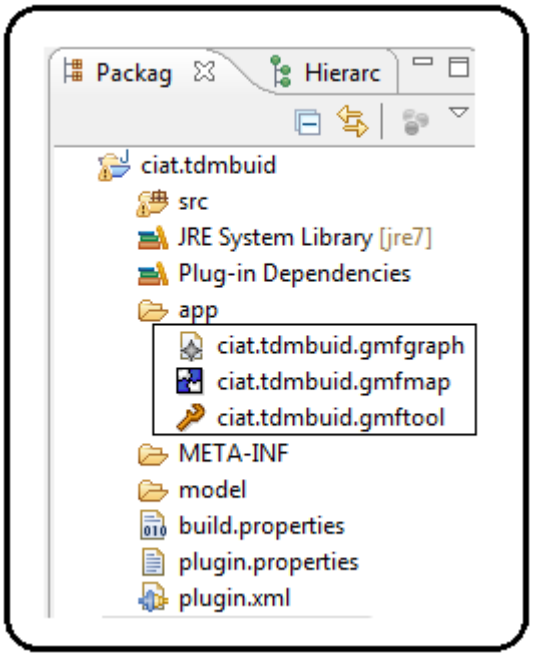

Figura 3.3: Estructura GMF para ciat.tdmbuid

La figura 3.3 muestra los tres archivos definidos para la estructura del proyecto *ciat.tdbmbuid* según [74].

3.3. *Graphical Editing Framework – GEF*

Graphical Editing Framework – GEF[75] permite desarrollar fácilmente representaciones gráficas de los modelos existentes. Es posible desarrollar editores gráficos de ricas características usando *GEF* [54]. *GEF* es un framework de código abierto dedicado a proveer un ambiente rico y consistente de edición gráfica para aplicaciones en la plataforma Eclipse[75]. Las posibilidades de edición con *GEF* permiten crear editores gráficos para casi todos los modelos. Con estos editores, es posible hacer modificaciones sencillas al modelo, como, por ejemplo, cambiar propiedades de los elementos u operaciones complejas o como cambiar la estructura del modelo de diferentes maneras al mismo tiempo. Todas estas modificaciones en el modelo se puede manejar en un editor gráfico con funciones muy comunes como: arrastrar y soltar, copiar y pegar, y las acciones invocadas desde los menús o barras de herramientas [54]. El proyecto *GEF* se compone de tres marcos principales: *Draw2D*, *Zesty* y *GEF*. *Draw2D*[1] es un marco de dibujo ligero para mostrar información gráfica sobre un lienzo *SWT – Standard Widget Toolkit*[2][3] pero no proporciona un comportamiento interactivo. *Zest*[4] está construido por encima de *Draw2D*, proporcionando una interfaz *JFace* para que sea fácil unirse a un modelo de Java con un diagrama Draw2D. *GEF* también se construye en la parte superior de Draw2D, proporcionando una API muy rica para

[1] `http://www.eclipse.org/gef/draw2d/index.php`
[2] `http://www.eclipse.org/swt/`
[3] `http://es.wikipedia.org/wiki/SWT`
[4] `http://www.eclipse.org/gef/zest/index.php`

producir diagramas interactivos con características avanzadas, incluyendo una gama de colores, soporte para arrastrar y soltar, una pila de comandos para deshacer y rehacer comandos, soporte para la impresión y otras características adicionales[70]. *GEF* es un framework *MVC*[5] interactivo construído en una capa superior de *Draw2D*[76].

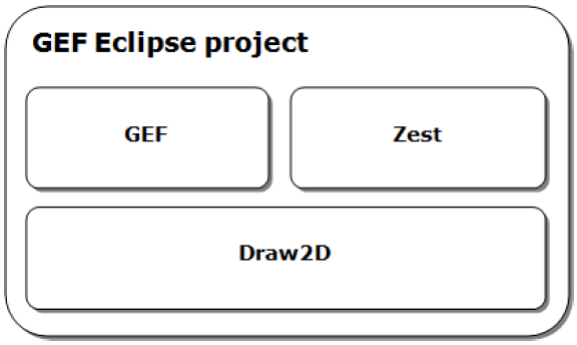

Figura 3.4: Tomado de [70]

Una representación de lo anterior puede verse en la figura 3.4, en la cual la base es *Draw2D* y como capas superiores *Zest* y *GEF*. *GEF* proporciona infraestructura para desarrollar el componente *Controller* de aplicaciones que sigan el patrón arquitectónico Model-View–Controller[8]. Aunque *GEF* no está ligado a ninguna librería, la manera más natural es trabajar con *Draw2D* en la parte *"View"* y *EMF* en la parte *"Model"*. En el patrón *MVC (Modelo–Vista–Controlador)*, el *Controlador* es a menudo la única conexión entre la *Vista* y el *Modelo*. El controlador es responsable de mantener la vista y de interpretar los eventos de interfaz de usuario y convertirlas en operaciones en el modelo. Una representación visual se puede obtener de la figura 3.5.

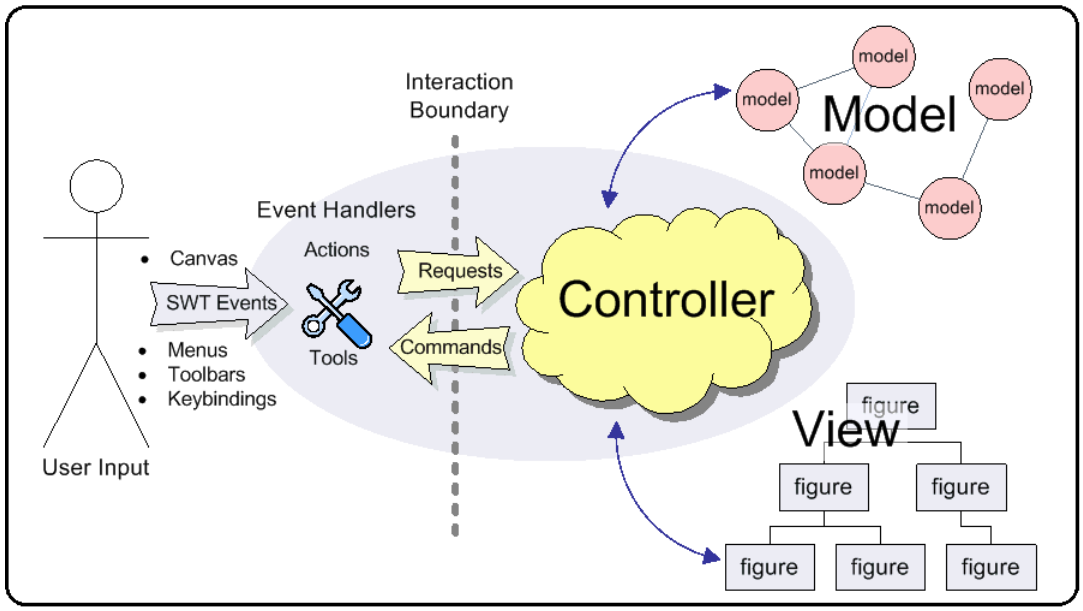

Figura 3.5: Patrón MVC. Tomado de [76]

[5]Pattern-Oriented Software Architecture: A system of Patters, en [8]

3.4. *gmfgraph*

Para Gronback [28], el modelo de definición gráfica posee tres "capas": en primer lugar, las figuras definen las representaciones visuales de los elementos del diagrama. En segundo lugar, los descriptores de las figuras y el acceso de referencia de las figuras en la primera se usan en la siguiente. En tercer lugar, los elementos de diagrama se definen para su uso en el modelo de mapeo y pueden contener información específica de la distribución del elemento. Estas tres capas proporcionan flexibilidad en el modelo de definición gráfica *gmfgraph* porque se permite la reutilización. Las figuras pueden ser reutilizadas para construir otras figuras, los descriptores de la figura puede ser usados por múltiples elementos de diagrama, y el mismo elemento del diagrama se puede utilizar en varios mapeos. Para Kolovos[37], el modelo gráfico *(gmfgraph)* especifíca los elementos gráficos (figuras, conexiones, etiquetas) que participan en el editor. Según Fuhrmann[21] un modelo gráfico es un modelo que puede tener una representación gráfica, por ejemplo, el diagrama de clases de UML [58, 71]. Para Ruscio[72] el modelo de definición gráfica contiene parte de la sintaxis concreta, este identifica elementos gráficos que pueden, de hecho, ser reutilizados para diferentes editores. Este modelo especifíca una galería de figuras incluyendo formas, etiquetas, líneas, etc., y los elementos de canvas para definir nodos, conexiones, compartimentos, y etiquetas de diagrama. Guerra, en [29], indica que el modelo gráfico *gmfgraph* contiene la especificación de la sintaxis del lenguaje gráfico, y también manifiesta que solo un editor en árbol está disponible para la especificación de las figuras de la sintaxis concreta, lo cual es engorroso y propenso a errores.

Para Molina, en [50], el modelo de definición gráfica define esa parte gráfica que se manejará en el editor (edición y visualización). El archivo *.gmfgraph* contiene, para el caso de la propuesta [26], la definición de la sintaxis concreta del CTT [63] y del diagrama de clases de UML [58, 71]. La paleta gráfica contiene los elementos: *compartment; rectangle; icon; label.*

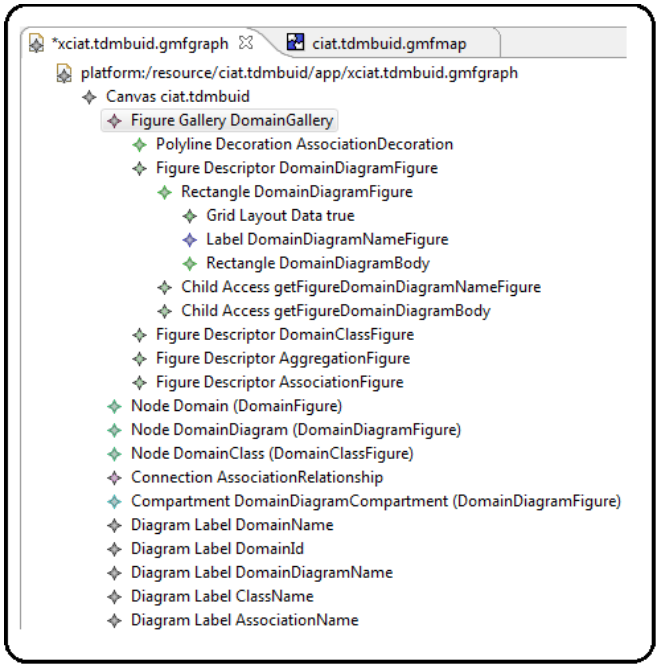

Figura 3.6: Modelo de Definición Gráfica para ciat.tdmbuid

Giraldo[26], afirma que la especificación de la sintaxis concreta debe ser uno de los pasos preliminares en la especificación de cualquier lenguaje para definir sus particularidades y el *mapping*

con respecto de la sintaxis abstracta. Esta especificación tiene como partida el análisis visual de cómo se presentarán al usuario desarrollador los elementos de modelado del lenguaje. Este modelo define la estructura de componentes gráficos usados para representar los conceptos.

En *GMF – Graphical Modeling Framework* [28] el *graph metamodel* describe la apariencia (aspecto) del editor gráfico generado por el modelo. Las metaclases *Canvas, Figure, Node, DiagramLabel, Connection* y *Compartment* son usadas para representar componentes del editor gráfico que se va a generar [69].

Para Köllner, en [36], el modelo de definición gráfica comprende un conjunto de primitivas de dibujo, tales como rectángulo, polilínea o Label (etiqueta). La representación visual de una instancia del modelo de dominio está completamente compuesta por los elementos de la definición gráfica.

3.4.1. *Canvas*

El modelo de definición gráfica consta de un elemento *Canvas*. Esta es la raíz de definición gráfica de todos los elementos. Este elemento raíz tiene una propiedad de nombre y referencias de contención a una o más *Figure Galleries, Nodes, Connections,* y *Labels*. En la definición gráfica de *GMF* [28] una figura se define dentro de una *Figure Gallery* que luego es referenciada por *Node, Connections, Compartment* y elementos *DiagramLabel*, a través de *Figure Descriptors* y *Accessors*. Estos elementos de diagrama son "hermanos" de los elementos de *Figure Gallery*, aunque se pueden referenciar definiciones gráficas de otros modelos de definición gráfica.

Los elementos de la superficie del diagrama se dan a partir de su definición en el *Canvas*. Un *Canvas* contiene un número de elementos *Figure Gallery*, y estos contienen elementos *Figure Descriptors* que definen las figuras reales. Un modelo de definición gráfica puede tener uno o más elementos *Figure Gallery*. El segundo componente representa los elementos *Canvas* donde *Nodes, Connections, Diagram Labels* y *Compartments* están definidos. Los elementos del *Canvas* actúan como referencias a las figuras definidas en los *Figure Descriptors* y se usan en el modelo de mapeo para asignar figuras correspondientes a elementos del modelo de dominio. En consecuencia, los *Nodes* referencian figuras, *Connections* referencian enlaces (links), *Diagram Labels* referencian etiquetas (labels) y *Compartments* referencian figuras anidadas [18].

3.4.2. *Figure Gallery*

Contiene figuras y un número de elementos como descriptores de figuras *(Figure Descriptor)*, y una propiedad opcional de implementación de paquete *(Implementation Bundle property)*. Un *Figure Descriptor* tiene un nombre y describe una figura mediante una referencia a la figura y a sus descriptores de acceso, si los hay. Los *Figure Descriptor* permiten galerías de figuras para ser reutilizadas con figuras anidadas dentro de otras figuras sin una referencia explícita a su padre. Un *Figure Descriptor* contiene una sola *Figure*, mientras que un *Figure Gallery* contiene un número de elementos *Figure*. Un *Figure Descriptor* puede usar una *Figure Reference* como su figura, pero estas no son permitidas como elementos contenidos de una *Figure Gallery* [28].

3.4.3. *Figure Descriptor*

Mantiene una referencia de contención a una figura que puede ser una figura real o una referencia de la figura. También puede contener un número de elementos *ChildAccess*, que, a su vez, hacen referencia a una figura [28]. Las figuras pueden ser reusadas en la misma definición gráfica o

desde otro modelo de definición gráfica.

3.4.4. *Nodes*

Una figura puede anidarse dentro de otras figuras, mientras que un nodo creado como un elemento del diagrama que hace referencia a una figura representa lo que se crea en el diagrama [28]. Un nodo representa una figura en la *Figure Gallery* mediante su propiedad *Figure*.

3.4.5. *Diagram Labels*

Extiende de *Node* en el modelo de definición gráfica y, por lo tanto, tiene todas sus propiedades [28].

3.4.6. *Connections*

Son elementos de diagrama simples que no contienen propiedades adicionales [28]. Para definir un *connection* es posible hacerlo adicionando un nombre, un *child* a la figura de referencia.

3.4.7. *Compartments*

Un compartimento se define como un elemento de diagrama, aunque siempre está contenido dentro de otro elemento, a diferencia de las conexiones, los nodos y las etiquetas.

3.5. *gmftool*

Köllner en [36] cita que el modelo de definición de herramientas permite al usuario definir la información como paletas de herramientas y menús para el editor gráfico. El modelo de definición *Tooling*, para Eloumri[18], es usado para definir una *Palette* que consiste de *Creation Tools* (herramientas de creación) que son usadas para seleccionar y crear figuras en la superficie del diagrama. El modelo de definición *Tooling* tiene un elemento raíz llamado *Tool Registry*[28]. El elemento *Tool Registry* consiste en el elemento *Palette* y otros elementos tales como *Toolbar* y *Menu*. Sobre el elemento *Palette* pueden ser adicionados otros elementos: *Tool Groups, Separators* y *Creation Tool* para crear nodos y enlaces (links), además de *Icon Images, Standard Tool* y *Generic Tool* [28].

Seehusen en [74] expresa que el archivo *.gmftool* especifíca las construcciones del lenguaje que deben aparecer en la paleta de herramientas del editor del lenguaje. Este se encuentra contenido en el nivel 1 de MOF[61]. Ruscio en [72] manifiesta que la definición del *tooling* tiene que ver con la funcionalidad del editor, proporcionando otra parte de la sintaxis concreta contribuyendo para los elementos *Palette, Toolbar* y *Menu*; igualmente para facilitar la gestión de los contenidos de diagramas, es decir, poder adicionar diferentes elementos gráficos en el diagrama [50]. Para Köllner[36] el editor de definición de herramientas debe listar las clases que se definen en el modelo de dominio. El modelo de herramientas, para Kolovos en [38][37], especifíca qué herramientas estarán disponibles en editor de *Palette* (por ejemplo, una herramienta que crea una nueva instancia de una clase particular).

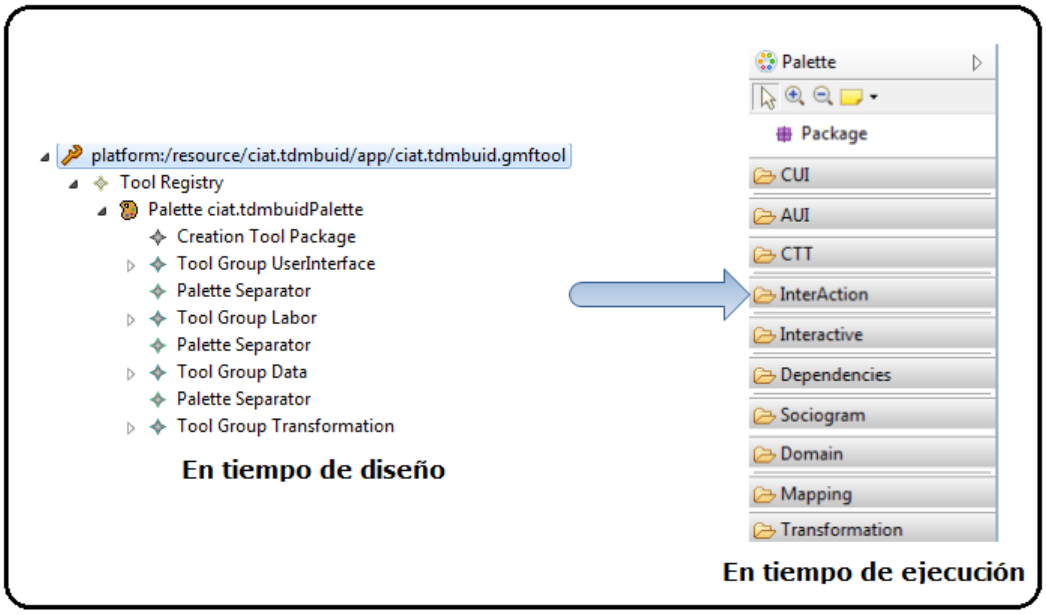

Figura 3.7: Definición de Herramienta para *ciat.tdmbuid*

3.6. *gmfmap*

El archivo de mapeo construido, en EMF, puede definirse como el nexo entre todos los otros modelos de definición: mapea *EObjects* con elementos gráficos y herramientas y es el archivo que define las posibles relaciones en el editor *(links y containment.)* [81]. Para Vanderdonckt, en [84], *Mapping Model* es un modelo que contiene una serie de relaciones establecidas (mapeadas) entre modelos y/o elementos de modelos.

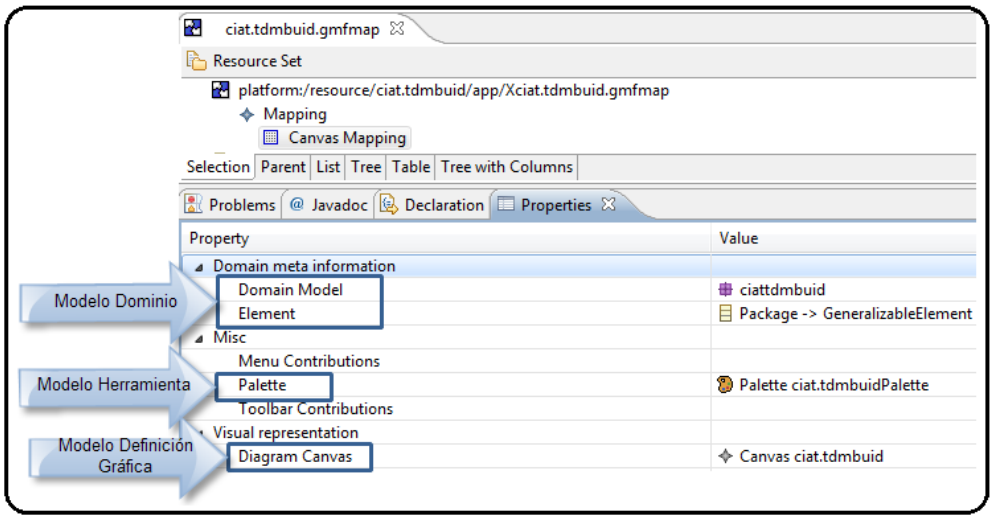

Figura 3.8: Modelo de Mapeo para ciat.tdmbuid

El modelo de mapeo *(gmfmap)* permite enlazar tres modelos: dominio, definición gráfica y definición de herramientas. Ver figura 3.8. Este es un modelo clave para el desarrollo en *GMF* y

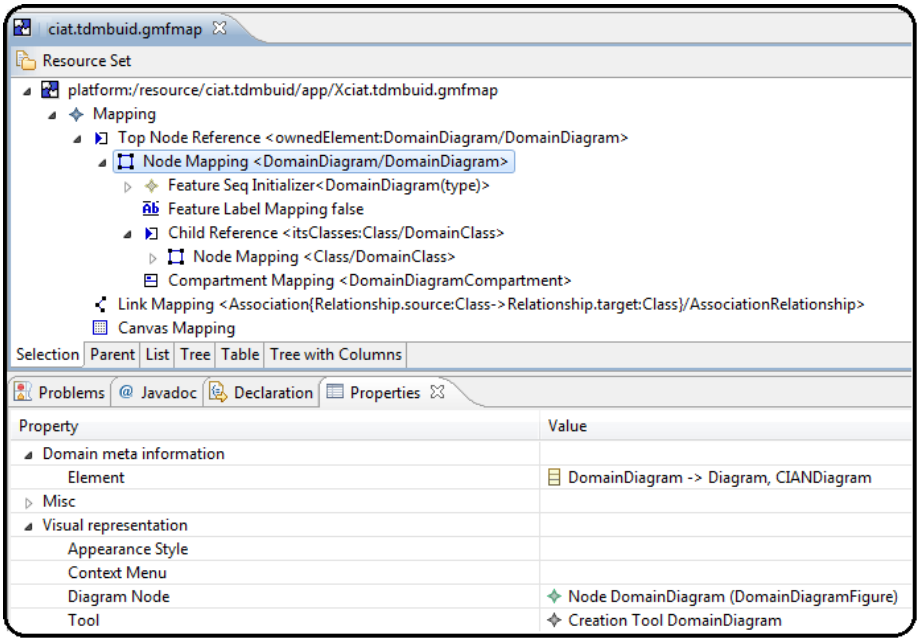

Figura 3.9: Ejemplo mapeo para diagrama de Dominio

se puede utilizar como entrada a una etapa de transformación que producirá el modelo final: modelo de generación *genmodel*. En este modelo de mapeo puede mapearse el modelo visual gráfico a la lógica de negocio *(modelo de dominio)*.

Para Köllner[36], el modelo de mapeo enlaza tres modelos juntos (modelo de dominio; modelo de definición gráfica y modelo de definición de herramienta). En él se especifíca qué elementos geométricos utilizar para un componente específico en el metamodelo, así como qué componente del metamodelo debería ser instanciado cuando el usuario selecciona un elemento específico en la barra de herramientas.

Como puede observarse en la figura 3.9 el *Node Mapping* CTTDiagram se representa en el *modelo* por el elemento *CTTDiagram*; a su vez este *Node* se representa gráficamente mediante el *Diagram Node* Node CTTDiagram que corresponde a uno de los elementos definidos gráficamente en el archivo *.gmfgrahp*; este elemento *Node CTTDiagram* podrá crearse en el editor mediante la herramienta *Creation Tool CTTDiagram*.

Según Molina[50] el modelo de mapeo se define mediante el archivo *.gmfmap*. En este archivo cada una de las relaciones de contención de los componentes gráficos se definen. Se trata de relaciones en el que un componente gráfico puede incluir algunos elementos gráficos demás. Es también, en este archivo, que los elementos gráficos están relacionados entre sí con la barra de heramientas. El conjunto de restricciones semánticas definidas para los modelos se incluye en el archivo *.gmfmap*. La evaluación de estas restricciones se llevará a cabo cada vez que una instancia de la clase específica (es decir, un elemento gráfico) se coloca sobre el canvas, o cuando una interacción con los elementos gráficos se lleva a cabo. También para Molina, en [50], en el archivo *.gmfmap* se especifíca la correspondencia (mapeo) entre lo abstracto y la sintaxis concreta de las notaciones (CTT y UML Class Diagram). El modelo de mapeo interconecta a los diferentes elementos modelados por el usuario y construido por medio de los editores. En este mismo archivo *.gmfmap* se incluye el conjunto de Constraints de Semántica. En el archivo *.gmfmap* se definen los elementos gráficos que se manejarán en el editor (edición y visualización). En este

archivo se define la sintaxis concreta del CTT y UML Class Diagram. La paleta gráfica contiene los elementos: *compartment; rectangle; icon; label.*

El modelo *gmfmap* indica qué acción tomar cuando se selecciona una herramienta, lo que las clases van a crear y lo que aparenta hacer cuando las clases se adicionan al diagrama [68]. El mapeo define, también, cómo recorrer la estructura del objeto del diagrama y mantenerlo actualizado cuando el objeto cambia de estructura.

Para Gronback[28], el modelo de mapeo es el corazón de los modelos en *GMF* y representa en sí mismo un diagrama. Un modelo de mapeo se transforma en uno o más modelos generadores que conducen a plantillas para la generación de código.

La raíz de este modelo, en [28], es el elemento *'Mapping'* y este puede tomar una serie de elementos como:

3.6.1. *Canvas Mapping*

Este elemento es necesario y representa el lienzo *canvas* del diagrama. El asistente de mapeo rellena este elemento *Domain Model (EPackage)* seleccionado, su *Element (EClass)* raíz, el *Canvas Diagram* de un modelo de definición gráfica y la *Palette* del modelo de definición de herramientas.

3.6.2. *Top Node Reference*

Representa los elementos que se crean en la superficie del diagrama. Este elemento contiene una referencia hija (*Children*) a un único *Nodo Mapping* (raíz). Los elementos deben estar contenidos en alguna parte en el modelo de dominio correspondiente cuando se crean instancias como elementos de diagrama. La propiedad *Containment Feature* especifíca dónde agregar estos nuevos objetos y, por defecto, dónde recuperarlos. En [81] se aprecia que la característica de Contención de un *Node Reference* especifíca dónde ubicar un hijo recién creado, mientras que las características de los hijos *(Children Feature)* dónde tomar los hijos con el fin de desplegarlos. Es bastante frecuente que estos dos *Node Reference* y *Children Feature* sean lo mismo, por lo tanto, las características por defecto de los hijos deben tener el valor de la característica de contención.

3.6.3. *Node Mapping*

Cada *Top Node Reference* y *Child Reference* contiene un único *Node Mapping* [28]. Un elemento *Node Mapping* une a un *Diagram Node*, *Tool* y elementos del *Domain Model*. El elemento de dominio es el *EClass* del modelo de dominio que es el nodo de mapeo que representa. Del mismo modo, el *Diagram Node* es el nodo de la definición gráfica que se usa para visualizar la sintaxis concreta (gráfica) para este nodo. El *Tool* es la herramienta de creación de la definición de herramientas que se utiliza para crear el nodo desde la paleta. Para Krogstie en [31] el *Node Mapping* define un mapeo de elemento de dominio para graficar un elemento *Node*. Este mapeo contiene submapeos para el *Content* y *Children*, por ejemplo: etiquetas y compartimientos. El *Node Mapping* mapea un elemento a su controlador. Contiene descripción anidada de mapeos de sus funciones *Features*.

3.6.4. *Link Mapping*

Para mapear un *Connection* desde la definición gráfica de un modelo de dominio y una herramienta de creación de paletas, se usa el elemento *Link Mapping.* Cuatro casos de uso principales

son soportados por el mapeo de enlaces: enlaces de diseño, referencias de elementos de dominio, enlaces que representan elementos del dominio de clases y enlaces de nodo *phantom*. Es posible crear links que no mapean elementos del dominio: los llamados links de diseño. En este caso, sólo se debe seleccionar una herramienta de paleta y un link de diagrama dejando todas las propiedades para el dominio en blanco. Los enlaces *(links)* creados de esta manera se pueden realizar entre todos los nodos de nivel superior en el diagrama, aunque no representan la información del modelo de dominio.

Los links crean conexiones entre los elementos del diagrama y pueden representar distintos tipos de relaciones de referencias encontradas en un modelo de dominio. Las referencias regulares son el uso más típico de los links en un diagrama y son los más sencillos de implementar. Los links también pueden representar completamente elementos de dominio *(EClass)*, con referencias de origen y destino. Las referencias de contención pueden ser representadas mediante links a nodos de nivel superior en un diagrama mediante el concepto de *'nodo fantasma'*. Para Kolovos[38] los nodos Phantom son particularmente útiles con el fin de visualizar referencias de contención que utilizan enlaces en lugar de contención espacial.

3.6.5. *Feature Label Mapping*

Permite asignar una característica de un elemento de dominio a su controlador de etiqueta [31].

3.7. *ATL – Atlas Transformation Language*

ATL (Atlas Transformation Language) es un lenguaje de transformación de modelos que especifica un meta–modelo y una sintaxis textual concreta. Su enfoque es un hibrido entre declarativo e imperativo. ATL permite la transformación utilizando múltiples entradas y múltiples salidas y utiliza un depurador de transformaciones. Surgió como respuesta del grupo de investigación ATLAS INRIA & LINA a la propuesta realizada por OMG con QVT.

ATL es un lenguaje propiamente dicho, con tres unidades gramaticales básicas[65]:

- *Module* (Módulo): corresponde a una transformación de modelo a modelo. Permite la definición de transformaciones. Contiene las declaraciones iniciales. Está constituido por 4 partes:
  - *Header* (Cabecera): define el nombre del módulo y los nombres de los meta-modelos de entrada y salida.
  - *Import* (Importación): que permite involucrar librerías ATL.
  - *Helper* (Ayudante): el equivalente ATL a un método Java. Puede ser utilizado para definir las variables y funciones (globales). Es una estructura intermedia, dentro de las transformaciones, que facilita la navegación, la modularización y el reuso. Permite definir operaciones.
  - *Rule* (Regla): define la forma en el modelo destino que será generada a partir del modelo origen. Existe también una construcción llamada *called rule*, y que representa un *procedure*. Esta puede contener argumentos y puede ser invocada por su nombre. La aplicación de la regla se realiza de forma no determinística, no se ha provisto ninguna construcción o claúsula que permita aplicar en forma condicional las reglas. Cabe mencionar que la invocación de *called rules* es determinística. Esta invocación, junto con la utilización de parámetros permite soportar recursividad.

*AUI_CUI.atl

```
module AUI_CUI;
create OUT : ciattdmbuid from IN : ciattdmbuid;

helper def : system           : ciattdmbuid!"ciattdmbuid::System"       = OclUndefined;
helper def : diagram          : ciattdmbuid!Diagram     = OclUndefined;
helper def : package          : ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"       = OclUndefined;
helper def : TaskPackage      : ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"       = OclUndefined;
helper def : DomainPackage    : ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"       = OclUndefined;
helper def : MappingPackage   : ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"       = OclUndefined;
helper def : AUIPackage       : ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"       = OclUndefined;
```

Figura 3.10: Definición de Module y Helpers para la transformación

AUI_CUI.atl

```
rule TaskPackage{
    from  i: ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package" (i.name = 'Task Model')
    to      o: ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"(name<-i.name,ownedElement<-i.ownedElement)
    do{
            thisModule.system.ownedElement<-thisModule.system.ownedElement->append(o);
            thisModule.TaskPackage<-o;
            for (t in i.ownedElement){
                thisModule.TaskDiagram(t);
            }

    }
}
rule DomainPackage{
    from  i: ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package" (i.name = 'Domain Model')
    to      o: ciattdmbuid!"ciattdmbuid::myuml::modelmanagement::Package"(name<-i.name,ownedElement<-i.ownedElement)
    do{
        thisModule.system.ownedElement<-thisModule.system.ownedElement->append(o);
        thisModule.DomainPackage<-o;
        for (t in i.ownedElement){
            thisModule.DomainDiagram(t);
        }
    }
}
```

Figura 3.11: Plugin ATL para la implementación de las transformaciones de modelo a modelo.

- *Query* (Consulta): puede ser visto como una operación que computa un valor primitivo desde un conjunto de modelos de origen. Para obtener elementos del modelos que cumplen con ciertas propiedades o restricciones.
- *Library* (Librería): permite la definición de un conjunto de helpers que pueden ser llamados desde una unidad gramatical ATL diferente. Funciones auxiliares reutilizables.

La figura 3.11 muestra el plugin de Eclipse utilizado para llevar a cabo las transformaciones utilizadas en esta definición de trasformación.

En cuanto a las reglas de transformación, ATL define un dominio *(metamodelo)* para el modelo origen *(source)* y otro dominio para el modelo destino *(target)*. Source y target pueden tener iguales o diferentes dominios (aunque sean iguales, ambos deben ser claramente identificados). Si bien las transformaciones son unidireccionales, ATL permite la definición de transformaciones bidireccionales como la implementación de dos transformaciones, una para cada dirección.

Capítulo 4

Desarrollo y aporte de este trabajo de grado

El desarrollo presentado en este trabajo de grado se centra en la construcción de la aplicación que soporta el trabajo realizado por Giraldo[26] junto con nuevas funcionalidades en notación y en sintaxis abstracta. Esta nueva aplicación se centra en la generación automática de interfaces gráficas de usuario a partir de las etapas y modelos expuestos por Limbourg[42], Giraldo[26] y Molina[50]. Esta aplicación se ha denominado *CIAT.TDMBUID – Collaborative Interaction Application Tool.Task & Data Model Based User Interface Development.*

Para el desarrollo de esta aplicación se consideró, como producto final, la generación del editor de modelos que permite la generación/construcción de los siguientes diagramas:

- Diagrama de tareas CTT y sus operadores temporales (relaciones);
- Diagrama de Dominio. Modelo de clases, atributos, operaciones y relaciones entre las clases: asociación, generalización, agregación y composición;
- Modelo de Trazabilidad: establecimiento de relaciones entre las tareas definidas en el diagrama CTT y las clases y elementos de las clases definidos en el diagrama de dominio.
- Sociograma;
- Diagrama de Inter-Acción (Inter-Action): representa todo el modelo de procesos de negocio a partir de la definición de las tareas: New Individual; Simple Individual; Composite Individual; New Cooperative; Simple Cooperative y Composite Cooperative.
- Diagrama Interactive (InterAction): representando, en la tarea InterAction, las tareas a realizar en el sistema.
- Diagrama AUI (Abstract User Interface): propone una serie de componentes que están centrados en las tareas que el usuario realiza, tanto en el negocio como en el sistema, y no tanto en la información manipulada por el mismo usuario.
- Diagrama CUI (Concrete User Interface): Modelo de interfaz tangible, por parte del usuario, y generado a partir de las reglas de transformación definidas para la obtención de los componentes gráficos de interfaz.
- Diagrama de Transformación: Propuesta de este trabajo de grado. Se trata de la gestión de las transformaciones a partir de la generación del diagrama y ejecución de las transformaciones elegidas por el usuario. En este caso, transformaciones de Limbourg[42] y Luyten[46].

Usando los anteriores diagramas el usuario de este editor de modelado podrá disponer de un modelo completo para construir, en términos de tareas, datos y transformaciones, interfaces gráficas de usuario. Esta herramienta usa para la parte visual diagramas, que es lo que las personas utilizan y comprenden. De manera que el usuario construye sus diagramas con la respectiva notación abstrayéndose de la representación interna. Sin embargo, esta aplicación, para su proceso interno y correspondientes transformaciones usa *'Modelos'*.

El desarrollo de la herramienta *CIAT.TDMBUID*, por medio del proyecto GMF[81], se lleva a cabo mediante los siguientes pasos:

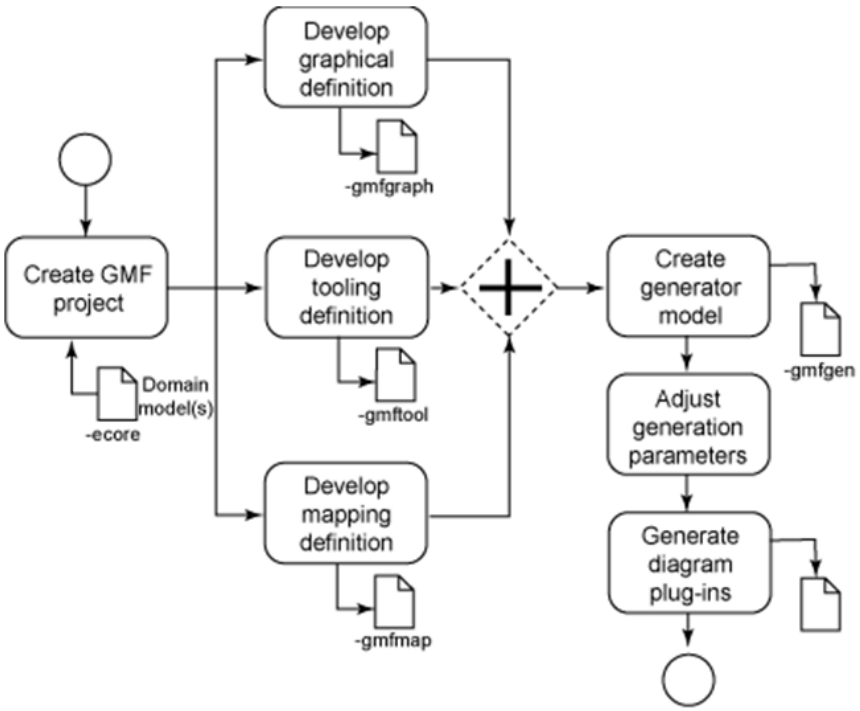

Figura 4.1: Metodología de desarrollo para la construcción de CIAT.TDMBUID. Tomado de [81]

Todos y cada uno de estos pasos se describen a continuación referenciando el aporte de este trabajo de grado.

4.1. Definición del modelo de dominio

De acuerdo a lo planteado en el objetivo general (ver 1.3.1) se han integrado, en este trabajo, a nivel de metamodelado, las notaciones CIAN[49], TD-MBUID[26] y usiXML[83].

Este modelo define la información no gráfica manejada por el editor, es decir, define el dominio que es modelado mediante el editor (por ejemplo, elementos incluidos en Diagrama de tareas CTT; Diagrama de Dominio: clases, atributos, operaciones y relaciones entre las clases; Modelo de Trazabilidad; Sociograma; Diagrama de Inter-Acción; Diagrama Interactive; Diagrama AUI; Diagrama CUI; diagramas de Transformación;). El modelo de dominio dentro de un proyecto GMF[81] es una representación EMF[77] del metamodelo que define la sintaxis abstracta del lenguaje que se desarrolla, es decir, la base de todos los artefactos que se encuentran presentes en la herramienta. Esta sintaxis abstracta actúa como una guía para la comunicación de todos los

artefactos y herramientas que intervienen en el desarrollo de una herramienta de soporte a MDE. La sintaxis define qué elementos de modelado existen en el lenguaje y cómo estos elementos de modelado se construyen en términos de otros. Cuando el lenguaje tiene una sintaxis gráfica, es importante definir la sintaxis en una notación de forma independiente (es decir, para definir la sintaxis abstracta del lenguaje). La sintaxis concreta se define entonces mediante un mapping entre la notación y la sintaxis abstracta[26]. El desarrollo del modelo de dominio dentro de Eclipse se hace mediante un modelo ECORE[79] en EMF[77]. El archivo *ciat.tdmbuid.ecore* contiene todos los modelos que combinan la semántica del lenguaje (sintaxis abstracta de los modelos) y la especificación de los elementos gráficos de la notación (sintaxis concreta de los modelos) [34].

Para iniciar esta integración fue necesario, primero, conocer y estudiar los metamodelos originales referidos por Giraldo[26]. Una vez realizado el reconocimiento de cada uno de los metamodelos, que se encontraban por separado, se procedió a realizar integración de estos metamodelos bajo una sola estructura denominada *CIAT.TDMBUID (Collaborative Interaction Application Tool.Task&Data Model Based User Interface Development).* Giraldo[26], en su propuesta, presentó por separado las tres notaciones que se integran, en este trabajo, bajo un mismo metamodelo: *CHICO* y *CIAT*. El metamodelo *CHICO* contiene las notaciones usiXML y TDMBUID, mientras que el metamodelo *CIAT* posee la notación CIAN[49]. Entonces, el primer objetivo consistió en obtener un metamodelo a partir de los metamodelos presentados, lo que obligó a buscar un mecanismo de entrada y recorrido al metamodelo propuesto para poder replicar la contención y relaciones existentes entre los elementos de modelado propuestos por las notaciones *CIAN*[49], *TD-MBUID*[26] y el lenguaje *usiXML*[41].

La generación del modelo de dominio, en ECORE[79], se dió a partir del procedimiento definido en 3.1.1.2. El metamodelo propuesto *CIAT.TDMBUID* (ver figura 4.2) fue integrado considerando los nombres de las relaciones de contención (composición), igualmente revisando los nombres, porque es a través de las relaciones que se crean los distintos elementos de modelado. Se consideró que, para la construcción del nuevo metamodelo, ningún elemento contenedor permitiera elementos de modelado por fuera de él, establecidos en su nivel de jerarquía. Sin embargo, para efectos de lograr las interfaces abstractas y concretas, fue necesario permitir que ciertos objetos pudieran representarse por fuera de sus contenedores, tal es el caso del objeto concreto *Window* y del objeto abstracto *AC*; de manera que una clase solo podrá ser dibujada dentro de un diagrama de dominio, un tarea de CTT solo podrá ser dibujada dentro de un diagrama CTT. Se presenta el metamodelo *CIAT.TDMBUID* en forma de árbol (ver figura 4.2). De acuerdo a esto, se presenta la reestructuración del metamodelo organizando los conceptos de acuerdo a su contexto: los objetos de *UML* quedan bajo el paquete *UML* y todo lo que corresponda con datos; los objetos abstractos, concretos y de interfaz gráfica quedan bajo el paquete *usiXML*; lo que corresponda con la notación *CIAN* queda bajo el paquete del mismo nombre, y, el aporte de este trabajo de grado queda bajo el paquete *TransformationManagement*.

La clase *'System'* (ver A.1) se define en este nivel (raíz) para efectos de organización de los modelos EMF[81]. De esta forma, cada que se inicia un modelo se obliga al desarrollador (diseñador) a definir primero la información principal del sistema que se diseña. A partir de la clase *'System'* se inicia la instanciación de los demás elementos de modelado que se requieran para completar el sistema.

Si el metamodelo no se construye adecuadamente, así se tenga la sintaxis abstracta y aunque se tenga la sintaxis concreta, si el metamodelo no se encuentra bien definido habrán cosas que no se podrán realizar en la visualización (ejecución de la herramienta sobre el editor *GMF*). Hay una situación, y es que se cuenta con diferentes vistas: de árbol y de diagramas. Por ejemplo, en la vista de árbol se tiene una estructura jerárquica. En esta vista de árbol no se representan directamente diagramas sino referencias a diagramas, es decir, para el caso de las transformaciones (propuesta de este trabajo de grado), se le indica al diagrama de transformación qué elementos

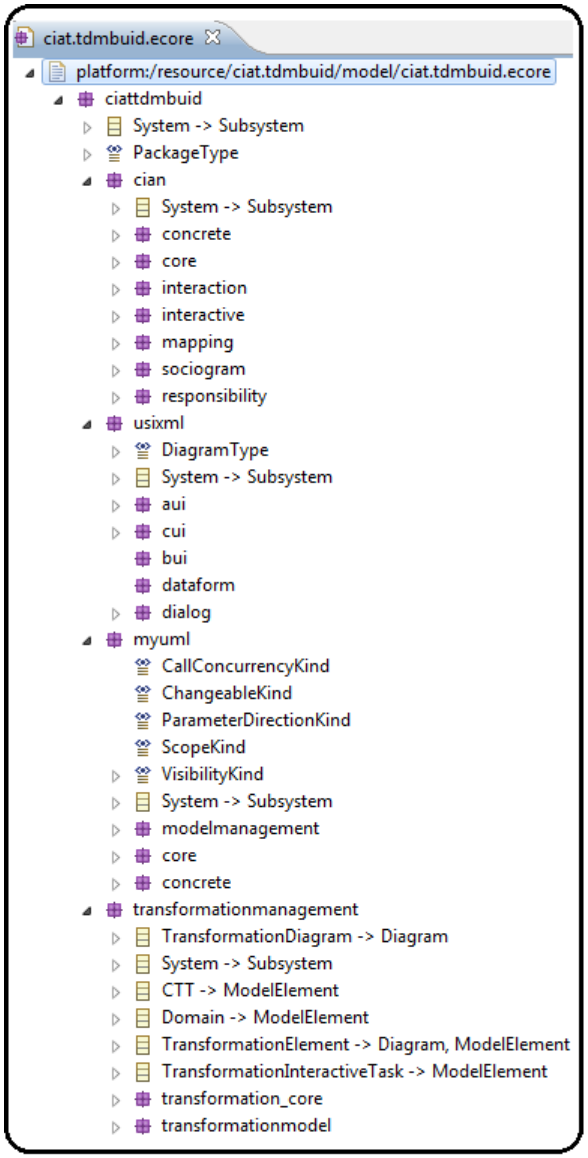

Figura 4.2: Metamodelo CIAT.TDMBUID

enlazar (referenciar) además de indicar la ruta en dónde se debe poner el resultado de la transformación. Para la extensión del metamodelo para la nueva versión de la herramienta, desarrollada con este trabajo de grado, la raíz de la gestión de diagramas es la metaclase *'Package'*. Esta nueva versión del metamodelo se basa en la gestión de modelos de *UML* (*ModelManagement*). De manera qué, basando el canvas sobre la EClass *'Package'*, a nivel de sintaxis concreta, que más adelante se definirá, el canvas puede tomar esa multiforma, para cada diagrama generado. Entonces en esta extensión al metamodelo (integración de notaciones y generación de nueva versión de herramienta) lo que se adicionó al metamodelo fueron los links de mapping sobre las clases. Ya no se trata entonces de metamodelos individuales para cada notación. Lo que cambió en la parte de la sintaxis abstracta fue cómo se conecta de un diagrama a otro con el propósito de guiar la generación automática de interfaces gráficas de usuario. Lo que había planteado, en la propuesta de Giraldo[26] era un modelo de *mapping* con un nivel de contención en el modelo y que relaciona todas las clases de diagramas.

En este nuevo metamodelo se presenta un punto relevante y es *'ModelManagement'* (ver figura A.2). Todos los elementos heredan de esta raíz, de manera que un modelo será un paquete, luego un diagrama será un paquete, todo es un paquete y con la clase *'Package'* es como se provee el *ContaimentFeature* de *OwnedElement*. La relación *OwnedElement* sirve para recorrer todo lo que son paquetes, para tener definidos los elementos de agrupación principalmente paquetes, no de la sintaxis sino de lo que es un modelo, cuántos paquetes dentro de un modelo de una aplicación se contienen, cómo se organizan. Con esto se define toda la parte abstracta: mappings, trazabilidades, diagramas, transformaciones. A través de esta relación *OwnedElement* puede llegarse a cualquier elemento de modelado pero a nivel de metamodelo (sintaxis abstracta), más no a nivel de *mapping* (sintaxis concreta), pues en este último se clasifican los elementos de acuerdo a sus contenedores. A nivel de metamodelo la estructura está abierta para todos los elementos de modelado, pero a nivel de mapping se cierra el modelado junto con expresiones OCL. El metamodelo, de esta propuesta, se maneja en varias capas: nivel abstracto (core, negocio y sistema) y nivel concreto, pero todo dentro del mismo metamodelo. El nivel abstracto contiene las capas *Core* y *ModelManagement*; pero de cara al usuario final (desarrollador) sólo va a ver el nivel concreto (editor de modelado); pero a nivel del metamodelo, la visualización concreta se soporta sobre el nivel concreto. Por ejemplo, en el *Core* está presente el concepto *Task*, pero a nivel del editor se pueden manipular tareas interactivas e interactives, queriendo decir esto que el nivel concreto se soporta sobre el nivel abstracto reusando elementos y no repitiendo la semántica. La relación de entrada para todos los elementos en el metamodelo, se decidió por la relación *OwnedElement*, porque de esta manera se permite la integración de la parte *UML* definida en el trabajo de Giraldo[26] con miras a extender el metamodelo. Teniendo la gestión de diagramas y elementos de modelado en el *Core* del metamodelo, se debieron llevar todos los elementos en el mapping (*CointamentFeature* - sintaxis concreta) por la relación *OwnedElement*. Se puede tener un DomainDiagram, por ejemplo, y también un ClassDiagram, y tener un objeto. El problema es que cuando se tiene una tarea interactiva, por ejemplo, la individual, esta tiene dos compartments y en uno se dibuja un rol y ese rol puede tener unos datos, lo que se quiere decir, es que en uno de los diagramas, ese objeto puede tener cierta apariencia y en el otro diagrama otra apariencia, otra cosa también es que, si la relación no lo permite navegar, desde el tool no se podrá arrastrar el elemento, entonces la decisión, como punto de partida, fue usar la relación *OwnedElement*, reestructurando el metamodelo. Esa es parte del núcleo de este trabajo de investigación, pues se trató de extender y refinar este metamodelo. También se tomó esta decisión de llevar todo por la relación *OwnedElement* porque *CIAT*, junto con *UML* de la propuesta de Giraldo[26], está basado sobre esta relación lo que permitió el reuso de lo existente y facilitó la integración de las notaciones.

Se debió, entonces, realizar un diseño, en donde el *canvas* (ver 3.6.1) pudiera tomar la forma deseada de cualquier cosa. En otras palabras, que en un momento dado (para un diagrama), el canvas pudiera ser *'CTTDiagram'*, o si se desea usar/crear un diagrama de transformación, el canvas, en ese momento pudiera ser *'TransformationDiagram'*. Por eso, el canvas debe ser la metaclase *'Package'*. *'Model'* es el root del metamodelo chico[26]. En la propuesta de Giraldo[26], inicialmente, el modelo se encontraba fraccionado pero a nivel de metamodelo, es decir, que en el *tool* solo se podía hacer una cosa *'Class'*. En esta propuesta inicial existen tres grandes bloques: *CHICO*; *CIAT* y *UML*. En este trabajo de grado se integraron tres variantes: UML (es data –domain) ; CIAN (es labor) y usiXML (interfaces). Toda la parte notacional se maneja en el mapping (.gmfmap). En cuanto al metamodelo, en el nivel abstracto solo se ponen las relaciones básicas, pero lo que es notación concreta se maneja en el mapping (.gmfmap). La modificación, entonces, en el metamodelo consiste en tres conceptos: UML (es data– domain) ; CIAN (es labor) y usiXML (interfaces) junto con la propuesta de este trabajo de grado: manejo de transformaciones. Antes, en el metamodelo (original), se usaba *'Model'* porque no se usaba 'Package', ni la lógica del manejo de diagramas. Luego de toda esta definición del modelo de dominio, debe estar lista la parte concreta para definir cual es el *.gmfmap*, el *.gmfgraph* y el

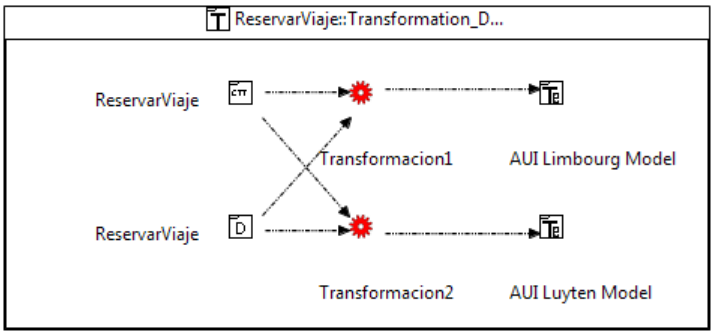

Figura 4.3: Diagrama de Transformaciones para *CIAT.TDMBUID*

.gmftool.

4.2. Modelo de Transformaciones

La intención de este nuevo diagrama (el de transformación) es comunicar que transformación se aborda en el diagrama. Este diagrama contiene, por dentro, referencias a otros tipos de diagramas. En el punto de partida se deben tener la gestión de las transformaciones (Diagrama de Transformaciones); lo segundo es que se ejecuten las transformaciones. El concepto del diagrama de transformación es usar referencias sobre los demás diagramas. Los diagramas de ctt, domain model y trazability model, usados para la transformación abstracta, se referencian en el diagrama de transformación por su representación, es decir, una metaclase que se referencia con su correspondencia (ver A.13).

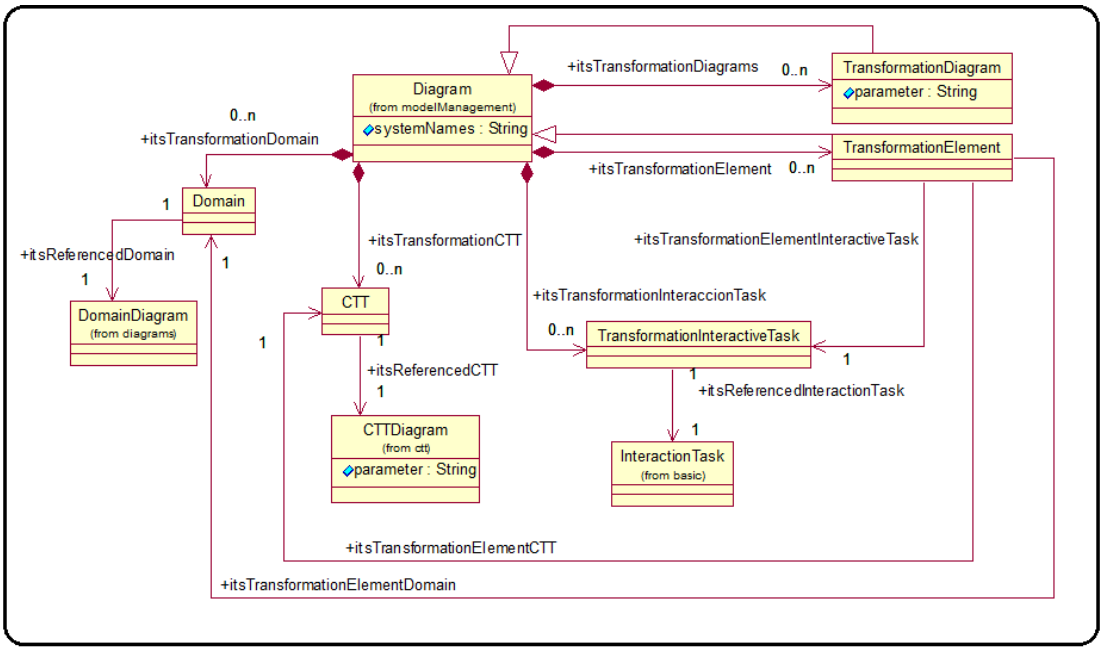

Figura 4.4: Modelo de Transformación.

Una transformación debe tomar la información de un diagrama y la información de otro diagrama y, a partir de eso, hacer el recorrido. Es por eso que el modelo de transformación, en este trabajo de grado, apunta hacia un diagrama y el nodo de transformación, guiado por el tipo de transformación, apunta a otro diagrama o a una tarea de interacción. En el caso de CIAT.TDMBUID se tiene una transformación y lo que le entra a esa transformación es una tarea (o diagrama) y sale una interfaz abstracta (*AUI*). Todos los diagramas heredan de *'Diagram'*. El elemento de transformación apunta a la transformación y apunta a los diagramas (a los elementos que se desean transformar: CTT; Domain e InteraccionTask). Este es el aporte a la herramienta. El diagrama de transformación (transformationElement en el metamodelo), su sintaxis abstracta y su respectiva sintaxis concreta.

Para proponer la ejecución de las transformaciones, lo que se hizo desde la propuesta original (chico)[26] fue revisar cómo se realizaban las transformaciones y actualizar, en términos de las relaciones entre las clases a nivel del metamodelo, todo el recorrido de la sintaxis abstracta (lo propuesto en el nuevo metamodelo).

Una transformación en CIAT.TDMBUID lo que hace es tomar la información de un diagrama y de otro diagrama y a partir de eso hace recorridos buscando correspondencias, de acuerdo a las reglas de transformación planteadas, para concretarlas. Por eso en el metamodelo es una clase que apunta a un diagrama y en la parte concreta una transformación apunta a un elemento(objeto) de transformación (que puede ser una tarea de interacción) por medio de un link de transformación (ver figuras 4.3 y A.13). *TransformationElement*, en el metamodelo, es la representación de algo que puede ser la fuente de una transformación, por eso apunta a un elemento de modelado (ver figura A.11). En la herramienta que se presenta se pueden referenciar diagramas de tareas ctt, diagrama de dominio, diagrama de mapping y la tarea interactive.

Entonces la propuesta es que la transformación tenga como entrada una tarea de interacción (*InteractionTask*) o un diagrama de tareas en conjunto con el diagrama de dominio (datos) y su respectivo modelo de trazabilidad (mapping) y que la salida sea una *AUI*. Esto es lo que hace el modelo de transformación Limbourg[42] para esta herramienta CIAT.TDMBUID.

La generación automática se basa en la definición de distintos componentes. Estos componentes gráficos individuales se obtienen por medio de la cardinalidad de los atributos en el modelo de datos. Se tiene una clase con sus atributos y la cardinalidad de esos atributos. Esta cardinalidad refleja los diferentes componentes gráficos individuales sobre la UI. La UI tiene dos componentes: uno es la presentación y el diálogo (es decir qué se activa y qué se bloquea). El modelo de diálogo es algo de la interfaz que indica cómo dialoga, cómo se presenta, cómo bloquea los componentes. El modelo de diálogo conserva el patrón *Model–View–Controller*. El proceso de transformación para la generación de UI, inicia porque se tiene una tarea principal (*Maintask*) y una vez que se ha identificado, dentro del correspondiente diagrama de Tareas – *CTT* – es como si se tuviera un recorrido para todo el diagrama principal y el recorrido se inicia cuando el diagrama se ha marcado y se sabe cuál es el diagrama de tareas y el diagrama de dominio. El recorrido *'barre'* todas las tareas hijas del diagrama de tareas. Por cada par, padre e hijo, y que tenga una relación de descomposición (que es una interdependencia de tipo *AggregationTransition*) se crea un contenedor que refleja la tarea padre y la tarea hija es un componente individual abstracto. El algoritmo de transformación de *Limbourg*[42] inicia buscando las relaciones de interdependencia de tipo (*AggregationTransition*) si no tiene esta interdependencia es porque no tiene hijos. Por cada tarea que se descomponga en otra(s) tarea(s) se genera un *AC*. Este *AC* es el que contiene las tareas hijas. Y luego se crea el *mapping* (en términos del modelo de trazabilidad). Todo esto lo hace el algoritmo de transformación. El algoritmo para transformar un *AC*, busca la tarea padre, de manera que, por ejemplo, para el diagrama de tareas (CTT_Diagram) debe existir el enlace *'mainTask'*, porque es asumido que las tareas van a estar dentro de un diagrama. Este es el enlace que sirve para la transformación. El algortimo de transformación se sirve del modelo de

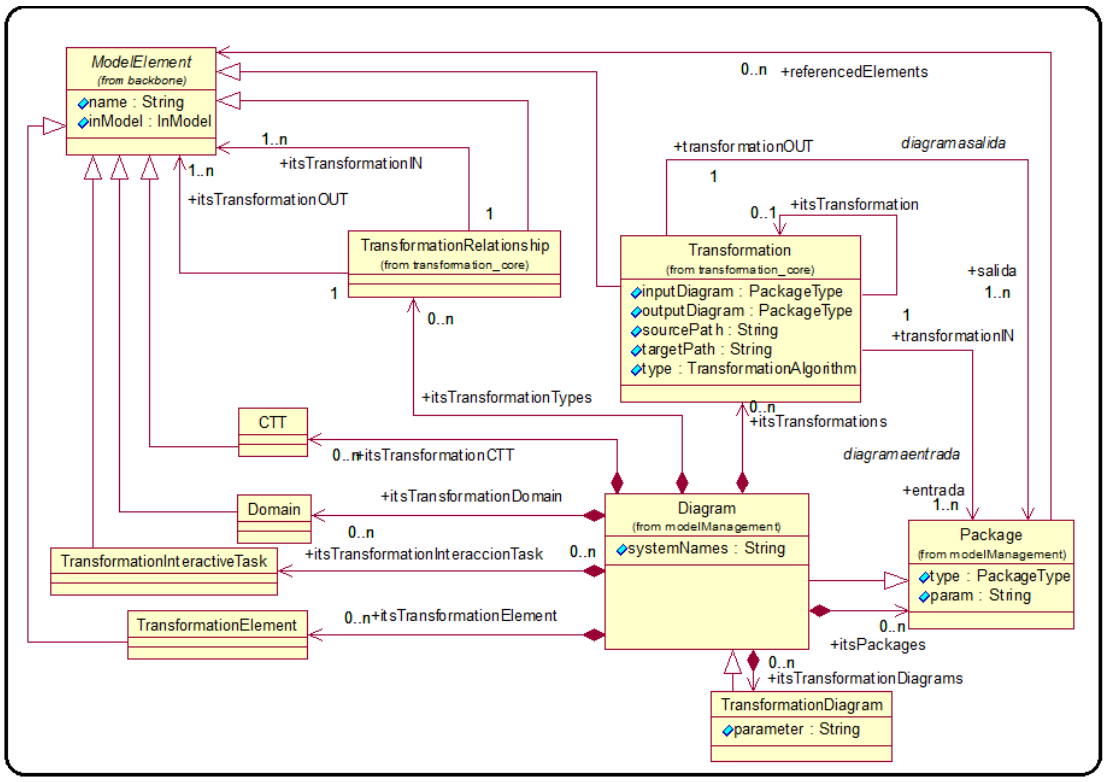

Figura 4.5: Contención de Transformaciones

mapping (a nivel del metamodelo: mapping diagram), que son relaciones entre modelo de tareas y de datos. Para poder soportar las transformaciones es necesario saber en qué nivel está trabajando. A nivel de metamodelo puede que los conceptos sean abstractos pero el metamodelo debe poder diferenciar qué elemento es qué elemento, pues debe estar en capacidad cuando apuntar, por ejemplo, a una tarea (task) o a un dato (class, attribute, operation). *CTT* es de los pocos diagramas que son dependientes del *Layout*, eso es inconveniente, es decir, podría dar lo mismo si se tiene, en notación *UML*, una clase A relacionada a una clase B, da lo mismo que se dibuje una clase B relacionada a A. Pero en un *CTT*, una tarea A relacionada a una tarea B, no es lo mismo tener B a A por cuestión del operador de temporalidad que las une, por ejemplo, una secuencia. Los diagramas de CTT se leen de izquierda a derecha. En CTT el Layout es parte de la semántica. Entonces la idea del metamodelador es que pueda referenciar qué elementos, en el metamodelo, soportan los conceptos de la parte concreta y usados en el editor: tareas, clases, relaciones de dominio, relaciones de tareas, etc. Si cuando se hace una transformación no se dejan rastros de cómo se van creando las cosas se rompe el flujo de la generación automática de interfaces. Porque es que se está creando el proceso de cómo se está avanzando. Tanto para ir hacia adelante como para poder devolverse. Esta función la cumple el *remapping*: deben haber links para decir que inicialmente iba por un camino, pero con el remapping se tomó otro camino, por si se toma la decisión de retornar o de cambiar de ruta.

Cuando se realiza una transformación lo que interesa es una fracción, un contenedor de algo, lo mínimo para realizar una transformación. Es decir, tomar unas tareas y unos datos y a partir de ahí obtener una UI. La UI no es de todo. Se van tomando fracciones y esto se engloba bajo el concepto de gestión de diagramas. Pero se requiere que, cuando el diagrama contenga una clase, a la clase se le puedan agregar atributos, pero si la clase se encuentra en otra realización, en esa

otra realización la clase pueda mostrar otros atributos. La idea entonces es poder seleccionar qué atributos se quieren ver y qué atributos pueden participar en el diagrama, es decir, que en una transformación deben tenerse en cuenta los atributos que participan sobre el diagrama. La generación de interfaces, con esta nueva herramienta, también puede darse a partir de la tarea *InteraccionTask*. El metamodelo soporta toda la funcionalidad.

A nivel de metamodelado se tomó la decisión como si se fueran a hacer elementos concretos. Lo que está en el modelo es algo que soporta lo que está en el proceso de generación automática y es lo que se requiere para saber qué está bien o mal en el *mapping* (modelo de trazabilidad) y lo que faltaría hacer. La decisión fue identificar qué elementos se afectaban por el proceso de transformación, es decir, que lo referencia Limbourg[42] en la transformación. Como esta tesis es la extensión de un metamodelo, una de las tareas fue identificar y saber cómo adicionar elementos nuevos que cumplieran con las características del modelo para que hicieran lo mismo: funcionar correctamente. En *Chico*[26] hubo modificaciones sobre la base que entregó Limbourg[42], pues no se maneja el concepto de *'ModelElement'*. Entonces el diseño del metamodelo debe ser capaz de soportar la representación de la notación y el lenguaje en una herramienta (GMF). Todo lo que en el metamodelo es concreto se llama *elemento de modelado*. A nivel concreto se puede hacer genérico cualquier elemento de modelado, pero a nivel abstracto se limitó en CIAT.TDMBUID, desde el modelo de mapping a través de relaciones entre las clases: triggers; updates; observes. Para el caso de este trabajo de grado y lo propuesto como extensión de la herramienta 'chico'[26], particularmente el mapeo de la transformación, desde el metamodelo, el elemento de Transformación extiende de la metaclase *'Package'*, considerando que *'Package'* es la que contiene la metaclase *'Diagram'*.

El mapping se realiza sobre la tarea interacción (interactive). Cada ícono corresponde a una clase del metamodelo, quiere decir que si se tiene 1 clase que se vea con 4 íconos, eso no se puede en el *.gmfmap*. Por eso el metamodelo tiene la parte concreta. El ícono, en el mapping, lo define la figura. El mapping (modelo de trazabilidad) puede presentar problema cuando no es capaz de discernir qué concepto es. Se puede usar el mismo concepto para dos cosas distintas. Entonces lo que hace falta es tener un concepto por encima de ese (en el metamodelo: nivel abstracto y concreto). Para solucionar esta situación, se realizó un diccionario de elementos de modelado (por lenguajes – los que están contenidos en el metamodelo) y se identificaron los nombres que se repetían. Luego se identificaron los niveles (abstracto o concreto) en el metamodelo y se restringió como elemento de modelado. Por ejemplo, que la relación de mapping saliera desde una *task* pero sólo con destino a un *ModelElement*. Lo que se hace con el mapping (modelo de trazabilidad) es navegar el modelo para crear lo que se desea crear. Por ejemplo, un componente individual abstracto de interfaz puede contener otro contenedor (AC). La IU tiene presentación y dialógo. Para una cardinalidad de 1:1 en el atributo (modelo de datos–dominio) *name* lo más probable es que se represente por un *TextBox*, pero para una cardinalidad de 1:n por ejemplo, el atribudo *titulos* (modelo de datos) de la clase persona, lo más probable es que su representación sea un *Combobox*. En términos del marco conceptual se está hablando de un *AIO* (en un nivel concreto: Textbox y Box o combox), pero a nivel abstracto se está hablando de un *AIC*, (haciendo referencia a que es el mismo elemento solo que con distinta cardinalidad). A nivel abstracto se tienen los componentes individuales (y contenedores) y a nivel concreto se tienen los elementos de la representación: textbox y combobox, entonces debe haber un *mapping* que trace la relación entre los dos para saber en qué elemento *AIC* se reifica qué elemento concreto y qué elemento concreto es la abstracción de un *AIC*, por esa razón debe existir un *mapping*. Si no se configura la cardinalidad de los atributos, la herramienta no funciona. El algoritmo de transformación de CIAT.TDMBUID considera esta característica. Si los atributos no tienen la cardinalidad definida la herramienta no funciona.

En cuanto al remapping visual (que es el concepto para tomar alternativas distintas de la propuesta de la transformación inicial) deben haber otros links para indicar que inicialmente se

llevaba un camino, pero por el remapping se tomó otra ruta. Es decir, se adicionó un elemento que no fue tenido en cuenta, o se posicionó un elemento de manera distinta al que fue propuesto en la transformación inicial (esto es para devolverse o poder seguir por otro camino, alguna modificación en el modelo transformado; una modificación a la interfaz concreta, por ejemplo, un cambio de ubicación de una tarea a un contenedor distinto). Al generar automáticamente una IU se crean los *links de trazabilidad* (mapping, automáticamente quedan enlazados los elementos con las tareas, primero la UI abstracta (AUI) y luego la UI concreta (CUI)). Si las tareas no se registran(mapean) en su totalidad lo más probable es que los elementos que las representan queden en distintas interfaces.

4.3. Creación del archivo generador de modelos

El siguiente paso es el de crear el generador de modelo *ciat.tdmbuid.genmodel* a partir del modelo de dominio. Este paso es automático y requiere de poca interacción por parte del desarrollador. Este generador de modelos es una herramienta de generación automática de código de EMF necesaria para construir las clases Java del modelo de dominio y los *EditParts* necesarios.

El archivo *.genmodel* contiene toda la información susceptible de ser convertida a clases Java tal como se conocen, esto es: si en el genmodel existe un objeto *Class* será traducido a una clase *Java* con los atributos que vengan especificados en el metamodelo (modelo de dominio). Este generador de modelos utiliza una serie de plantillas predefinidas que son especificadas en el lenguaje de emisión de plantillas Java (JET). La tecnología JET es utilizada para la transformación de modelos a texto a partir de modelos ECORE.

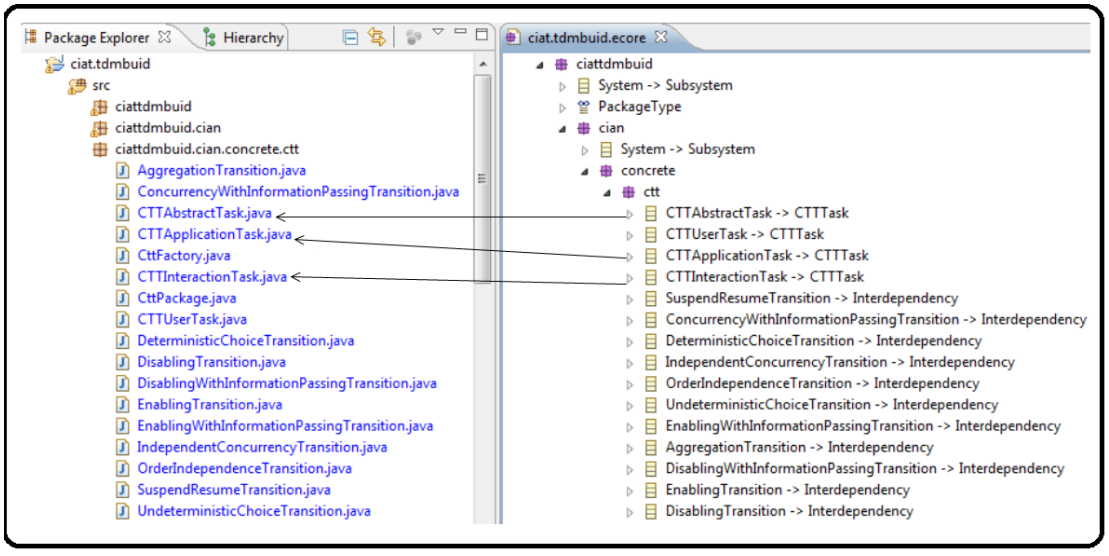

Figura 4.6: Correspondencia entre .genmodel y código generado automáticamente

EMF genera las clases y establece relaciones entre ellas basándose en el modelo MOF[61] que se haya construído. Toda la lógica de los editores en términos de modelos de dominio se puede crear con EMF y delegar en dicha tecnología para que escriba el código asociado a modelos de dominio [22]. Un ejemplo de lo anterior puede observarse en la figura 4.6

4.4. Definición del modelo gráfico (*Sintaxis Concreta*)

La especificación de la sintaxis concreta debe ser uno de los pasos preliminares en la especificación de cualquier lenguaje para definir sus particularidades y el *mapping* con respecto de la sintaxis abstracta. Esta especificación tiene como partida el análisis visual de cómo se presentarán al desarrollador los elementos de modelado del lenguaje. Este modelo define la estructura de componentes gráficos usados para representar los conceptos[26].

Para crear la definición gráfica del editor se usa el archivo *ciat.tdmbuid.gmfgraph* el cual especifíca el aspecto de los componentes gráficos de la notación que se desee implementar. El diseño de la definición de la sintaxis concreta debe considerar aspectos de *layout*, etiquetado, estructura y contención (*compartments*). Ver elementos de la definición gráfica en 3.4. Cada componente de la notación gráfica del lenguaje que se diseña debe ser especificado utilizando elementos del modelo gráfico GEF[80], por ejemplo: rectángulo, etiqueta, nodo, enlace, etc.

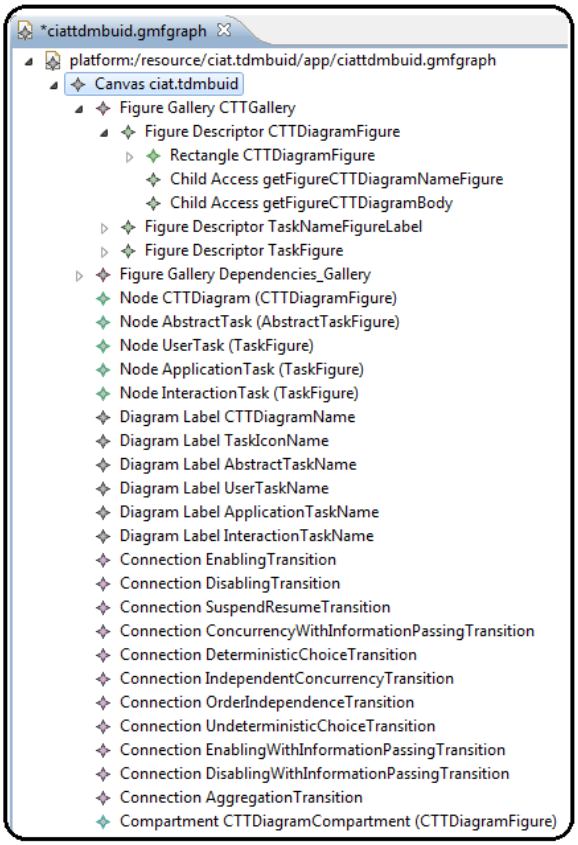

Figura 4.7: Definición gráfica para ciat.tdmbuid

Para ilustar el funcionamiento de la definición gráfica se han seleccionado los elementos que construyen el diagrama de Tareas CTT. Ver figura 4.8. En este diagrama se requiere de un rectángulo que contiene otro rectángulo, además de una etiqueta que da nombre al diagrama. El rectángulo llamado *CTTDiagramFigure* representa el rectángulo exterior. La etiqueta *CTTDiagramNameFigure*, que es un objeto de tipo *Label* es la que da el nombre al diagrama. El rectángulo llamado *CTTDiagramBody* es el rectángulo interno contenedor de los demás elementos. El objeto label *TaskNameFigureLabel* es quien identifica la tarea (el nombre). El rectángulo *TaskFigure* es el contenedor de la tarea y el objeto label *TaskIconFigure*, aunque usa el mismo

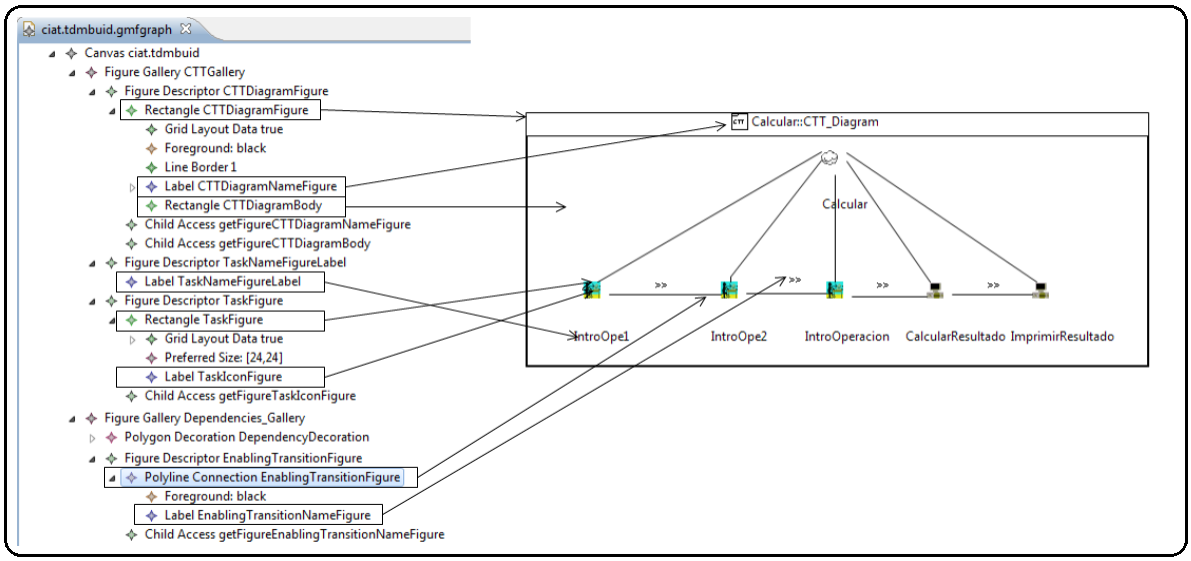

Figura 4.8: Definición Gráfica para el diagrama CTT.

descriptor de figura asociado, representa el tipo de tarea. Quiere decir esto que se puede usar el mismo tipo de figura para representar distintos tipos de tarea.

Lo nuevo de la sintaxis concreta, que se presenta en este trabajo, es: notación y modelado de la interacción. Aqui se tienen cinco tareas nuevas: simple cooperativa, nueva cooperativa, simple individual, nueva individual e Individual Interaction (interactive). Para esto se definió la parte abstracta (metamodelo) y la parte concreta (EMF-GMF). El concepto Task soporta las tareas interactivas y las interactives, dado que es el mismo core, de manera que los conceptos en el metamodelo se soportan y por eso se pueden relacionar/visualizar.
El diagrama de Interacción es un CTT, con este diagrama puede darse el diálogo externo aplicando el algoritmo de Luyten[46]. Cada tarea debe estar asociada a una ventana y no a un widget (si fuera lo último, sería diálogo interno). Los algoritmos de Limbourg[42] y de Luyten[46] son puros contenedores; Limbourg solo revisa y genera contenedores. El diagrama CTT no es un diagrama de UI, es un diagrama de tareas, entonces las personas terminan pensando en UI pero haciendo tareas, el asunto real es pensar en tareas interactivas, qué hace el usuario y qué hace el pc, pero nada tiene que ver la UI. La propuesta *TDBMUID* recalca la separación entre el modelado de tareas, y los datos que se manipulan en las tareas con la presentación de la UI. Del diagrama de tareas sale el modelo de diálogo: Interno y Externo (algoritmo de Luyten[46]).

Por otra parte, *CIAN* apunta hacia los procesos de negocio, en el modelado de la actividad (es decir CIAN) se requiere conservar la notación *CIAN* original, pero adicionando el modelado de negocio: diagrama de actividad; diagrama de objetos de negocio y máquinas de estado. A nivel de modelado de negocio se requiere hacer el equivalente entre el proceso de negocio (caso de uso de negocio: *UML*), actor sería *Worker*; lo que se requiere sobre *CIAN* es tener los conceptos de Cliente y Worker. La deficiencia de la actual notación es que en las tareas internas no se sabe qué relación hay entre el cliente/actor con las entidades (datos presentados en la tarea interactive) y tampoco es posible ver la relación entre los datos (con la tarea cooperativa actual). Con la tarea *NewCooperative*, presentada como parte de la nueva sintaxis concreta de este trabajo, se da reemplazo a los tres diagramas de *UML*.

Esta herramienta CIAT.TDMBUID posee gestión de diagramas y gestión de modelos. La or-

ganización del modelo es una parte del lenguaje. La estructura del modelo debe ser parte del lenguaje, es decir, no solo el diagrama sino la estructura completa debe ser parte del lenguaje; la navegación, también, debe ser parte del lenguaje. También el metamodelo debe soportar el esquema de navegación. El lenguaje debe soportar las transformaciones. *UML* no tiene elementos de modelado ni se refiere a cómo hacer *mapping* (modelo de trazabilidad), esto debe ser parte del lenguaje. Los elementos notacionales son los elementos de modelado. Generalmente no se trabaja con modelos sino con diagramas. Un diagrama se compone de un canvas (lienzo), elementos de modelado. Es decir, casi nunca directamente se hacen modelos sino diagramas, y el modelo, entonces, serán los documentos. Un modelo entonces, puede resumirse en un conjunto de diagramas y documentos. Se expresa a través de diagramas y documentos. Pero el modelo lo que tiene es un repositorio, y este repositorio es casi siempre abstracto. Casi siempre este repositorio está representado por el *browser*, ya hablando a nivel tecnológico (es decir, de un tool). A través de los elementos de modelado es que se construyen modelos. La interfaz que con esta herramienta se provee, para el uso del lenguaje, son los diagramas, el canvas, los elementos de modelado y el browser. A la persona (ingeniero) que use CIAT.TDMBUID se le dejan ver solamente los elementos de modelado, él no ve el metamodelo. Esto se llama notación. La notación es lo que se le permite a la persona que usa la herramienta. La notación es la interfaz del lenguaje. Por otra parte están los elementos notacionales. Esta es la parte concreta del lenguaje. La tarea entonces fue identificar todos los elementos notacionales, los elementos de abstracción y los niveles de granularidad. En el trabajo de la elaboración/integración de la herramienta CIAT.TDMBUID se trabajó sobre tres dimensiones:

1. Interfaz de Usuario (usiXML).
2. Datos (UML).
3. Labor (actividad; CIAN)

El diálogo, sobre CIAT.TDMBUID, se concibe con tareas, datos y *widgets* sobre diagramas temporales (mapping – modelo de trazabilidad). Los anteriores elementos modelan cosas distintas. La metodología propuesta por Giraldo[26] trabaja primero la parte semántica basada en los datos (tratamiento de la adyacencia) y luego la parte interactiva basada en los recorridos cognitivos.

El comportamiento de la interfaz de usuario puede darse con distintos niveles de granularidad. Deben manejarse distintos niveles de granularidad, sobre las interfaces, porque conceptualmente pueden estar separados del proceso cognitivo, CTT es un lenguaje para manejar tareas. CTT no maneja niveles de granularidad. Con CTT toca estar abstrayendo los niveles de granularidad (una consecuencia del lenguaje es permitir mejorar la expresividad). Al no tener niveles de granularidad los diagramas crecen en exceso. CTT está concebido para expresar tareas. Lo usan, generalmente, para generar interfaces de usuario, pero esa no es la concepción original. Giraldo[26] aisla estos inconvenientes en su metodología modelando a partir de los datos, a partir de estos se obtiene una primera versión de UI, luego la segunda versión la obtiene de las tareas y luego, lo que hace es repasar las tareas buscando algo 'bueno'para meter a los datos (*remapping*). La primera versión de UI obtenida a partir de datos se ubica a nivel de negocio, allí no hay *widgets* porque no se habla de software, y la versión de UI que provee CTT es una que hereda la estructura de layout, semántica y adyacencia, por ejemplo, que el nombre vaya al lado del apellido, eso se decide junto con el usuario, y ese *mapping* debe existir en el metamodelo para que se pueda soportar esta función. El *remapping* no se hace solo, este se hace a mano. Limbourg[42] establece las condiciones para realizar un mapping. El CTT, en [26] sale del recorrido cognitivo que se realiza con el usuario: los datos. Los datos de la forma. El cliente realiza un modelo mental pero que es un prototipo en papel (no confundir con modelar el sistema) solo se modelan los datos que se manejan en un negocio respecto de la Interfaz de Usuario de Negocio. Esto es basado en cómo el cliente quiere ver sus datos. La interfaz de usuario es una expresin de la forma. Los datos de la forma son la persistencia de la UI, en otras palabras, la persistencia

de la presentación. Los datos de la forma también sirven cuando se realizan consultas a la base de datos, que sería la estructura de datos que se enlaza a la base de datos y luego se mostrarían los resultados de esa consulta. Los reportes también son interfaz de usuario. Por lo tanto, el reporte debe estar atado a los datos de la forma. El reporte primero debe llevarse a los datos de la forma y el script debe coincidir con la plantilla que genere el reporte. Generalmente la presentación del reporte está dispuesta según el modelo de la base de datos y no presentado de acuerdo al modelo de la UI. Lo que se requiere es una entidad intermediaria basada en el modelo mental del usuario, en el que se preformateen los datos antes de enviarlos por cualquier canal al usuario: papel; interfaz; etc. (uso del patrón decorador; es un plantilla). Cuando se habla de UI generalmente se piensa en un pc, pero cuando el pc sea papel, la interfaz será de papel. Esto se trata de analizar la forma y describir los datos que reflejen fielmente la forma. Por eso se llaman datos de la forma. La parte que representa los datos de la forma no se encuentra en el metamodelo CIAT.TDMBUID. A nivel abstracto se considera como el mismo diagrama de clases (DomainDiagram) sobre el metamodelo. Básicamente los datos de la forma se van a convertir en el DomainDiagram, pero el DomainDiagram está ubicado en el nivel de sistema, mientras que los datos de la forma están ubicados a nivel de negocio. Las ventanas de contexto son pequeños bloques de interfaz. Se llama ventana de contexto porque es reutilizable. Y es una ventana dentro de un contexto específico. Y está asociado a los datos.

En los métodos tradicionales la persona realiza el modelo de tareas pero sin pensar en la interfaz, solo piensan en las tareas pero pensando en la interfaz. Hacen tareas pero pensando en qué interfaz van a obtener como resultado. En cambio, Giraldo[26], inicia a partir de una interfaz que es basada en el modelo mental y el cliente realiza un recorrido cognitivo de esos datos, de cómo le gustaría usar esos datos. (Qué quiere hacer el cliente y qué quiere que haga el sistema: tareas; y qué quiere hacer con el sistema en conjunto). Esto, todo, corresponde al modelo de Interacción. El diálogo se basa en la relación entre tareas los datos y los componentes de UI. La estructura, el layout, la semántica, la interfaz van a estar más de acuerdo al modelo mental del usuario, no a la generación automática de las tareas; por eso la metodología propuesta por Giraldo[26] realiza un balance, porque lo que se debe realizar que el cliente se enfoque en las tareas en lo que quiere del sistema, y que se enfoque en lo que la UI va a tener cuando esté pensando en la UI y no en las tareas. Se tiene, por ejemplo, un árbol de CTT, en donde se pueden realizar distintas operaciones y que el sistema pueda calcular distintos factores y resultados; esto es, un árbol con diferentes variaciones. Una cosa es la cantidad de controles que salen y otra es la cantidad de interacciones que se vayan a hacer sobre eso, eso es un problema que ha tenido CTT, que se ha usado para hacer UI sabiendo que es una notación para representar interacción de tareas. El CTT siempre va de izquierda a derecha y tiene un nivel jerárquico. La primer tarea que puede realizarse es la del lado izquierdo. Limbourg[42] realizó el algoritmo indicando que si una tarea tiene hijos esa tarea es un contenedor.

4.5. Generación de la paleta de herramientas

La paleta de herramientas es la opción requerida para manipular los elementos de modelado. Esta se compone por la barra de herramientas y diferentes menús que pueden definirse para un diagrama. El propósito de la la paleta de herramientas es crear instancias de cada uno de los elementos sobre el *canvas mapping* para sean reconocidos sobre el modelo de mapeo. Esta paleta de herramientas es la *'barra de herramientas'* que el desarrollador tiene disponible.

4.6. Definición del modelo de mapping

Este modelo relaciona el modelo de dominio (ver 4.1), el modelo de definición gráfica (ver 4.4) y la paleta de herramientas (ver 4.5). El mapeo define las relaciones de contención de los ele-

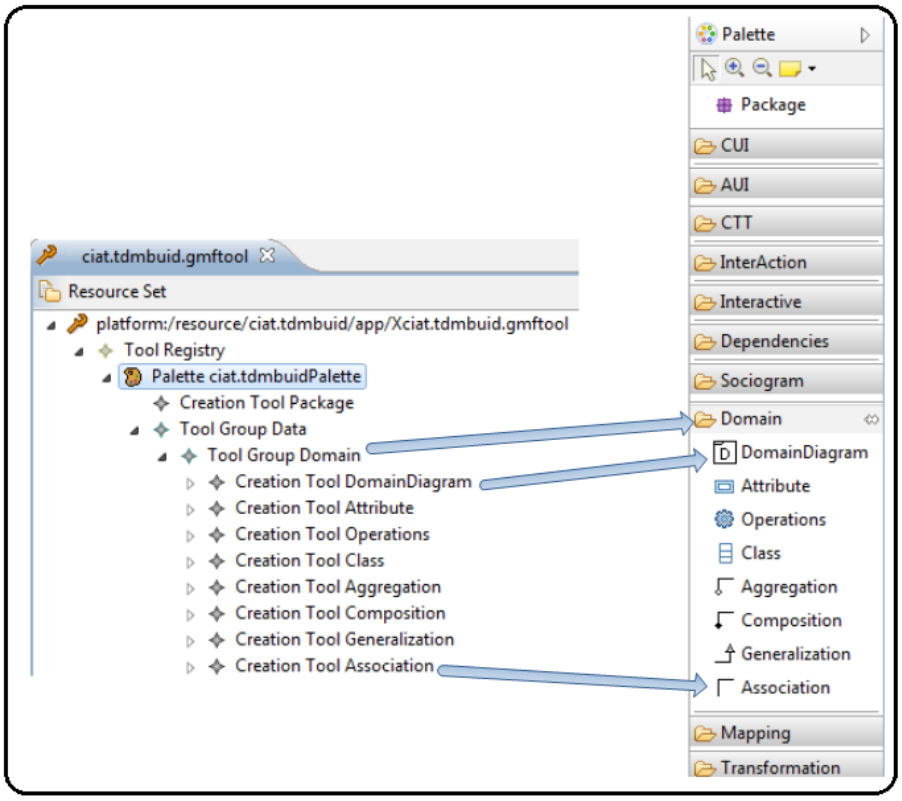

Figura 4.9: Definición Herramienta para ciat.tdmbuid

mentos gráficos para poder instanciar los objetos de dominio que representan su información. Para lograr el *Modelo de Mapping* es necesario conocer muy detalladamente la estructura de la sintaxis abstracta y concreta. Un cambio en los modelos que enlaza el *mapping* y éste debe ser actualizado, pues es sensible a sus modelos referenciados de acuerdo a la especificación, contención e instanciación de sus elementos. El mapeo es utilizado para la creación del modelo de generación de diagrama de extensión '.gmfgen'.

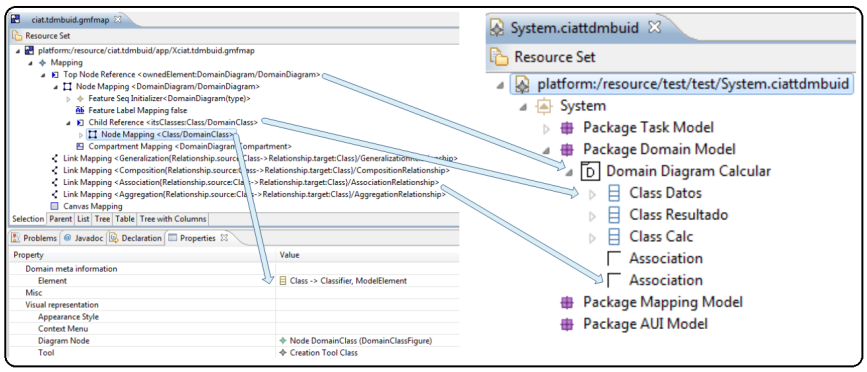

Figura 4.10: Definición Mapping para ciat.tdmbuid

El archivo de *mapping* contiene todos los *Top Node References* y los *Links* y, al final, el *Canvas Mapping* (ver sección 3.6). Un editor solo posee nodos y links; y sobre el mapping es que se

"arma" el editor de programas. Se llama node (en el mapping) por que son rectángulos. Lo que se está construyendo, con este trabajo de grado, es una herramienta de modelado que solo tiene Nodos y/o Links. En la jerarquía del mapping el elemento superior es el Top Node Reference. Lo que se busca en el mapping es asociar a un elemento de modelado (EClass) un *Tool* y un *diagram*. Esto es lo que hace el mapping. Cada elemento que aparece en el editor proviene del mapping. El mapping es algo que relaciona el gráfico con el elemento(dominio) y con el tool. En el modelo de mapping, el elemento Canvas Mapping, sobre la propiedad Element, para el caso de CIAT.TDMBUID, es la EClass *'Package'*. El concepto canvas (ver 3.6.1), está representado en el metamodelo por la metaclase *'Package'*. A partir de que diga *Mapping* (en el modelo de Mapping) todo es heredado de la metaclase *'Package'*. Si en *'Package'*, hablando de la vista de árbol, se desean ver los diferentes elementos, entonces por cada tarea se debe realizar hacer una entrada. La entrada significa relacionar el elemento que se va a arrastrar del *tool* al sitio que se va a soltar, identificando el destino correcto (sitio donde se va a soltar el elemento del tool seleccionado) y adicionar el ícono que lo represente (*.gmfgraph* ver 3.4). En otras palabras, para el caso de la *EClass 'Package'*, esta EClass debe tener una relación que sea capaz de poder dibujar sus elementos superpuestos (a manera de contenedor) y de esta manera se consigue la libertad de realizar el dibujo que se desee. La única forma de poner un elemento es a través del canvas o de un compartent (elemento que permite la anidación de figuras). La manera de permitir generar un elemento dentro de una caja, es que esa caja sea un compartment. La EClass que se asocie al *canvas* debe ser una clase que sea capaz de contener todo el modelo, es decir, todo lo que se le quiera adicionar al canvas. Expresado de otra manera, la clase que genere contención es la clase que se debe referenciar en el *mapping*. Si una clase no tiene contención no se le puede hacer mapping. El mapping puede referenciar todos los elementos que se deseen siempre y cuando el modelo (metamodelo) lo soporte (solo se puede mapear lo que el metamodelo permita); y a su vez, todo lo que se desee ver solo se puede ver si está referenciado en el mapping. Todo lo que se hace con el mapping es navegar el modelo para crear algo que se desee crear. Cada elemento que aparezca en el editor es un mapping, porque el mapping es algo que relaciona el gráfico, con el elemento con el tool. El modelo (Diagram) es el que contiene el diagrama y el diagrama es de tipo modelo (Diagram), para que los elementos que se tengan por dentro, en cualquier momento, puedan ser visualizados en un diagrama a parte. Sólo se puede mapear, a nivel visual, lo que el metamodelo permita.

Visualización de un Paquete sobre el Canvas. Esa es una decisión que se tomó a nivel de mapeado. Por ejemplo, para el diagrama de transformación, se puede adicionar cualquier tipo de diagrama (CTT; Domain; InteractionTask), pero no los elementos internos de ese diagrama, en otras palabras, por ejemplo, cuando se selecciona el CTTDiagram, solo se muestra el diagrama pero no las tareas. En eso radica el *'qui'* del metamodelo, si éste no se encuentra bien realizado, habrán elementos que no se pueden representar en la herramienta de modelado CIAT.TDMBUID. Lo que se quiere representar es una referencia a cualquier diagrama y no un diagrama como tal, por lo que se exige un diseño en el metamodelo que permita, en la sintaxis concreta (refiriéndose al canvas), ser cualquier tipo de diagrama, no solamente la clase *Diagram* (que es la genera contención en el metamodelo) sino que en cualquier momento, la raíz fuera, por ejemplo, un diagrama en particular, (CTTDIagram; DomainDiagram; etc.), finalmente, como está diseñado el metamodelo, la clase *Diagram* (ver figura A.2) puede contener cualquier cosa. La primera decisión que debió tomarse fue qué se quiere dibujar sobre el canvas. Si se desea ver en la clase *Diagram* cualquier cosa, toca realizar entradas para cada tarea, significa esto que, lo que se vaya a arrastrar del *tool* sobre el *canvas* debe tener una relación. En otras palabras, la clase *Diagram* debe tener una relación que sea capaz de poder dibujar las tareas sobre el lienzo, y de esa manera obtener la total libertad de poder adicionar el dibujo que se desee. Un paquete sobre el canvas puede visualizarse de diferentes formas, pero no todas las veces se pueden escoger muchas opciones. Al seleccionar una opción de visualización, las demás opciones de visualización pueden quedar inhabilitadas, es decir, la selección de visualización elegida es excluyente para las demás opciones. Es posible anidar las visualizaciones, es decir, se puede

seleccionar la visualización de un paquete sobre un canvas, pero también se puede seleccionar la visualizacin de un paquete dentro de otro paquete. En el caso del nuevo diagrama (el de transformación), este diagrama puede, por dentro, contener otros tipos de diagrama. Para este caso, lo interesante es que el diagrama comunica que transformación se aborda en el diagrama. De tal forma que la manera de visualización de un diagrama puede ser múltiple, pero siempre excluyente a las demás opciones de visualización. Las diferentes maneras de poder visualizar un diagrama radican en que en un momento un diagrama puede hacer parte de un paquete, en otra visualización el diagrama hace parte de un *TransformationDiagram*, por ejemplo, y en otra visualización el diagrama puede hacer parte del canvas, pero no es porque sea el canvas. Si el metamodelo no se construye adecuadamente, así se tenga la sintaxis abstracta y aunque se tenga la sintaxis concreta, si el metamodelo no se encuentra bien definido habrán cosas que no se podrán realizar en la visualización. Debe decidirse, primero, lo que se quiere dibujar. Luego debe revisarse el modelo (metamodelo). Se debió, entonces, realizar un diseño, en donde el canvas pudiera tomar la forma deseada de cualquier cosa. En otras palabras, que en un momento dado (para un diagrama), el canvas pueda ser CTTDiagram, o si se desea usar/crear un diagrama de transformación, el canvas, en ese momento sea *TransformationDiagram*. Por eso, el canvas debe ser del tipo de la metaclase *'Package'*. De manera que basando el canvas sobre la EClass *'Package'*, el canvas puede tomar esa multiforma, para cada diagrama generado.

Los conceptos utilizados para la generación de la definición gráfica son: tooling = tool; graph = node, link, compartment, label; ecore = class y relationships. La representación de esta definición gráfica es el *Canvas Mapping*. Este referencia la EClass *'Package'* para el caso de esta herramienta CIAT.TDMBUID. Esta clase *'Package'* dentro de cualquier diagrama, por ejemplo, el de tareas CTT, que hereda de *'Package'* puede crear un *'CTT_Diagram'*. Esta es la forma como se generó este mapping. Ubicándose en un elemento de modelado, en este caso *'Package'* y analizando qué otros elementos se podían crear (buscando contención) a partir de ese propio elemento. Luego, se analizaron los nombres de las relaciones de contención (composición) porque es a través de estas relaciones que se crean los distintos elementos de modelado. La relación de contención que va desde *'Package'* hasta el concepto *'Task'* se llama *'itsTasks'*, quiere decir esto, que si se está en la clase *'Package'* se puede crear cualquier tarea CTT por medio de la relación *'itsTasks'*. El canvas es la raíz de definición gráfica de todos los elementos. Quiere decir esto que en este nivel de anidamiento, si el canvas es *'Package'*, en este nivel se puede crear cualquier tarea CTT. Pero entonces una cosa es que se pueda crear una tarea en este preciso lugar y otra cosa es que realmente se quiera eso para la herramienta. Podía suponerse que sólo se quiere que las tareas CTT estén contenidas visualmente en un diagrama CTT (*'CTT_Diagram'*), entonces aunque perfectamente es válido crear una tarea en el canvas (porque esta posibilidad la está dando el modelo de dominio porque así se diseñó el metamodelo) no se quiere eso de manera gráfica. Entonces se podría tomar la decisión de que ningún elemento de modelado quede por fuera de un contenedor. De esta forma, no sería conveniente que una tarea CTT pueda ir sola en el canvas sin que en ese mismo canvas haya definido primero un diagrama ctt *'CTT_Diagram'*. La idea es que el elemento contenedor (la contención) no permita elementos por fuera suyo establecidos en el nivel de jerarquía del modelo. Pero otra cosa es a nivel gráfico. Lo que se está definiendo es una herramienta gráfica de modelado, es decir, el modelo puede permitir ciertas cosas aunque no se quieran representar sobre la herramienta gráfica. Lo que no establezca en el *mapping* no se representa en la herramienta aunque el modelo lo pueda soportar. Este modelo soporta crear tareas CTT a nivel de *'Package'* y como *'Package'* es el canvas por ende permite crear tareas en el canvas, gráficamente hablando.

4.7. Creación del generador del plugin

El archivo *'.gmfgen'* es a partir del cual se genera todo el código final del editor en forma de *plugins*. Dicho archivo toma la información especificada en los archivos *'ciat.tdmbuid.gmfmap'*, *'ciat.tdmbuid.gmfgraph'* y *'ciat.tdmbuid.gmftool'* y construye el modelo jerárquico final que da lugar al código del editor de creación de diagramas de la herramienta de modelado.

4.8. Implementación de transformaciones

El punto de partida de la construcción de la interfaz de usuario es la especificación de las tareas y los datos del dominio. A partir de esta representación inicial se obtiene la interfaz de usuario abstracta (ver 2.1.3). Se trata de una especificación de la interfaz de usuario en términos de *Objetos de Interfaz Abstracta* (ver 2.1.3) y de relaciones abstractas. En esta especificación de *AUI* se definen elementos contenedores generales. Primero se identifican los *Abstract Containers* (ver 2.1.3) y los *Abstract Individual Componentes*. Luego, se sitúan los *AIC* dentro dos contenedores identificados previamente. Finalmente se adiciona el diálogo entre los elementos abstractos. De manera que se debió implementar una serie de transformaciones entre los dos modelos iniciales: modelo de tareas y modelo de dominio y sus representaciones para llegar a obtener la *AUI*.

4.8.1. Implementación de la Interfaz de Usuario Abstracta - AUI

Primero se identifican los elementos *Abstract Containers (AC)* y *Abstract Individual Components (AIC)* según sea la distribución del modelo de tareas CTT. Esta identificación es reconocer las tareas de las que no se deriva ninguna subtarea, es decir, las *tareas hoja* que serán elementos de la interfaz y se recorre el árbol recursivamente para crear los contenedores. Cuando se encuentra una tarea que se descompone en subtareas se genera un *AC* y varios *AIC* por cada subtarea. Cada hoja tarea tiene dos atributos: *actionType* y *actionItem*. Los verbos describen el tipo de la acción *(Action Type)* y los sustantivos los objetos que manipulan los verbos *(Action Item)*.

Segundo se seleccionan los *Abstract Individual Components (AIC)* dada la configuración de los atributos: *(Action Type)* y *(Action Item)* se obtiene un *facet* (ver 2.1.3) que se asocia al *AIC* que corresponde a la tarea.

Tercero, se establecen las relaciones *espacio-temporales* entre *AIOs*. Se establece la relación *Abstract Adjacency* (esto significa que dos AIOs lógicamente adyacentes deben aparecer contiguos en el modelo de interfaz concreta – CUI) entre AIOs según los operadores *'>>'* del árbol de tareas. Por ejemplo dos tareas que sean hermanas y estén unidas por una relación *'>>'* dan lugar a una relación *Abstract Adjacency* entre los AIOs a los que correspondan.

Cuarto, se establece el diálogo de control abstracto. Se construyen las relaciones *Dialog Control* entre AICs. Esta relación permite especificar un flujo de control de la interacción entre objetos abstractos. Para indicar este tipo de relación se utilizan los mismos operadores temporales que los utilizados para relacionar tareas. Un AIC que sea destino de una relación *Dialog Control* no estará operativo hasta que el AIC que es fuente haya terminado.

Quinto, obtener la intefaz de usuario abstracta *(AUI)* a partir de las relaciones de trazabilidad. Las tareas que sean fuentes de una relación de trazabilidad, sobre los métodos del dominio, derivan en una relación *triggers* entre el *AIC* y el elemento del dominio (clase, atributo, método).

Entonces en este punto, para las reglas hay que poner para cada regla de transformación la transformación en ATL para demostrar cómo se hace la transformación sobre el metamodelo.

actionType/actionItem	***AIC Facet Type/actionType/actionItem***
Start-Go/Operation	Control
Stop-exit/Operation	Control
Select/Element	Input/Select/(Attribute value)
Select/Collection	Input/Select/(Object Set)
Create/Element	Input/Create/(Attribute value)
Create/Collection	Input/Create/(Object Set)
Delete/Element	Input/Delete/(Attribute value)
Delete/Collection	Input/Delete/(Object Set)
Modify/Element	Input/Update/(Attribute value)
Modify/Collection	Input/Update/(Object Set)
View/Element	Output/View/(Attribute value)
View/Collection	Output/View/(Object Set)
Monitor/Element	Output/Monitor/(Attribute value)
Monitor/Collection	Output/Monitor/(Object Set)
Move/Element	Input/Move/(Attribute value)
Move/Collection	Input/Move/(Object Set)
Duplicate/Element	Input/Duplicate/(Attribute value)
Duplicate/Collection	Input/Duplicate/(Object Set)

Tabla 4.1: Correspondencia entre la configuración de AIOs y los facets que formarán la AUI [42]

4.8.2. Reglas de Transformación

Estas reglas de transformación son la propuesta de Limbourg[42]. Este utiliza notación gráfica, en forma de grafo, para representar dichas reglas. Las partes que constituyen este grafo son:

4.8.2.1. *LHS – Left Hand Side*

Expresa un patrón gráfico que, si coincide en el gráfico anfitrión, será modificado para dar lugar a otro gráfico llamado grafo resultante. Un *LHS* puede ser visto como una condición bajo la cual una regla de transformación es aplicable. Establece qué patrones del modelo son objetivo de la transformación.

4.8.2.2. *RHS – Right Hand Side*

Establece el patrón de salida, una vez ejecutada la transformación.

4.8.2.3. *NAC – Negative Application Condition*

Funciona como una precondición que indica qué elementos *NO* deben existir antes de ejecutar la regla. También conocidos como *contextos prohibídos*, son afirmaciones que se han de mantener falsas antes de la aplicación de una regla.

4.8.2.4. *Identificación de la Estructura AUI*

Se identifican primero las tareas hoja (tareas de las que no deriva ninguna sub–tarea) que van a ser elementos de la interfaz y se irá subiendo recursivamente en el árbol para construir los contenedores. Cuando en un árbol se encuentra una tarea que se descompone en varias sub–tareas se genera un AC y varios AIC por cada sub–tarea.

Regla 1. Para cada hoja de tarea del árbol de tareas, crear un AIC. Para cada tarea, padre de una hoja de tareas, crear un AC. Unir el AC y el AIC por una relación de contención. Esta regla

detecta qué patrones del árbol CTT son susceptibles de transformarse en un *Abstract Container*.

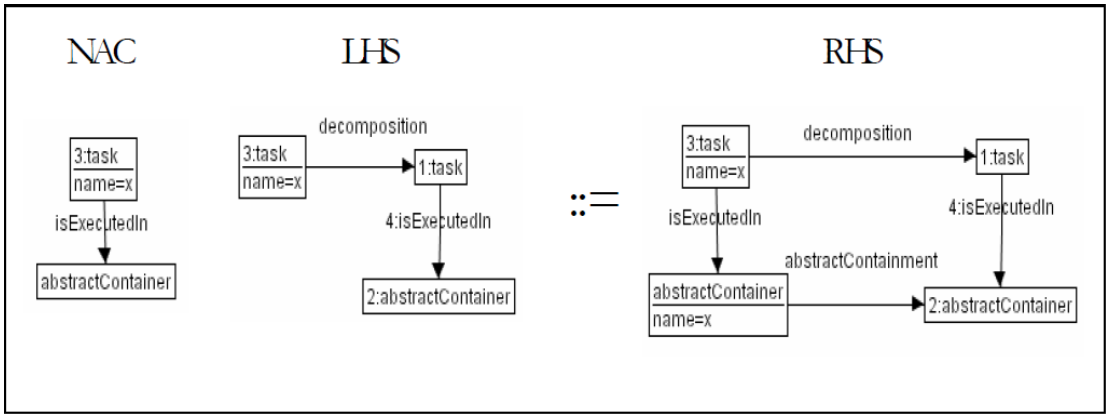

Figura 4.11: Creación de un contenedor abstracto AC derivado de la estructura del modelo de tareas.

Regla 2. Cuando una tarea se descompone en otra(s) tarea(s), crear un AC que contenga todas las tareas en las que se descompone la tarea principal. Establece que se han de buscar los padres de las tareas que actúan de raíz de los sub–árboles encontrados por la anterior regla e ir recorriendo el árbol de abajo hacia arriba hasta llegar a la raíz principal del árbol, agrupando cada porción del árbol en un AC que contenga los AC situados más abajo.

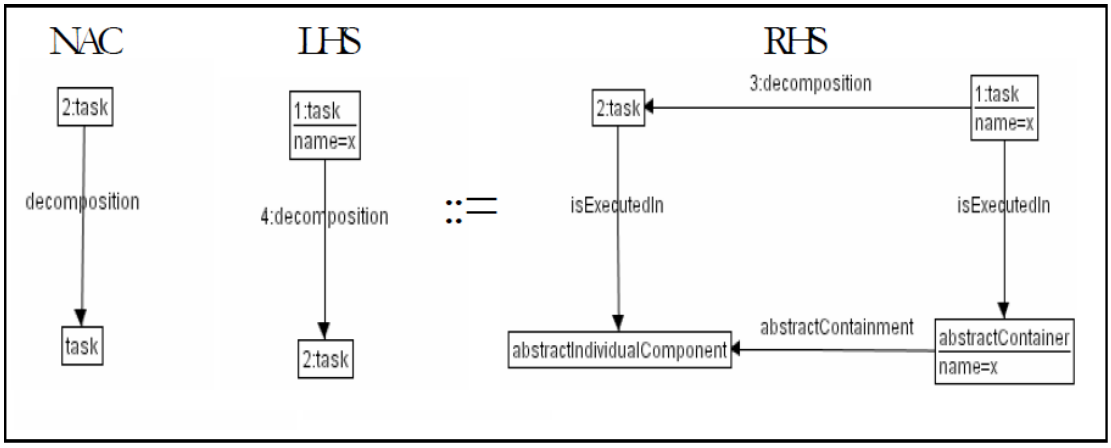

Figura 4.12: Creación de un Componente Individual Abstracto AIC derivado de la estructura del modelo de tareas.

4.8.2.5. *Selección de Componentes Abstractos Individuales*

Las tareas hoja de los árboles CTT tienen asociado un objeto de interacción abstracto que tiene dos atributos: *actionType* y *actionItem*. Según sea la configuración de estos atributos dará lugar a un *facet* que se asociará al AIC que corresponde a la tarea (esta correspondencia se establece en la anterior transformación). En la tabla 4.1 se muestran las posibles configuraciones de los

objetos de interacción abstractos y los *facets* a los que dan lugar en esta transformación.

Regla 3. Por cada *AIC* asociado a una tarea, si la tarea tiene establecida una relación de trazabilidad a un método, *(actionType=Start/Go)*, se asigna un *Facet* al *AIC*, que active el método.

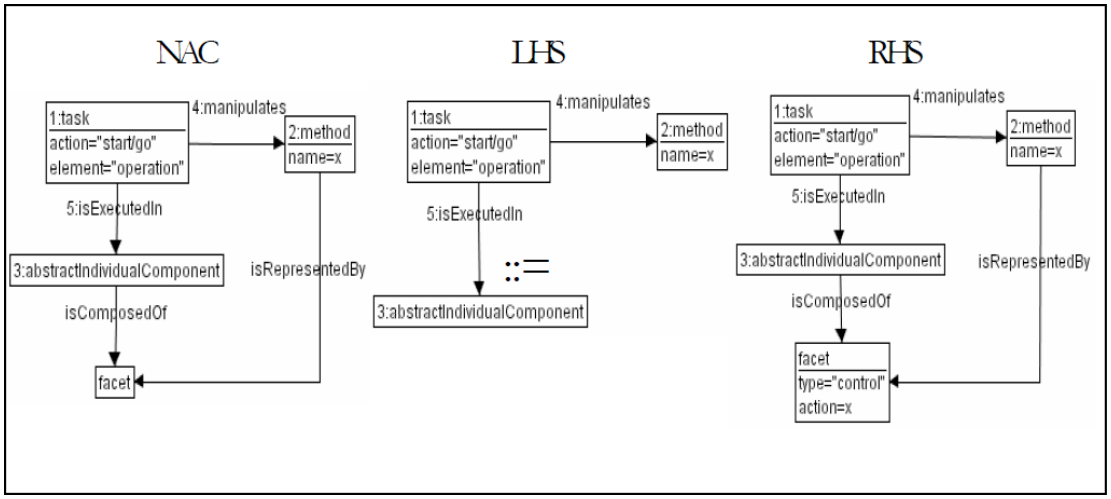

Figura 4.13: Creación de Facets para AICs derivado de la estructura del tipo de acción de tarea.

4.8.2.6. *Disposición Espacio–Temporal de Objetos de Interacción Abstractos*

Regla 4. Si existen dos tareas en el mismo nivel (hermanas) secuenciales (de tipo '>>') y estas tareas tienen asignado un *AIC*, crear una relación de tipo *'Abstract Adjacency'* entre los *AIOs* correspondientes.

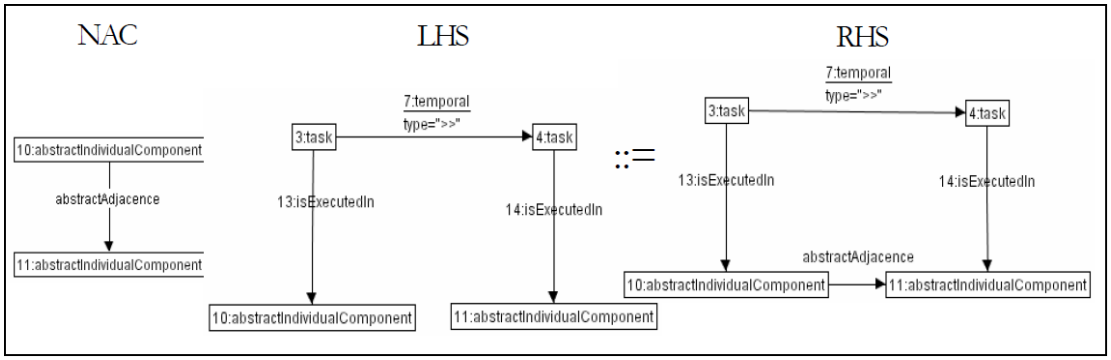

Figura 4.14: Creación de adyacencia entre AIOs derivada de relaciones temporales de secuencialización.

4.8.2.7. *Definición de Control de Diálogo Abstracto (ADC)*

Regla 5. Por cada pareja de tareas hermanas que tengan asignados *AIC* y que estén unidos por una relación temporal, crear una relación temporal entre los *AIC* que las contienen con

la misma semántica que la relación temporal que las une en el modelo de tareas (definir una relación *Dialog Control* entre esos dos *AICs*).

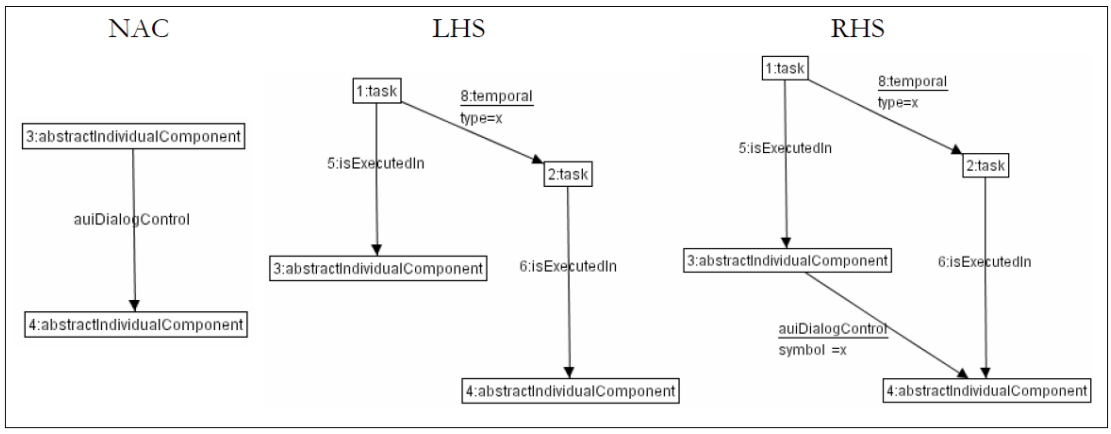

Figura 4.15: Creación de *Dialog Control* derivado del modelo de tareas.

4.8.2.8. *Derivación de AUI a relaciones de dominios*

Regla 6. Por cada tarea que manipula *(manipulates)* un método, el *AIC* que la representa desencadena/ejecuta *(triggers)* el método.

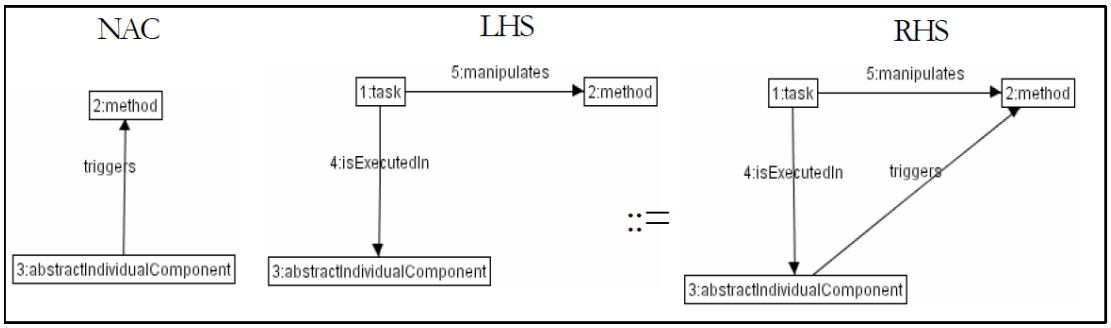

Figura 4.16: Ejecución de mtodo derivado del mapeo entre modelo de tareas y modelo de dominio.

Regla 7. Si dos tareas hermanas manipulan el mismo atributo, la relación temporal entre ellas es de tipo *ActivaciónConPasoDeInformación '[] >>'*, y además cada una de estas tareas ha sido mapeada en un *AIC*, se establece una doble relación Observa *Observes* entre el primer *AIC* y el correspondiente atributo, y ua relación Observa *Observes* entre el segundo *AIC* y el mismo atributo.

Regla 8. Si dos tareas hermanas unidas por una relación *'manipulates'* con el mismo atributo, y estas tareas, en el modelo de tareas, están unidas por una relación temporal de tipo *ConcurrenciaConPasoDeInformación '|[]|'*, y además cada una de estas tareas ha sido mapeada en un *AIC*, se establece una doble relación *updates* y *observes* entre los dos *AICs* y el atributo que está mapeado a las tareas.

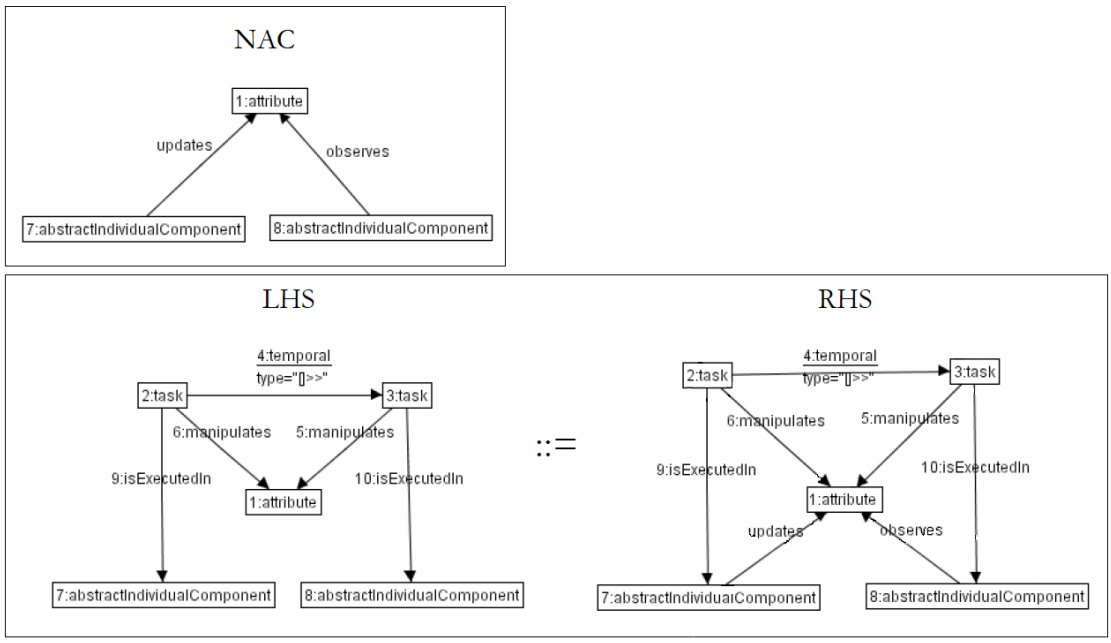

Figura 4.17: Derivación de relaciones Updates y Observes a partir de *ActivaciónConPasoDeInformación '[] >>'*.

4.8.2.9. *Transformación de AC (Abstract Container) a GC (Graphical Container) (reificación)*

Regla 9. Cada *Abstract Container (AC)* que está en el primer nivel de la jerarquía del árbol se reifica (transforma) en una ventana. Observar que un *AC* es siempre reificado en una caja del nivel concreto. Consiste en obtener el tipo de contenedor gráfico que le corresponde a cada abstracción. Este proceso de reificación (cosificación) presenta la dificultad de definir qué contención exacta corresponde a cada elemento gráfico de la interfaz final. Generalmente el *AC* que presente más contención será *Window* y los *AC* contenidos serán *Box*. Para la aplicación de esta transformación se requiere retroalimentación del árbol de tareas. Concretamente todas las tareas abstractas que estén en el nivel 1 del árbol (hijas de la raíz principal), no sean hojas y su *AC* asociado tenga otros *AC* contenidos dará lugar a un *Window*. Si el árbol de tareas solo tiene dos niveles (una raíz y varias hojas como hijas) la transformación es directa.

Regla 10. Cada *Abstract Container (AC)* contenido en un *AC* que es reificado (relación *isReifiedBy*) en una ventana es transformado en un área de contenido del tipo horizontal: *Box* dentro de la ventana.

4.8.2.10. *Selección de GIC (Graphical Individual Components)*

Regla 11. Cada *Facet* de tipo *Input* de un *AIC* da lugar a un *GIC* de tipo *EditableTextComponent* (TextBox). Se identifican los elementos gráficos, o *widgets*, (textBox, comboBox, Button, etc.) que son apropiados según los distintos tipos de facets (ver 4.1) que se pueden encontrar en el modelo de interfaz abstracto. A un facet de tipo Input y Element se le asociará un widget TextComponent, mientras que a uno de tipo Output y Collection le corresponderá un widget ComboBox.

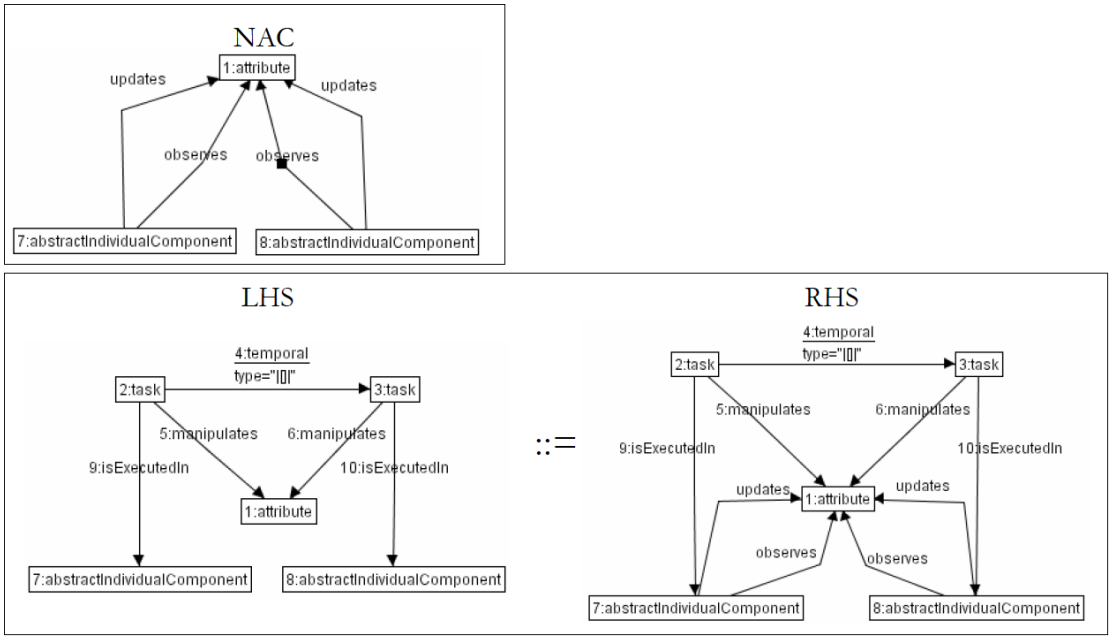

Figura 4.18: Derivación de relaciones Updates y Observes a partir de *ConcurrenciaConPasoDeInformación '|[]|'*.

4.8.2.11. *Disposición de Componentes Individuales Concretos. Obtención de GIC (Graphical Individual Components)*

Regla 12. Por cada pareja de *AIC* relacionados por medio de *'AbstractAdjacency'* y reificados en *GIC*, generar una relación *'ConcreteAdjacency'* entre los *GICs*. Con la información que aportan las relaciones *'AbstractAdjacency'* que existen entre los AIC, se establecen las relaciones *'Graphical Transition'* correspondientes entre los *GIC* reificados a partir de los *AIC*.

4.8.2.12. *Definición de la Navegación. (Efecto de la reificación de AC en ConcreteContainers)*

Regla 13. Para cada contenedor relacionado con otro contenedor, en distintas ventanas, crear un botón en el contenedor fuente que apunta a la ventana que tiene el contenedor destino. En este paso se obtiene el flujo de navegación que habrá entre contenedores que estén en distintos *Windows*, con el fin de establecer el orden correcto de aparición de los *Windows*.

4.8.2.13. *Definición del Diálogo de Control Concreto. (Especificación del flujo de control entre distintos objetos de interacción)*

Regla 14. Por cada pareja de *AC* con una relación *AUIDialogControl*, trasponer esta relación a la pareja de contenedores concretos *(Graphical Containers)* que reifican a la pareja de *AC*. Se buscan las correspondencias entre las relaciones *Dialog Control* abstractas y las propias del nivel concreto.

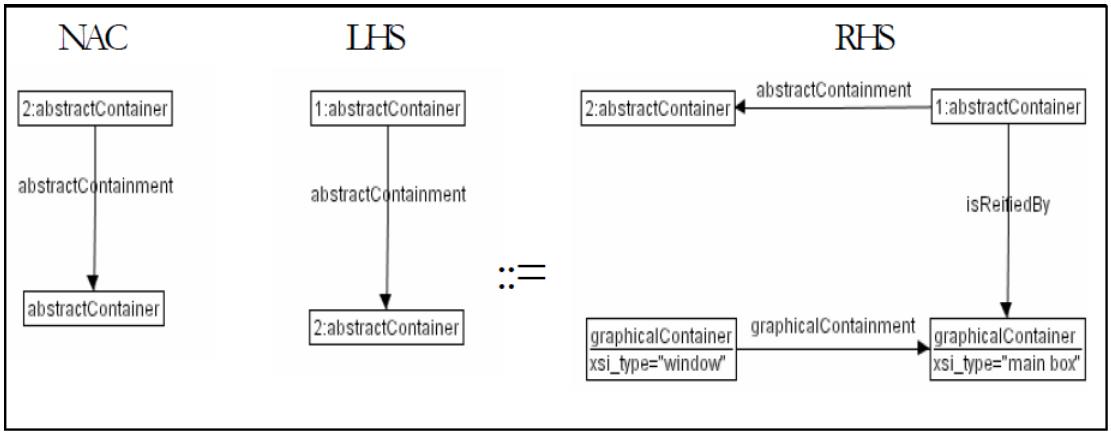

Figura 4.19: Creación de ventana a partir de relaciones de contención.

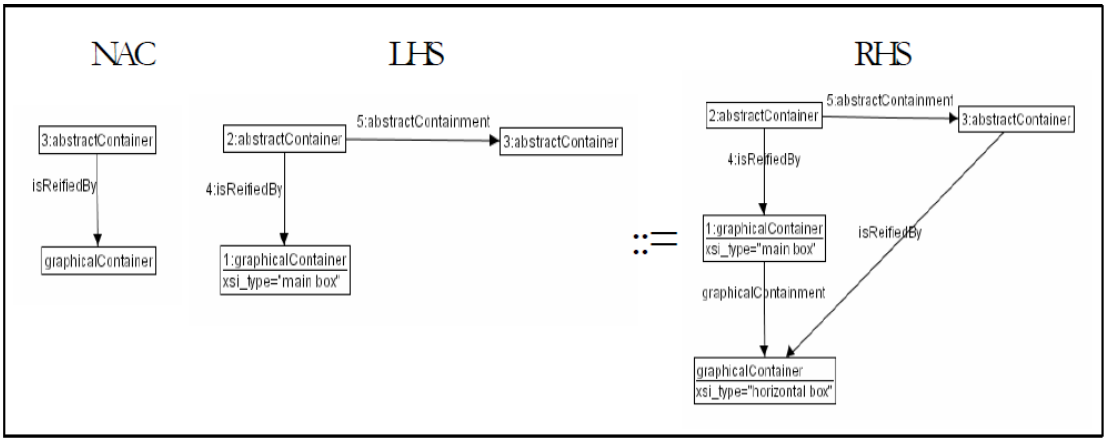

Figura 4.20: Creación de estructura de ventana a partir de sus AC.

4.8.2.14. *Obtención de CUI a partir de las relaciones de dominio*

Regla 15. Por cada *AIC* que establezca una relación *Updates* con algún elemento del dominio (attribute), si un *GIC* reifica este *AIC*, entonces este *GIC* también establece la relación *'Updates'* a ese mismo elemento de dominio (attribute). Se replican las relaciones *triggers* que se han establecido en el nivel abstracto en sus correspondencias concretas.

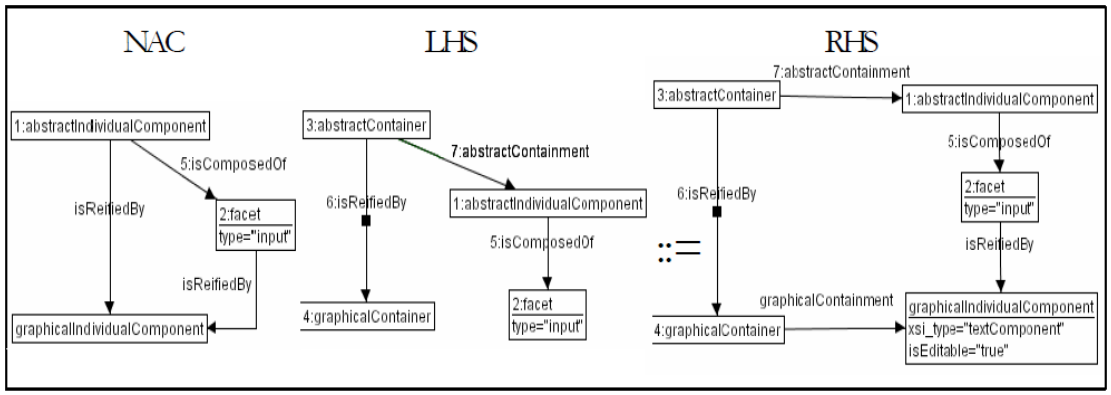

Figura 4.21: Creación de componentes gráficos a partir de sus facets.

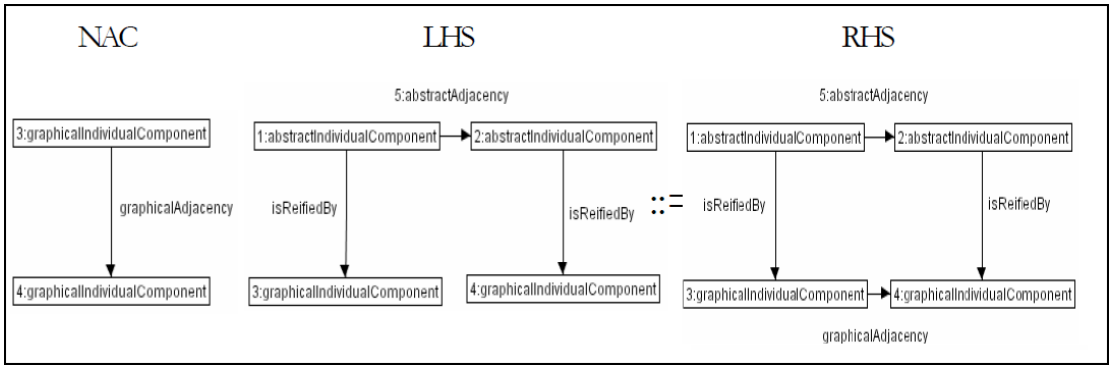

Figura 4.22: Creación de relaciónes *'ConcreteAdjacency'* y *'Graphical Transition'*.

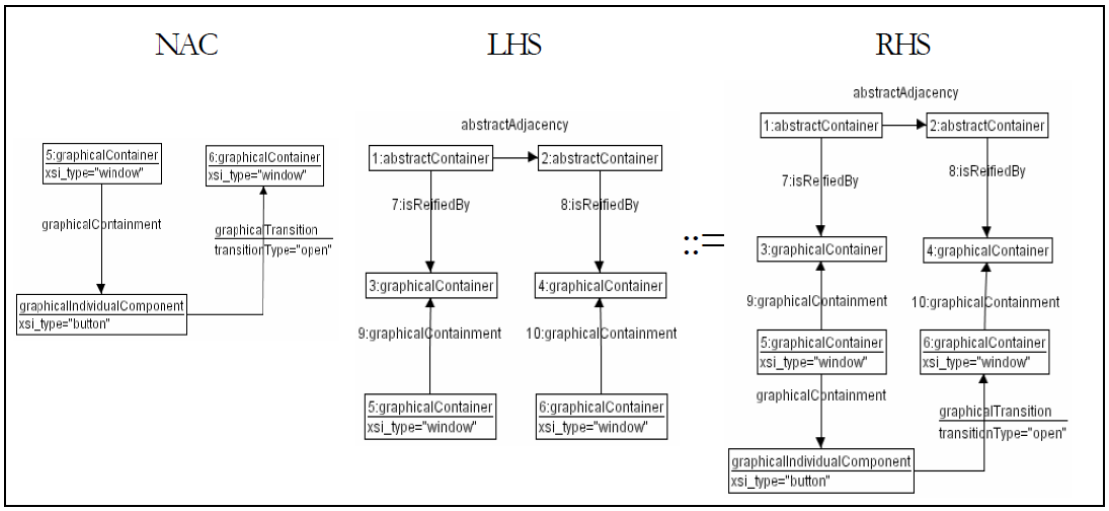

Figura 4.23: Definición de la navegación en distintos contenedores.

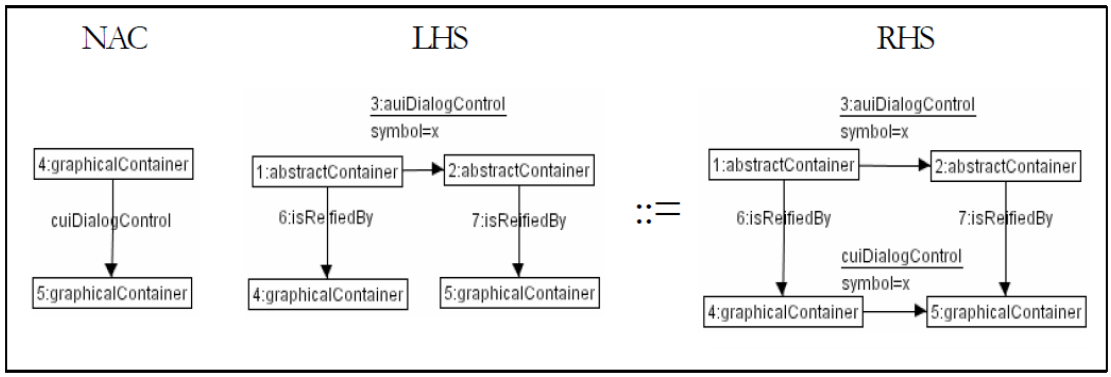

Figura 4.24: Definición del diálogo concreto.

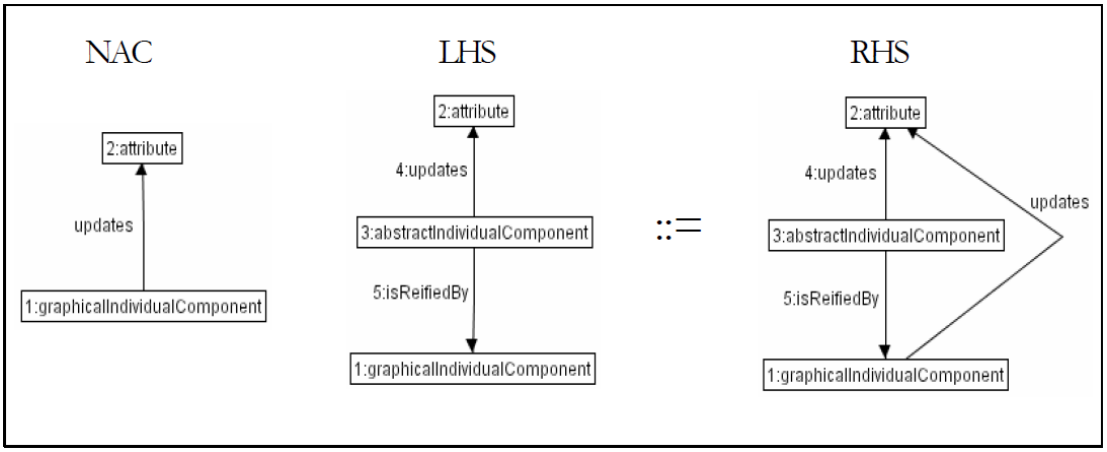

Figura 4.25: Creación de la relación *'Updates'*.

Capítulo 5

Validación de la propuesta mediante Caso de Estudio

En este capítulo se presenta la descripción del proceso de desarrollo de la interfaz de usuario para un caso de estudio en concreto. Para tal fin se aborda como caso de estudio un sistema para la reserva de un viaje aéreo y su respectiva compra. Se trata de un caso de estudio de media complejidad para facilitar la comprensión; pero también amplio como para mostrar la propuesta de desarrollo que persigue este trabajo de grado.

5.1. Enunciado Caso de Estudio

Este caso de estudio está basado en una situación común que se presenta a la hora de reservar y comprar un viaje aéreo. Las características de esta situación permiten aplicar un modelo de negocio sobre la herramienta propuesta, con el fin de estudiar los resultados obtenidos y validar el presente trabajo de grado. Una reserva de un viaje aéreo puede tener diversos puntos de partida, distintos lugares de destino y, también, diferentes precios. Un asesor de viajes puede sugerir, a un viajero interesado, una ruta con escalas o sin ellas. Dependiendo de esta selección el valor del tiquete podría verse afectado.

Para casos como éste y similares, no es realista que se desarrolle un sistema completo con estas características para una agencia de viajes, esto puede resultar bastante costoso existiendo muchos productos comerciales a disposición en el mercado. Sin embargo, los productos del mercado por lo general son bastante complejos y el personal debe recibir capacitación antes de que pueda realizar alguna tarea u operación sobre ese sistema. Para este caso de estudio se ha simplificado el modelo de negocio para minimizar la especificación. Por ejemplo, ignorar que las reservas y compras de un viaje pueden realizarse en línea; ignorar el chequeo de la disponibilidad de vuelos; no contemplar el equipaje aforado en bodega, el tipo de comida solicitado durante el trayecto; etc.

Dado lo anterior, los procesos más importantes de este tipo de negocio, podrían identificarse a continuación:

- Reservar viaje.
- Ingresar información: origen y destino.
- Seleccionar opciones.
- Facturar.

A continuación se presenta el desarrollo del caso de estudio siguiendo el enfoque MBUID (ver 2.1.3). Es importante mencionar que el desarrollo del caso de estudio se centra en el proceso *'Reservar Viaje'*. Primero se empezará modelando el diagrama CTT y el dominio del sistema, para después ir obteniendo productos intermedios (diagrama de transformación; interfaces *AUI* y *CUI*) y finales (*FUI*).

5.2. Diseño de la Interacción

Diagrama CTT. *Task Model*. Un modelo de tareas describe un ordenamiento lógico y temporal de las tareas que son realizadas por los usuarios en la interacción con un sistema. Especifica las principales tareas que se desarrollan en el negocio (entiéndase organización). Un modelo de tareas describe tanto las tareas actuales de los usuarios como la forma en que se pueden realizar en el futuro. Los elementos individuales en un modelo de tareas representan las acciones específicas que el usuario puede llevar a cabo. Para cada acción o tarea habitualmente se especifíca la siguiente información: meta de la tarea, opciones que permiten alcanzar el objetivo, resultado de la tarea, efectos secundarios o riesgos, frecuencia, duración, flexibilidad, exigencias físicas y mentales, dependencias, seguridad o cuestiones ambientales, etc.[26].

En este nivel, para representar la descomposición de las tareas, se usan los modelos de tareas CTT, ya que son muy apropiados para el diseño de interfaces usables[63]. La descomposición de tareas en subtareas de menor dificultad es el modo habitual de manejo de la complejidad que emplean los seres humanos en su trabajo.

Aunque la mayoría de las actividades son de tipo individual se supone que son actividades en las que existe una interacción con los clientes; sin embargo, solo se modela la interacción entre los usuarios del sistema. Muchas veces los usuarios desean ejecutar las tareas en un orden diferente, en el nivel de negocio se debe establecer un modelo que represente todas las posibles secuencias lógicas que serán llevadas a cabo dentro del dominio del sistema. Las tareas se describen de forma general y con independencia de la tecnología con la que se implemente la solución. En la figura 5.1 se observa el diagrama CTT que modela la tarea de un agente de viajes *'Reservar Viaje'*. En esta tarea el agente de viajes debe ingresar la información de origen del viaje, considerando la fecha de partida. Una vez se completen estas tareas, la siguente tarea es ingresar la información de destino. Una vez se hayan completado estas tareas previas, debe indicársele al sistema que despligue las distintas opciones disponibles para el trayecto deseado. Una vez el sistema muestre las opciones disponibles, el agente de viaje debe seleccionar, bien sea, la opción de mostrar por precio o la opción de mostrar por duración del trayecto. Una vez que se haya elegido la opción deseada por el cliente, el agente de viaje realiza la tarea de facturar la compra.

5.3. Diseño del modelo de Dominio

Es necesario definir el modelo de la información del dominio de una forma que sea independiente de cómo es utilizada por la interfaz. Este modelo es independiente de la representación de los datos que son almacenados en el sistema, aunque están relacionados. Se corresponde con el modelo de datos del sistema, que representa clases, atributos, métodos, objetos y relaciones del dominio, de manera que permite relacionar cada tarea presente en el Modelo de Tareas (ver 5.1) con los datos del sistema.

La figura 5.2 representa el modelo de dominio para el caso de estudio propuesto. Cada entidad corresponde a una colección de registros del mismo tipo. La entidad *Reserva* contiene un registro por cada reserva que se puede encontrar en el sistema. Al igual que la entidad *Vuelo* que tiene un registro por cada uno de los vuelos programados y realizados por las diferentes aerolíneas. Cada

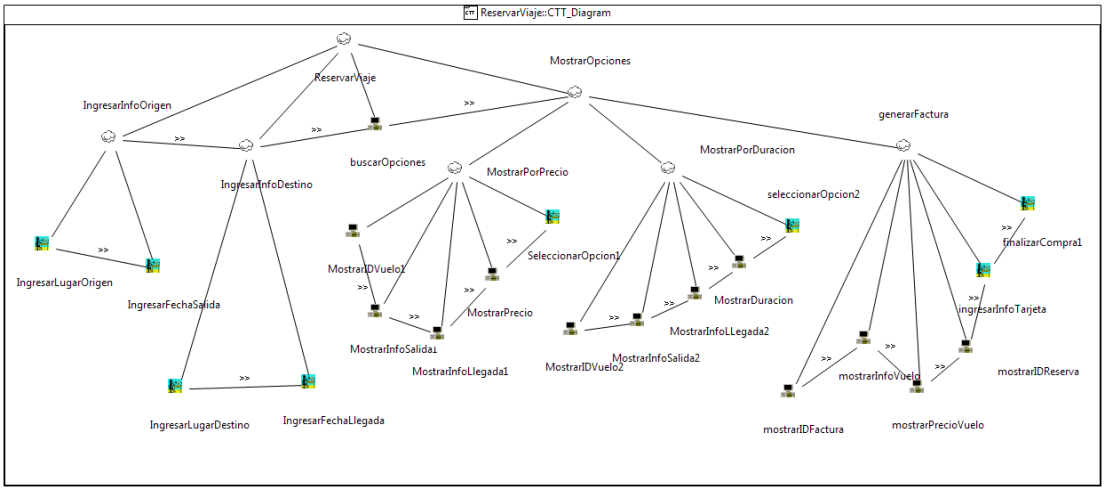

Figura 5.1: Diagrama CTT para tarea *ReservarViaje*

Reserva puede estar compuesta de varios *Vuelos*. Igualmente, cada *Reserva* es una colección de *Viajes* que ha podido realizar un *Viajero* al cual se le han *Facturado*.

El modelo de dominio se ha creado con la herramienta CIAT.TDMBUID al igual que los demás diagramas presentados en este trabajo.

Al finalizar el modelado de dominio y el modelado de las tareas se deben relacionar las tareas con los objetos que se manejan durante la interacción.

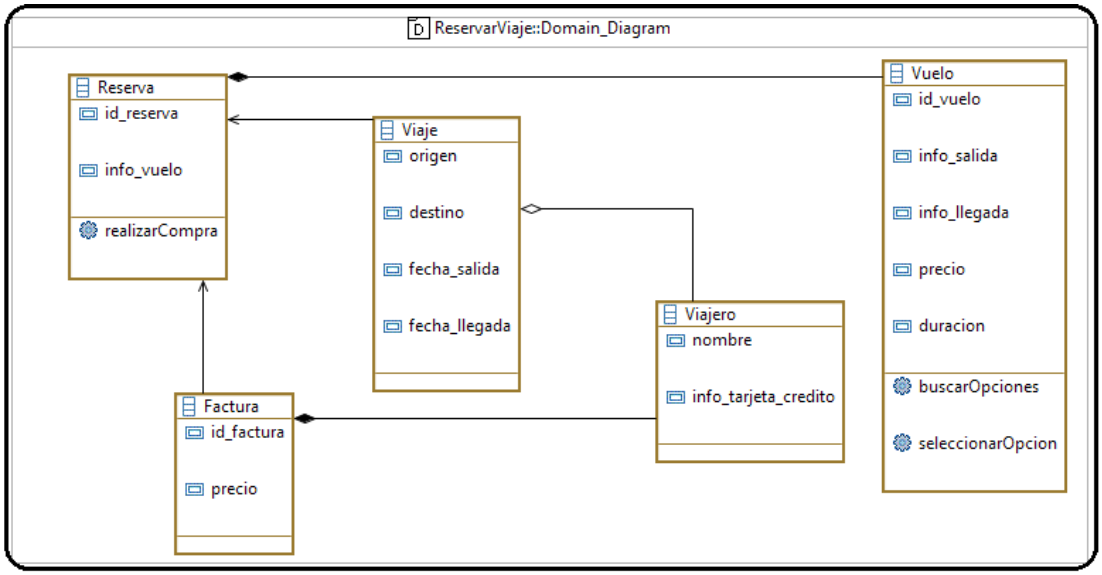

Figura 5.2: Representación del Dominio para caso de estudio propuesto

5.4. Diseño del modelo de Mapping

En el desarrollo de aplicaciones interactivas, es más importante comenzar por la identificación de tareas y, a continuación, los objetos relacionados. Una vez que se define el modelo de tareas se deben relacionar los objetos manipulados por la interfaz de usuario. En la Figura 5.3 se presenta el modelo de *Mapping* para la tarea de dominio *ReservarViaje*. Se puede observar el modo en que se relacionan, mediante lo que se ha denominado *'relaciones de trazabilidad'*, las tareas con los conceptos del modelo de dominio.

El modelo de *Mapping* estudia cómo la parte interactiva del sistema se relaciona con toda la información del dominio. La interfaz de usuario manipulará ciertos objetos del dominio, por tanto, entre las tareas (del modelo CTT ver figura 5.1) y la información (representada mediante el modelo de dominio ver figura 5.2) se establecerán relaciones de trazabilidad. La figura 5.3 contiene el mapeo completo establecido para la tarea *'ReservarViaje'*. Por ejemplo: la sub–tarea *'IngresarLugarOrigen'* se asocia al campo *'Origen'* de la clase *'Viaje'*, este vínculo se establece para relacionar dentro del diseño de la interfaz el componente gráfico y el modelo de dominio.

Desde el punto de vista de la interfaz, la ejecución de una tarea requerirá que se lleven a cabo una serie de acciones sobre los elementos de la interfaz, por ejemplo, un campo de texto puede ser leído o modificado, una función del sistema puede ser representada por un botón u otro elemento. De esta forma se agregan nuevas funciones, métodos, en el modelo de dominio del sistema; esto se puede observar en la tarea *'FinalizarCompra'* la cual requiere el soporte del método *'RealizarCompra'* de la clase *'Viaje'* como se ilustra en la figura 5.3.

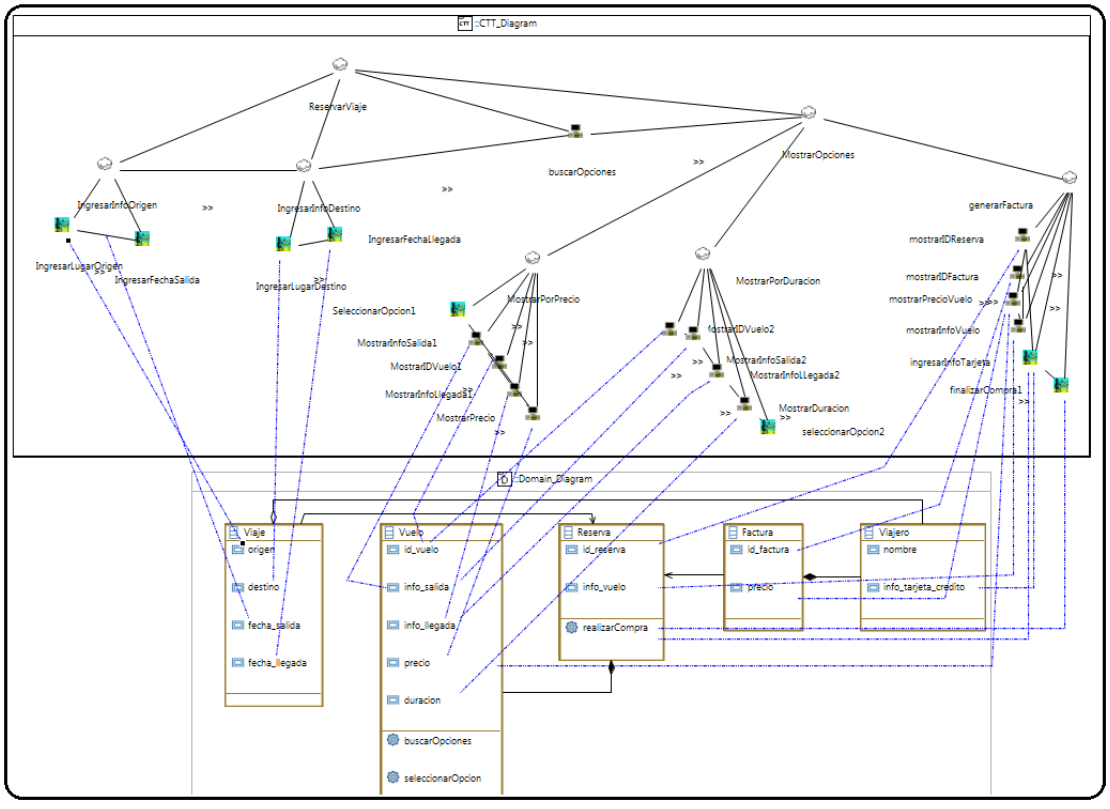

Figura 5.3: Modelo de Mapping para la tarea *ReservarViaje*

La configuración de las acciones que llevan a cabo las tareas *'IngresarLugarOrigen'*, *'MostrarIDVuelo1'*, *'finalizarCompra1'* se muestra en la figura 5.4. Esta configuración se ha de realizar en este nivel de abstracción para que, posteriormente, se puedan obtener los elementos que forman el modelo de interfaz abstracta. Para la tarea *'IngresarLugarOrigen'* se requiere que la interfaz de usuario permita seleccionar un lugar como un único elemento *'(Element, Select)'*. Para la tarea *'MostrarIDVuelo1'* se requiere que la interfaz de usuario permita ver una identificación del vuelo como un único elemento *'(Element, View)'*. Finalmente, para la tarea *'finalizarCompra1'* se requiere que la interfaz de usuario permita activar un método mediante un componente de la interfaz *'(Operation, Start_Go)'*.

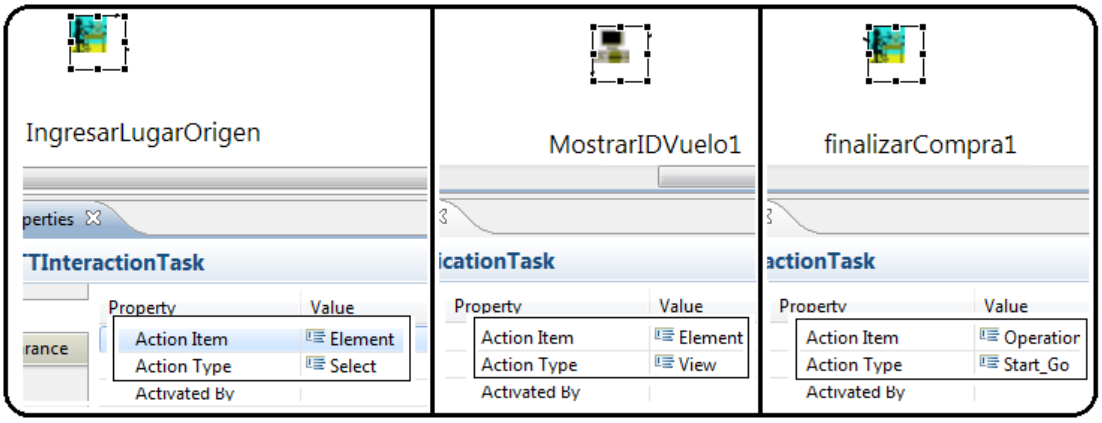

Figura 5.4: Configuración de tareas del modelo CTT de acuerdo a la acción que llevan a cabo sobre el modelo de domino.

De esta forma el modelo de interacción representa el *mapping* que existe entre las tareas y los datos que son necesarios para soportar dichas tareas. Además, representa el vínculo encargado de establecer la conexión entre el desarrollo de la interfaz de usuario y la funcionalidad del sistema y representa la definición de múltiples facetas que pueden ser utilizadas para la especificación de los componentes de la interfaz de usuario[26]. Se ha mostrado como el *modelo de mapeo* creado en esta etapa representa la relación existente entre las tareas y los datos que éstas manejan. Este modelo establece el vínculo encargado de conectar la interfaz de usuario con la funcionalidad del sistema.

5.5. Obtención de la Interfaz de Usuario Abstracta

Una vez que se han definido los anteriores modelos para el caso de estudio se procede a realizar la transformación de modelo a modelo. El resultado de esta acción se aprecia en la figura 5.5. Se puede observar que el contenedor abstracto *ReservarViaje* fue propuesto a partir de la tarea abstracta (en el modelo de tareas CTT) del mismo nombre, aplicando la regla de transformación ver 4.8.2.4 *Regla 1*. Una vez identificado el AC *ReservarViaje* se generó el AC *IngresarInfoOrigen* a partir de la tarea asbtracta de su mismo nombre, ver figura 5.1 aplicando la regla de transformación ver 4.8.2.4 *Regla 2*. Las reglas de transformación 4 y 5 (ver 4.8.2.6 y 4.8.2.7 respectivamente) se aplicaron para lograr las relaciones de diálogo y adyacencia entre los AIOs. De esta manera se entrega parte del control y navegación entre los distintos componentes presentes en la generación de la Interfaz de Usuario Abstracta.

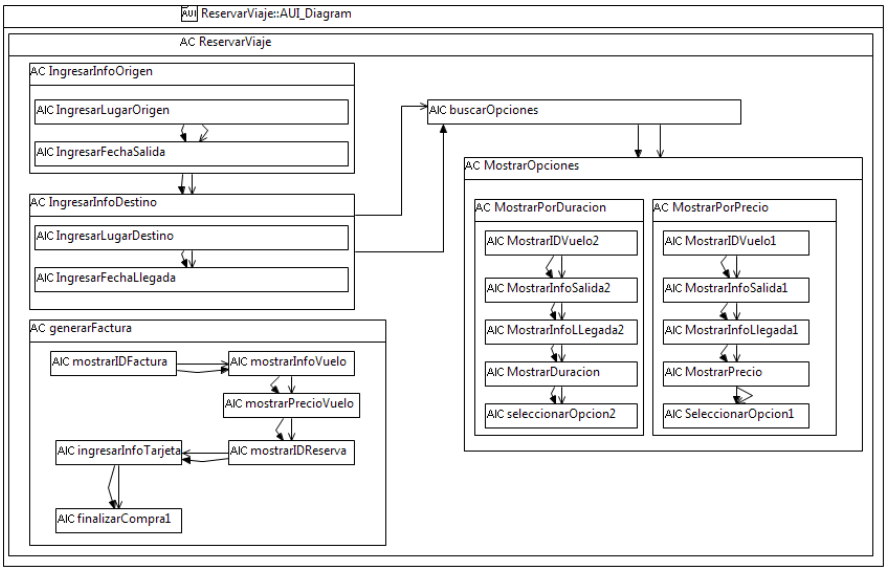

Figura 5.5: Interfaz Abstracta de Usuario: *ReservarViaje*.

5.6. Obtención de la Interfaz de Usuario Concreta

La obtención de cada ventana presente, específicamente, en este caso de estudio, se dio a partir de la aplicación de la regla 9 (ver regla 4.8.2.9). Una vez se obtienen las ventanas, cada componente concreto, presente en este ejemplo, se obtiene a partir de la regla de transformación 11 (ver 4.8.2.10).

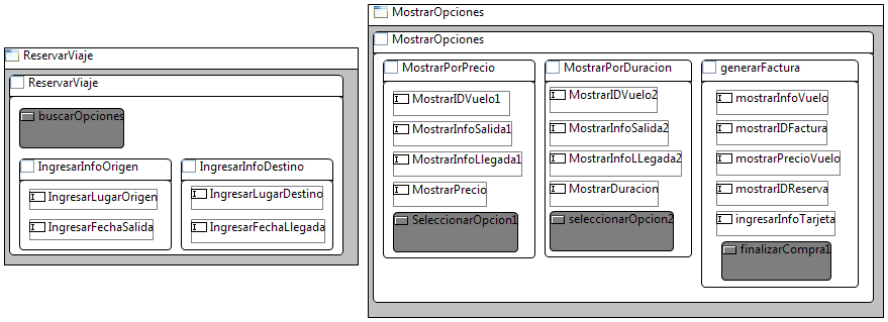

Figura 5.6: Interfaz Concreta de Usuario: *ReservarViaje*.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

El objetivo general de este trabajo (ver 1.3.1) es desarrollar un metamodelo que apoye la herramienta CIAT.TDMBUID. Para ello se han estudiado los trabajos previos[26],[25],[22],[16]. El estudio realizado ha permitido generar una nueva versión, no solo del metamodelo sino, también, de la herramienta. Esta herramienta genera la Interfaz de Usuario en código *XAML* y la representación interna de los modelos está implementada en UsiXML[83].

La realización de este trabajo dota al desarrollador de sistemas software de una herramienta para la generación automática de interfaces gráficas de usuario, basada no solo en el conocimiento de los datos manejados en el negocio, sino en la manera y orden en que se ejecutan las tareas que manipulan esos datos, obteniendo una especificación detallada de la interfaz gráfica y aportando a los desarrolladores de software el modelo de datos y el modelo de tareas.

Este trabajo ha desarrollado la especificación de un metamodelo con el fin de apoyar y extender la propuesta *TD–MBUID*[26], con el propósito de integrar distintas notaciones, tanto en la sintaxis abstracta como en la sintaxis concreta, de manera que al ejecutar el nuevo editor de diagramas y modelos (herramienta *ciat.tdmbuid*) el usuario desarrollador encuentre a su disposición las herramientas y diagramas requeridos para expresar la estructura en datos de un negocio y su modelo de ejecución y realización de tareas con el propósito de contribuir a la generación de interfaces gráficas de usuario a partir de la generación y transformación de modelos. Este trabajo hace uso intensivo de metamodelado, de la propuesta de Limbourg[42] y de la propuesta de Giraldo[26]. En las anteriores propuestas se plantea el esquema de generación y obtención de Interfaces de Usuario Finales. Este trabajo ha seguido ese planteamiento y ha reunido en un metamodelo las distintas notaciones (CIAN[85] y UML[58]) y lenguajes (usiXML[83]). En este trabajo se conformó la nueva sintaxis abstracta (ver sección 4.1) y sintaxis concreta del lenguaje y la herramienta *CIAT.TDMBUID* (ver secciones 4.4,4.5,4.6) teniendo en cuenta la incorporación del diagrama de transformación dando cumplimiento al objetivo 1.3.2.1.

La especificación de la sintaxis concreta fue ardua pues, como se considera en [29], solo disponer de un editor en forma de árbol o sino directamente la sintaxis de XML, convierte este proceso de especificación en una tarea dispendiosa. Sin dejar de mencionar que cualquier cambio realizado a la sintaxis abstracta debe verse reflejado en la sintaxis concreta, haciendo estos cambios/actualizaciones bastantes difíciles de editar.

El aporte de este trabajo de grado es proponer un sistema de transformaciones entre modelos, es decir, una vez que se hayan definido modelos de tareas y modelos de datos, junto con modelos de trazabalidad, el usuario pueda seleccionar el algoritmo deseado para realizar la transforma-

ción a partir de la referencia a sus diagramas. Esta propuesta de transformaciones incluye los algoritmos de Limbourg[42] y de Luyten[46]. La intención de este nuevo diagrama es comunicar que transformación se aborda en el diagrama. Este diagrama contiene, por dentro, referencias a otros tipos de diagramas. El concepto del diagrama de transformación es usar referencias hacia los demás diagramas.

También en este trabajo de grado se realiza aporte sobre la sintaxis abstracta y sintaxis gráfica del marco conceptual de la notación CIAN[51]. Este aporte consiste en presentar cinco nuevas tareas sobre el Modelado de la interacción (simple cooperativa, nueva cooperativa, simple individual, nueva individual e Individual Interaction (interactive)).

6.2. Trabajo Futuro

Una futura línea de investigación podría ser encontrar una solución a la navegación entre pantallas resultantes (*Adicionar Control de Diálogo*) y poder establecer un método para lograrlo sería una importante contribución.

Ampliar la generación de FUI en otros lenguajes de programación.

Adicionar a la transformación la creación de la trazabilidad para los demás operadores temporales distintos de *AggregationTransition* y *EnablingTransition*.

Poder aplicar recursividad al algoritmo de transformación para que pueda estar capacitado para tomar una decisión de cuál es la tarea padre: *Maintask* si el diagrama CTT no la trae definida.

En el metamodelo de CIAT.TDMBUID debe implementarse una función de remapping. Podría considerarse establecer nuevas relaciones a nivel abstracto, y también concreto, que soporten el remapping tanto en el metamodelo como en la herramienta.

Realizar la transformación de modelo a código (M2T) para poder obtener la interfaz final de usuario (FUI), dado que en este trabajo se elaboró el algoritmo pero no se logró obtener el resultado deseado.

Anexos A

Metamodelo CIAT.TDMBUID

A.1. Paquete Model Management

El paquete *'Model Management'* tiene como objetivo la definición de las metaclases necesarias para la gestión de los modelos y los elementos de modelo. Desde el punto de vista de la organización de los elementos de modelado, la principal metaclase es la de *'Namespace'*. Esta metaclase tiene como objetivo la definición de una estructura de datos jerárquica para identificar la ubicación exacta de cada abstracción dentro de un modelo. El nombre de una abstracción (elemento de modelo) dentro del modelo se genera combinando su nombre (atributo *'name'* de *'ModelElement'*) con la cadena construida jerárquicamente por medio de combinaciones de *'Namespace'*. Dicha cadena se genera por medio del atributo *'name'* heredado de *'ModelElement'*. En términos generales, todo elemento de modelo que se crea en un modelo es propiedad de un *'Namespace'*. Para el caso de las metaclases *'ElementReference'* y *'ElementOwnership'* se hizo una variación en el metamodelo. Estas metaclases soportan las referencias y la visibilidad entre espacios de nombres. El principal uso de estas metaclases es la de gestionar la organización de elementos de modelado.

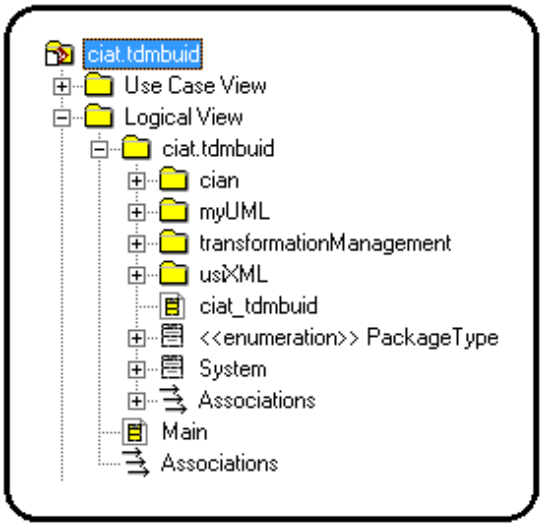

Figura A.1: Modelo CIAT.TDMBUID

La clase *'System'* (ver A.1) se define en este nivel (raíz) para efectos de organización de los modelos EMF[81]. De esta forma, cada que se inicia un modelo se obliga al desarrollador (di-

señador) a definir primero la información principal del sistema que se diseña. A partir de la clase *'System'* se inicia la instanciación de los demás elementos de modelado que se requieran para completar el sistema.

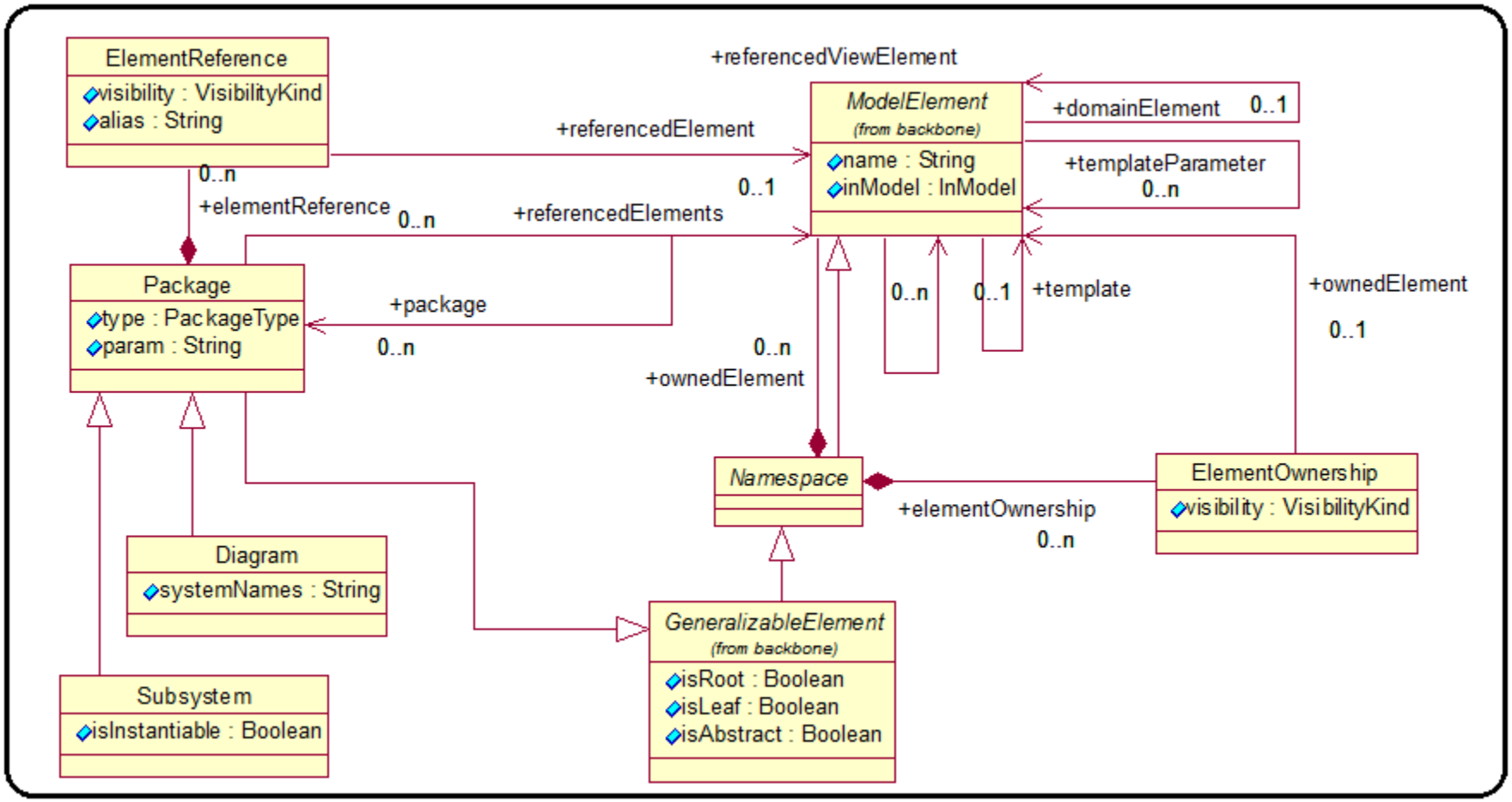

Figura A.2: Gestión de Diagramas. Paquete *Model Management*.

A.2. Modelo de Dominio

El paquete *'backbone'* define las metaclases para describir el modelo de dominio. Los objetos pueden describir tanto sus atributos como sus operaciones y definir sus respectivas cardinalidades, al igual que las relaciones entre las clases.

La metaclase más básica dentro del metamodelo es *'Element'*. Por medio de esta metaclase se puede identificar cada una de las abstracciones que participan en un modelo por medio de un identificador único. La siguiente metaclase en importancia, desde el punto de vista de la organización e identificación de abstracciones, es la metaclase *'ModelElement'*. Un *'ModelElement'* es un elemento que es una abstracción procedente del sistema que está siendo modelado. A partir de esta metaclase se definen todas las características que son comunes a los elementos de modelado dentro de UML. El principal atributo de *'ModelElement'* es *'name'* para dar nombre a cada abstracción.

Un *'Classifier'* tiene características tanto estructurales como dinámicas, de ahí que su principal metaclase sea *'Class'*.

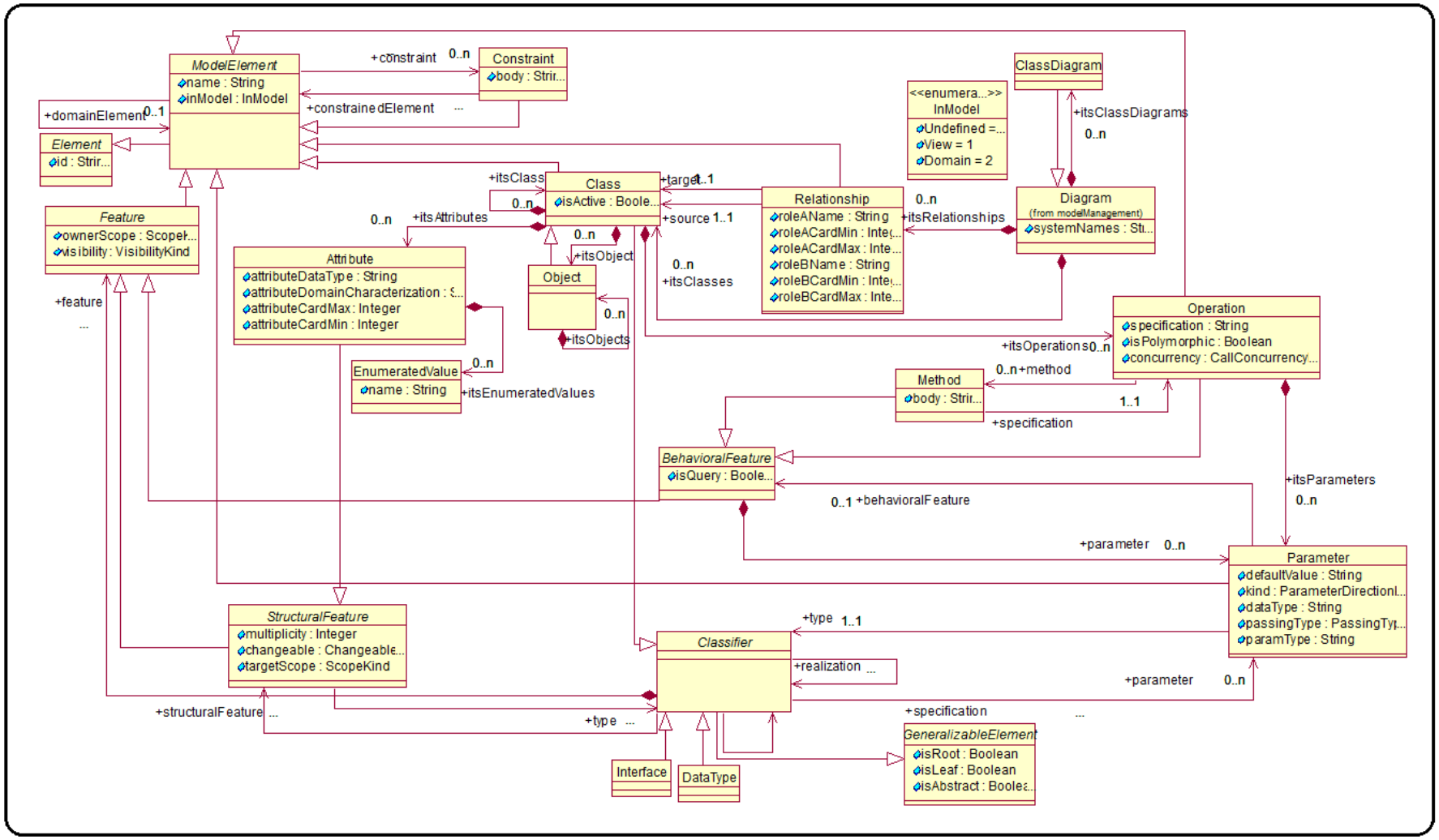

Figura A.3: Paquete *backbone*. Modelo de Dominio para el editor implementado

A.3. Modelo CIAN.core

La metaclase *'Task'* es la principal abstracción para el modelado de la actividad en CIAN, por esta razón posee la mayor cantidad de atributos y referencias que son típicas de este tipo de modelado. Las interdependencias entre tareas son en principio del mismo tipo que en los diagramas CTT, identificadas por medio de la enumeracin *'Operator'*. Cabe anotar que todas estas clases son del tipo *'ModelElement'*, por lo tanto, cada una de estas metaclases están en la posibilidad de pertenecer a un modelo.

La metaclase *'WorkProduct'* se define para generalizar las distintas abstracciones que forman parte de la entrada o salida de una tarea. La metaclase *'People'* se define para generalizar todas las abstracciones asociadas a personas, roles o sistemas que participan en una tarea. La metaclase *'Tool'* son los sistemas o herramientas que soportan las tareas. Las metaclases *'Initial'* y *'Final'*son definidas para describir las tareas de inicio y parada de los diagramas de Inter-Acción. Estas metaclases aunque tengan igual representación visual que en los diagramas de actividad representan abstracciones distintas. La metaclase *'Task'* tiene un conjunto de relaciones que facilitan la navegación de los modelos a la hora de proveer servicios a los desarrolladores. Por ejemplo: las relaciones *'next'* y *'previous'* identifican las tareas que están ubicadas de forma consecutiva, a pesar de que, la metaclase para relacionar las tareas es *'Interdependency'*. En una transformación es más simple acceder a estas relaciones para identificar una tarea que buscar en todas las interdependencias.

Del mismo modo, las relaciones *'father'* y *'subtask'* facilitan la identificación jerárquica de las tareas de padre a hijo y viceversa.

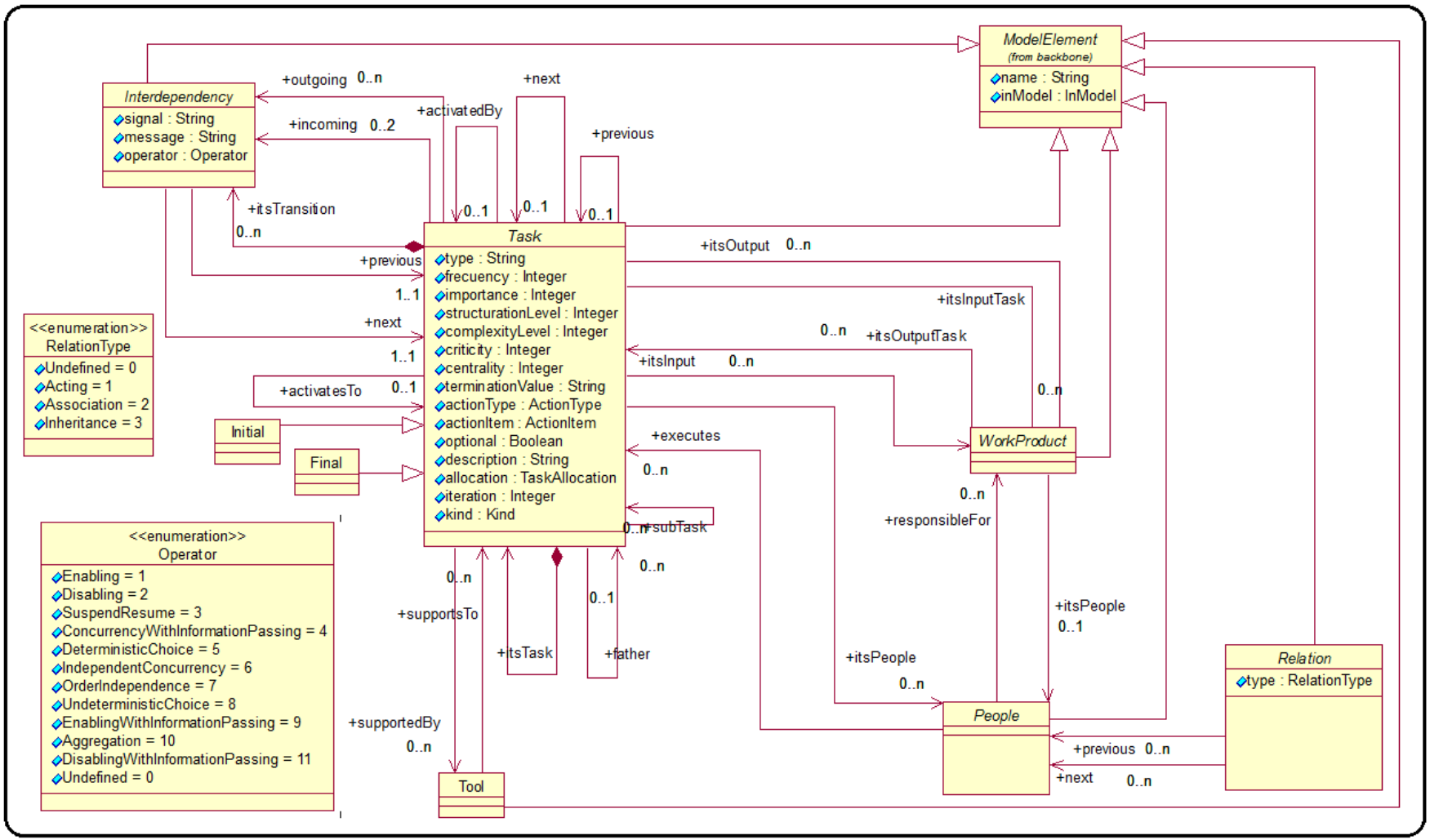

Figura A.4: Paquete CIAN.core

A.4. Paquete CTT

Este paquete define las tareas de tipo CTT y los tipos de datos de configuración de los distintos tipos de tareas CTT. Define las metaclases para el modelado de las tareas CTT y sus relaciones.

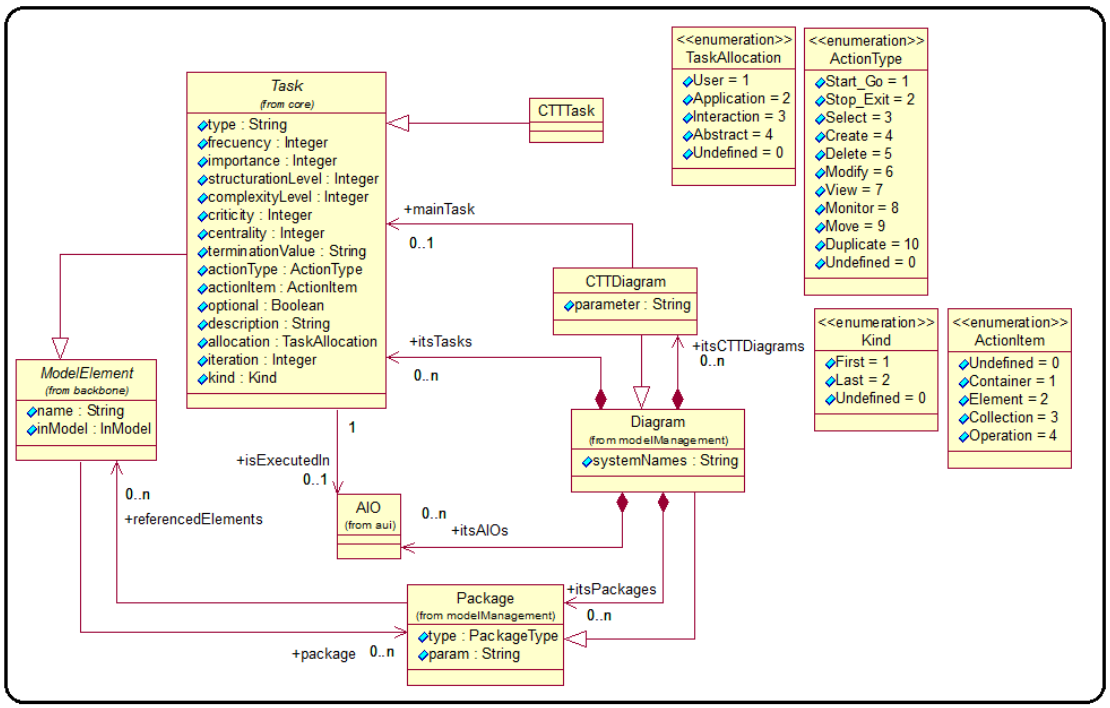

Figura A.5: Paquete CTT.

En la figura A.6 se observa que el paquete *'CIAN.Concrete.CTT'* tiene las definiciones básicas para implementar la sintaxis concreta de CTT.

A.5. Paquete AUI

Define las metaclases para describir la Interfaz de Usuario Abstracta. Se destaca el clasificador *'Facet'* que se encarga de soportar distintos tipos de interacción entre los componentes individuales abstractos y los usuarios. Por medio de este clasificador se implementan los objetos de interacción abstractos. Cada componente individual abstracto tiene la posibilidad de tener asociado múltiples facetas (*'Facet'*). De esta forma se puede capturar la información necesaria para soportar diversas interacciones asociadas a un mismo componente de la interfaz de usuario.

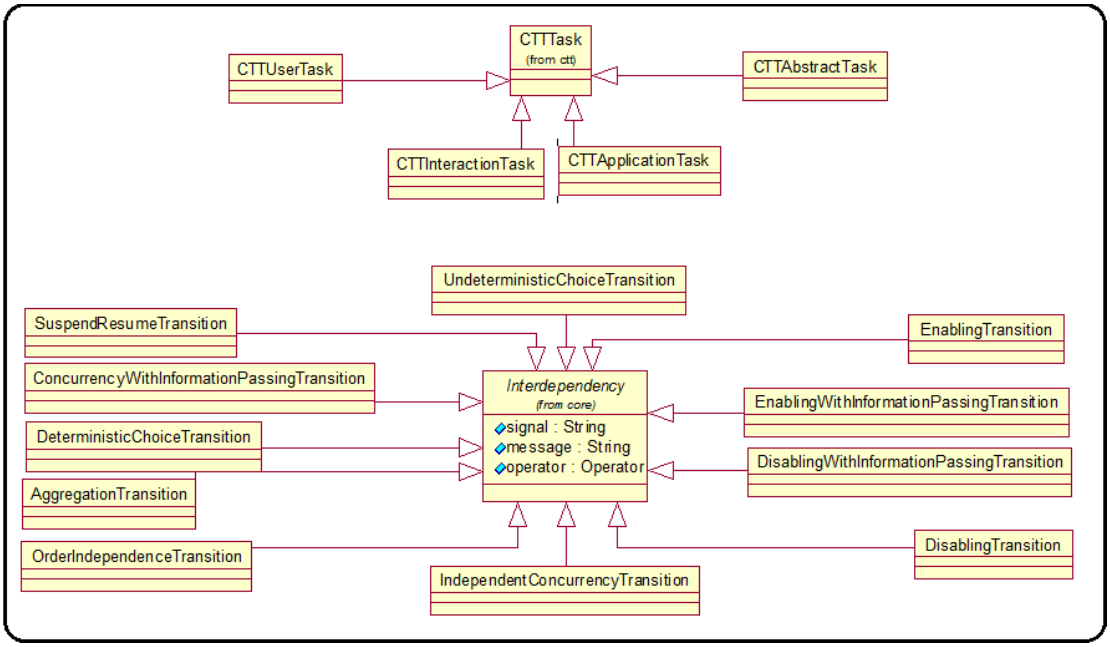

Figura A.6: Tipos de Tareas CTT.

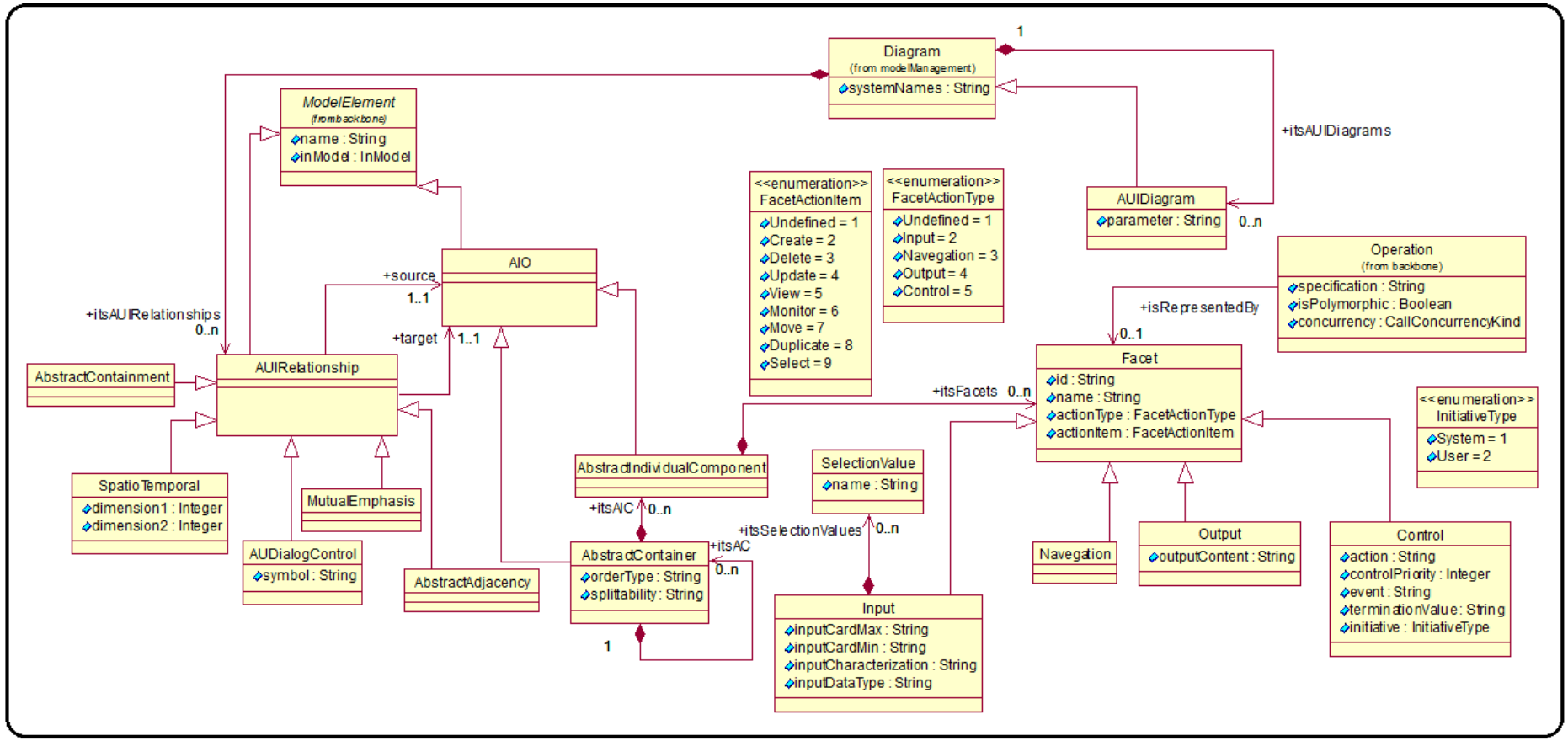

Figura A.7: Modelo AUI para el editor implementado

A.6. Paquete CUI

En este paquete se definen tanto los componentes concretos como las relaciones que intervienen en dichos diagramas de la interfaz de usuario.

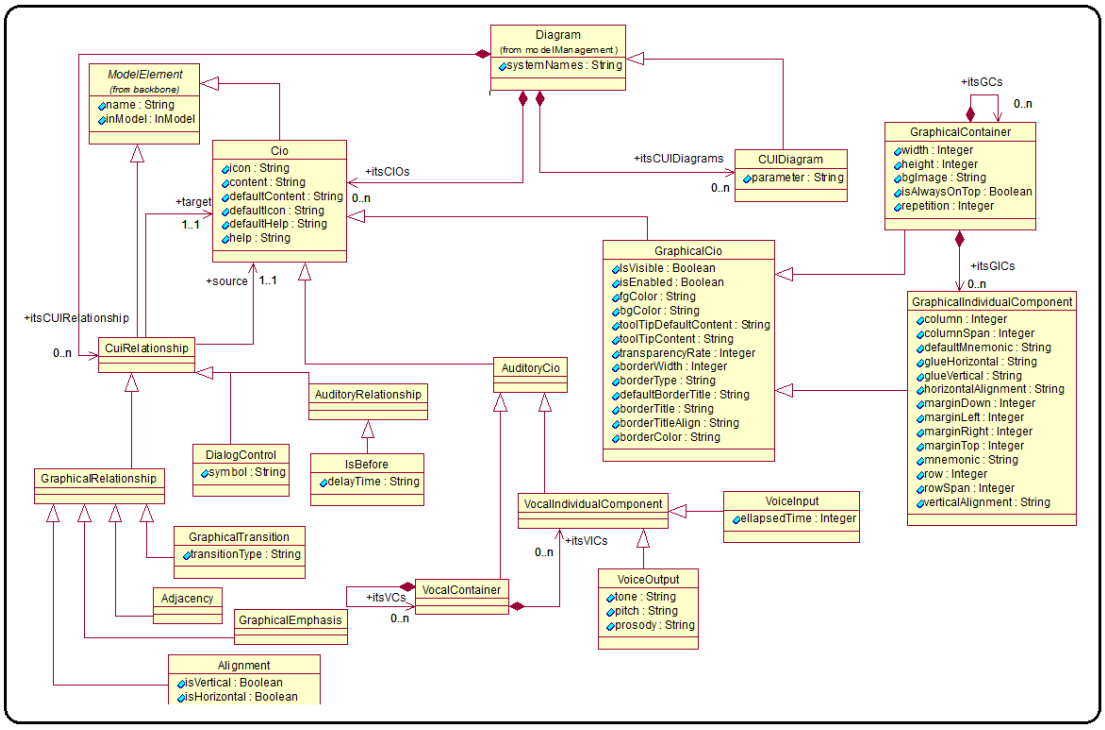

Figura A.8: Paquete CUI para el editor implementado.

A.7. Paquete Mapping

El paquete *'Mapping'* define las metaclases necesarias para describir las relaciones de mapping que permiten estructurar la información que hace posible la trazabilidad entre modelos. Esta trazabilidad provee la información necesaria para soportar el patrón Modelo–Vista–Controlador que se implementa entre componentes de la interfaz de usuario y objetos de la aplicación.

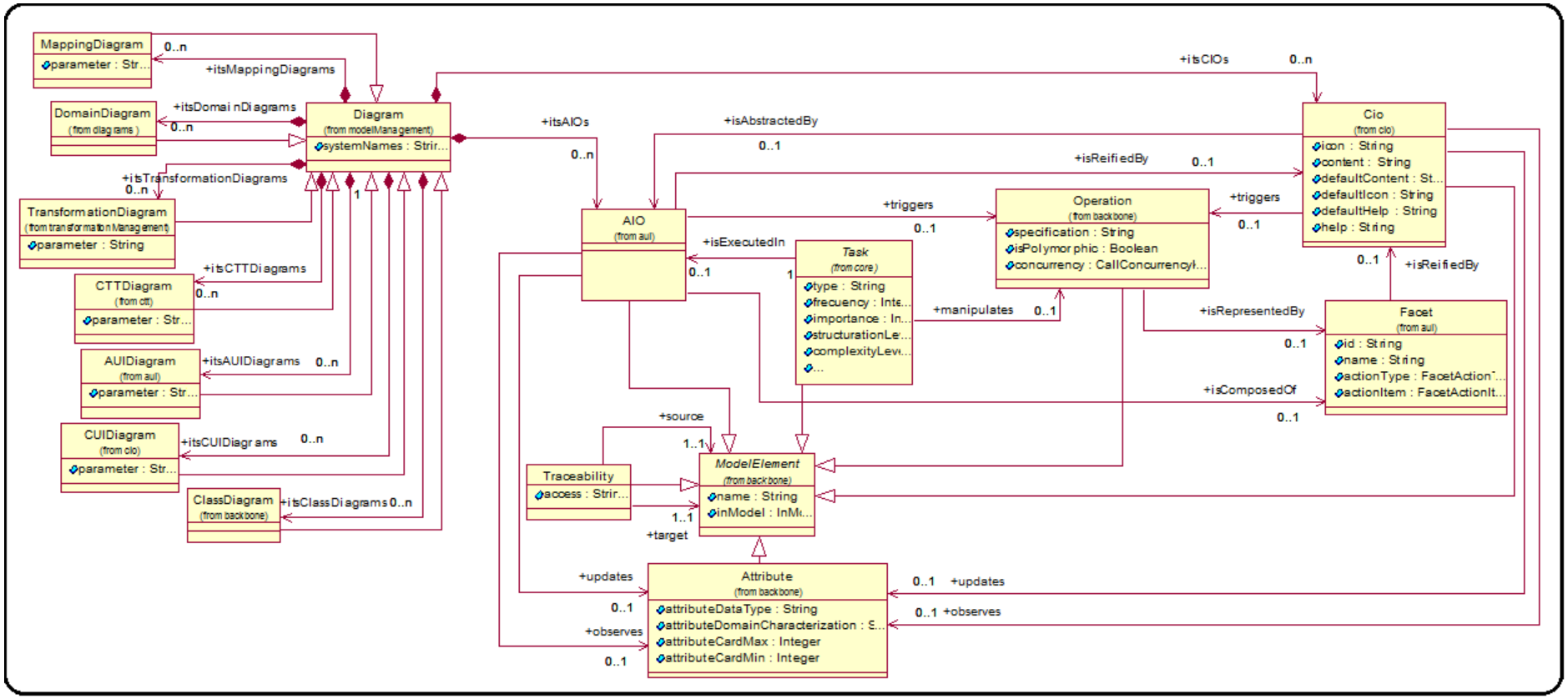

Figura A.9: Paquete Mapping

A.8. Paquete TransformationManagement

La intención del paquete de transformación es poder gestionar las transformaciones entre modelos. La clase *'System'* (ver A.10) se define en este nivel (raíz) para efectos de organización de los modelos EMF[81]. A partir de la clase *'System'* se inicia la instanciación de los demás elementos de modelado que se requieran para completar el modelo de transformación.

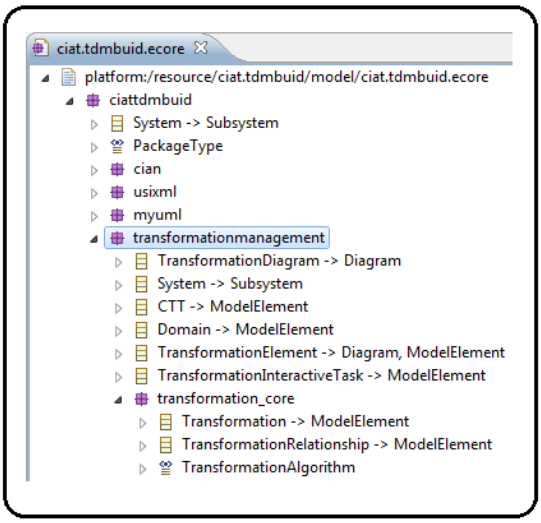

Figura A.10: Modelo de Gestión de Transformaciones para CIAT.TDMBUID

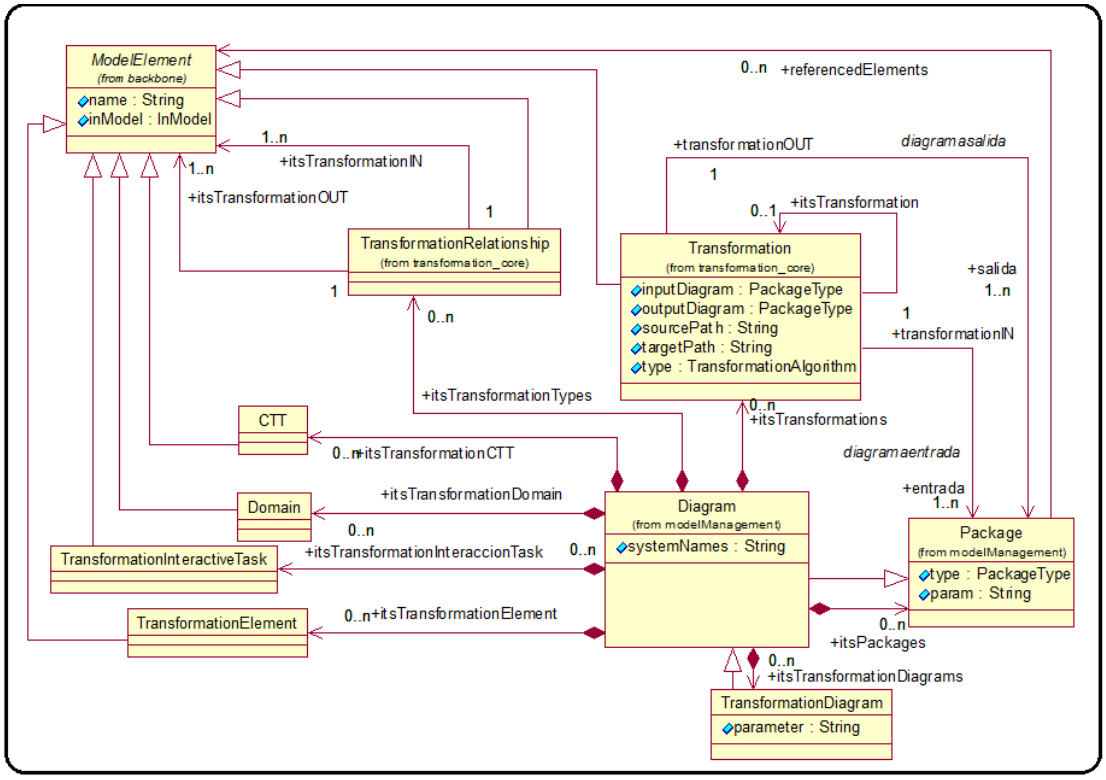

Figura A.11: Modelo de Contención para Transformaciones

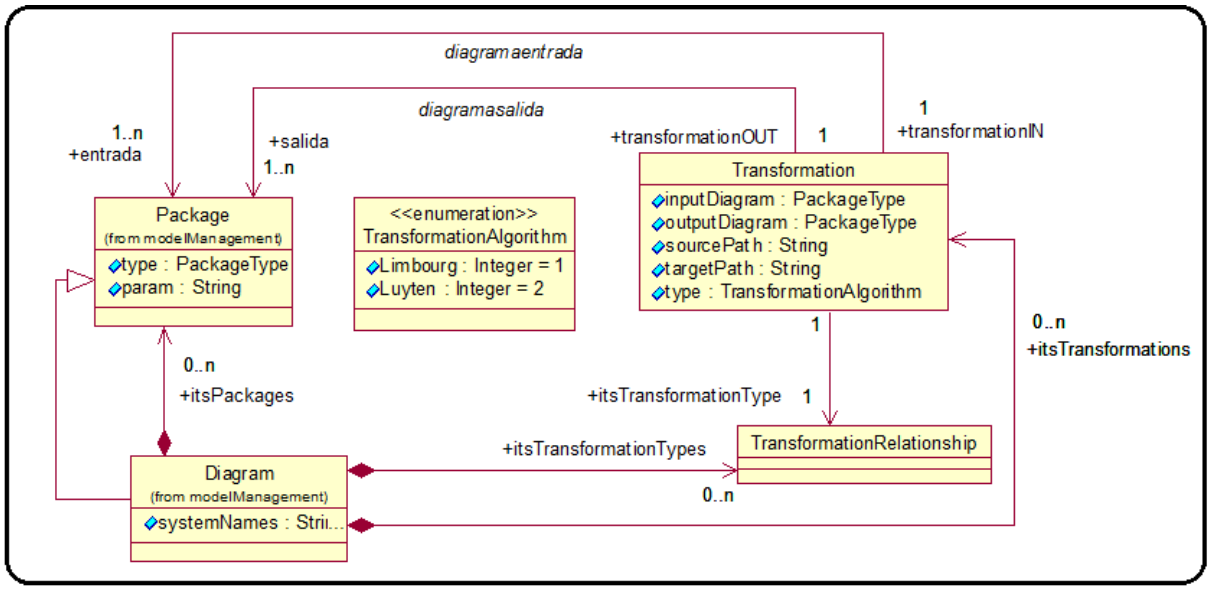

Figura A.12: Modelo de Transformación

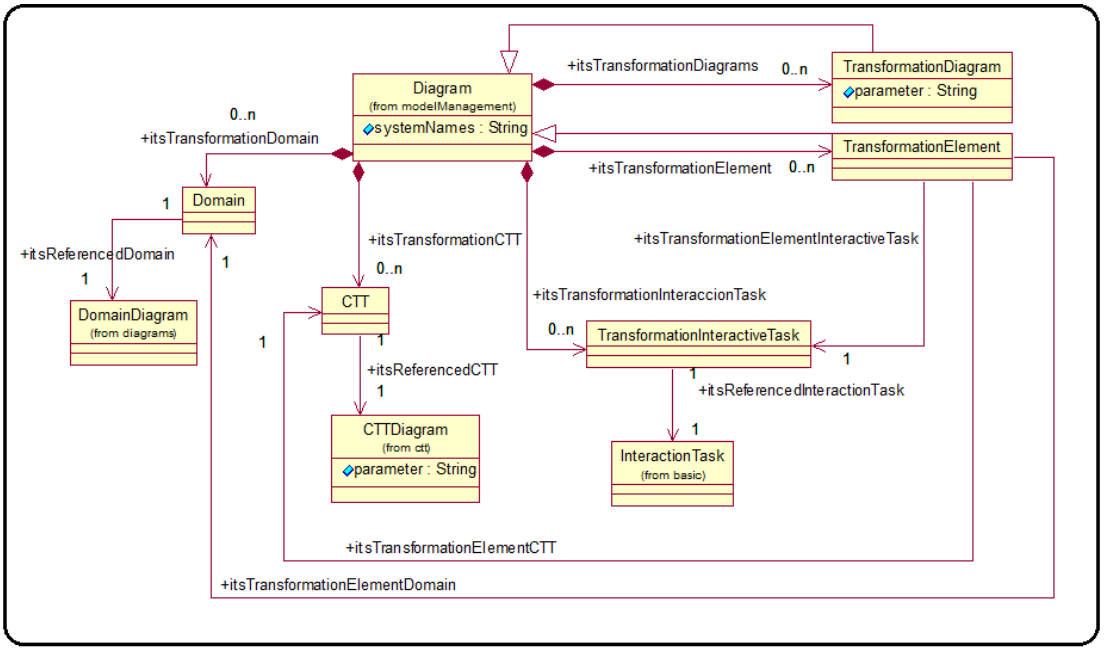

Figura A.13: Modelo de Transformación de Elementos

Bibliografía

[1] Amaya Pablo; González Carlos; Murillo Juan M. Separación de aspectos en mda: Una aproximación basada en múltiples vistas. In *Actas del I Taller sobre Desarrollo Dirigido por Modelos MDA y Aplicaciones (DSDM'04)*, 2004.

[2] Amaya Pablo; González Carlos; Murillo Juan M. Aspectmda: Hacia un desarrollo incremental consistente integrando mda y orientado a aspectos. In *Actas del II Taller sobre Desarrollo Dirigido por Modelos MDA y Aplicaciones (DSDM'05)*, 2005.

[3] Anacleto Valerio Adrin. Arquitectura dirigida por modelos. *Revista Code*, (31):60–64, 2006.

[4] Bärisch, S. *Domain-Specific Model-Driven Testing*. Software Engineering Research. Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2009.

[5] Beydeda, S. and Book, M. and Gruhn, V. *Model-Driven Software Development*. Springer, 2010.

[6] Bèzivin Jean; Farcet Nicolas; Jèzèque Jean-Marc; Langlois Benoît; Pollet Damien. Reflective model driven engineering. In *Proceedings of UML 2003*, volume 2863 of LNCS, pages 175–189. Springer, 2003.

[7] Boca Paul, Bowen Jonathan P, Siddiqi Jawed I, Editors. *Formal Methods: State of the Art and New Directions*. Springer London, 2010.

[8] Buschmann Frank; Meunier Regine; Rohnert Hans; Sornmerlad Peter; Stal Michael. *Pattern-oriented software architecture: A system of patters*. John Wiley&Sons, 1999.

[9] C. M. Sperberg-McQueen and Henry Thompson. *XML Schema*, 2000. XML Schema.

[10] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[11] Cesar De la Torre, Unai Zorrilla, Miguel Ángel Ramos, Javier Calvarro. *Guía de Arquitectura NCapas orientada al Dominio con .NET 4.0*. Krasis Consulting S.L., 2011.

[12] Clarke Siobhán; Harrison William; Ossher Harold; Tarr Peri. Subject–oriented design: Towards improved alignment of requirements, design and code. In *Object–Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.

[13] Czarnecki Krzysztof; Helsen Simon. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model– Driven Architecture*, 2003.

[14] Daniel Exertier; Benoît Langlois; Xavier Le Roux. Pim definition and description. In *Proceedings First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, pages 17–18, 2004.

[15] Dave Steinberg; Frank Budinsky; Marcelo Paternostro; Ed Merks. *EMF: Eclipse Modeling Framework, Second Edition.* Addison-Wesley Professional, 2008.

[16] Díaz García, Laura. Diseño de interfaces de usuario orientado a modelos: Extensión de ciat. Master's thesis, UNIVERSIDAD REY JUAN CARLOS. ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA. España., 2012.

[17] B. Dines., J. A. I. of Science, and Technology. *Domain Engineering: Technology Management, Research and Engineering.* COE research monography series. JAIST Press, 2009.

[18] Eloumri, Miloud Salem S. Graphical editors generation with the graphical modeling framework: A case study. Master's thesis, Queen's University. Canada, 2011.

[19] V. J. Faure D. User interface extensible markup language. *EICS '10 Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 361 – 362, 2010.

[20] Fowler Martin. *Domain Specific Languages.* Addison-Wesley Professional, 2010.

[21] Fuhrmann, Hauke and von Hanxleden, Reinhard. Taming graphical modeling. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] García, Pérez Guillermo. *CIAT-GUI: Generación Automática de Interfaces Gráficas de Usuario en el Contexto de CIAM*, 2010.

[23] Gasević, D. and Djurić, D. and Devedžić, V. *Model Driven Engineering and Ontology Development.* Springer London, Limited, 2009.

[24] Ghosh, Debasish. *Dsls in Action.* Manning Pubs Co Series. Manning, 2010.

[25] W. Giraldo, A. Molina, C. Collazos, M. Ortega, and M. Redondo. Ciat, a model-based tool for designing groupware user interfaces using ciam. In V. Lopez Jaquero, F. Montero Simarro, J. P. Molina Masso, and J. Vanderdonckt, editors, *Computer-Aided Design of User Interfaces VI*, pages 201–212. Springer London, 2009.

[26] Giraldo Orozco William Joseph. *Marco de Desarrollo de Sistemas Groupware Interactivos Basado en la Integración de Procesos y Notaciones.* PhD thesis, Ciudad Real: Escuela Superior de Informática de Ciudad Real, 2010.

[27] González Rubén Antolín. Aplicación de ingeniería dirigida por modelos (mde) en procesos de negocio (bpm). Master's thesis, Departamento de Ingeniería Informática. Escuela Politécnica Superior. Universidad Autónoma de Madrid, 2008.

[28] Gronback, Richard C. *Eclipse Modeling Project: A Domain–Specific Language (DSL) Toolkit.* The Eclipse Series. Pearson Education, 2009.

[29] Guerra, Esther and Lara, Juan and Wimmer, Manuel and Kappel, Gerti and Kusel, Angelika and Retschitzegger, Werner and Schönböck, Johannes and Schwinger, Wieland. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20:5–46, 2013.

[30] IEEE Software. Special issue on model–driven development. Technical Report 5, IEEE, 2003.

[31] John Krogstie. *Model-driven Development of Information Systems*, 2009. Model-driven Development of Information Systems.

[32] Jurack, Stefan and Taentzer, Gabriele. Towards composite model transformations using distributed graph transformation concepts. In A. Schrr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 226–240. Springer Berlin Heidelberg, 2009.

[33] Karlsch, Martin. A model-driven framework for domain specific languages demonstrated on a test automation language. Master's thesis, Hasso-Plattner-Institute of Software Systems Engineering Potsdam, Germany, 2007.

[34] Kelly, S. and Tolvanen, J.P. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[35] Kleppe Anneke. Warmer Jos. Bast Wim. *MDA Explained, The Model Driven Architecture: Practice and Promise.* The Addison-Wesley Object Technology Series. Addison-Wesley, 2003.

[36] Köllner, Christian and Dummer, Georg and Rentschler, Andreas and Müller-Glaser, K. D. Designing a graphical domain-specific modelling language targeting a filter-based data analysis framework. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ISORCW '10, pages 152–157, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Kolovos, Dimitrios S. and Rose, Louis M. and Abid, Saad Bin and Paige, Richard F. and Polack, Fiona A. C. and Botterweck, Goetz. Taming emf and gmf using model transformation. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] Kolovos, Dimitrios S. and Rose, Louis M. and Paige, Richard F. and Polack, Fiona A. C. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 13–19, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Lieberman, B.A. *The Art of Software Modeling.* Taylor & Francis, 2006.

[40] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and M. Florins. Usixml: A user interface description language supporting multiple levels of independence. In *ICWE Workshops*, pages 325–338, 2004.

[41] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lpez-Jaquero. Usixml: A language supporting multi-path development of user interfaces. In R. Bastide, P. Palanque, and J. Roth, editors, *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer Berlin Heidelberg, 2005.

[42] Limbourg Quentin. *Multi-Path Development of User Interfaces.* PhD thesis, Universitè Catholique de Louvain, 2004.

[43] V. J. Limbourg Quentin. Addressing the mapping problem in user interface design with usixml. *TAMODIA '04 Proceedings of the 3rd annual conference on Task models and diagrams*, pages 155 – 163, 2004.

[44] López L. Edna D; González G. Moisés; López S. Máximo; Iduñate R. Erick L. Proceso de desarrollo de software mediante herramientas mda. In *CISCI 6 Conferencia Iberoamericana en sistemas cibernética e informática.* Departamento de Ciencias Computacionales. Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET). México, 2007.

[45] Lucas Martínez Francisco Javier; Molina Molina Fernando; Toval Álvarez Ambrosio. Una propuesta de proceso explícito de v&v en el marco de mda. *Grupo de Ingeniería del Software. Departamento de Informática y Sistemas. Universidad de Murcia*, 157, 2003.

[46] Luyten Kris. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model–Based User Interface Development.* PhD thesis, Transnationale Universiteit Limburg. School voor Informatietechnologie, 2004.

[47] Marinescu, F. and Avram, A. *Domain-Driven Design: Quickly.* InfoQ : Enterprise software development series. Lulu.com, 2007.

[48] Mendel, Joanne. A taxonomy of models used in the design process. *interactions*, 19(1):81–85, Jan. 2012.

[49] A. Molina, M. Redondo, and M. Ortega. A conceptual and methodological framework for modeling interactive groupware applications. In Y. Dimitriadis, I. Zigurs, and E. Gmez-Snchez, editors, *Groupware: Design, Implementation, and Use*, volume 4154 of *Lecture Notes in Computer Science*, pages 413–420. Springer Berlin Heidelberg, 2006.

[50] A. I. Molina, W. J. Giraldo, J. Gallardo, M. A. Redondo, M. Ortega, and G. Garca. Ciat-gui: A mde-compliant environment for developing graphical user interfaces of information systems. *Advances in Engineering Software*, 52(0):10 – 29, 2012.

[51] A. I. Molina, M. A. Redondo, and M. Ortega. Una revisión de notaciones para el modelado conceptual de sistemas interactivos para el soporte del trabajo en grupo.

[52] A. I. Molina, M. A. Redondo, M. Ortega, and U. Hoppe. Ciam: A methodology for the development of groupware user interfaces. *J. UCS*, 14(9):1435–1446, 2008.

[53] M. F. Molina J, Vanderdonckt J and G. P. Towards virtualization of user interfaces based on usixml. *Web3D '05 Proceedings of the tenth international conference on 3D Web technology*, pages 169 – 178, 2005.

[54] Moore, B. *Eclipse Development Using the Graphical Editing Framework And the Eclipse Modeling Framework.* IBM Redbooks. Vervante, 2004.

[55] Mu, Liping and Gjosaeter, Terje and Prinz, Andreas and Tveit, Merete Skjelten. Specification of modelling languages in a flexible meta-model architecture. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 302–308, New York, NY, USA, 2010. ACM.

[56] Muller Pierre-Alain. Model transformations. an overview. http://www.irisa.fr/triskell/members/pierre-alain.muller/teaching/aboutmodeltransfo, 2005.

[57] Neil Carlos Gerardo. *Arquitectura de software dirigida por modelos - Diseño de un almacén de datos históricos en el marco del desarrollo de software dirigido por modelos.* PhD thesis, Universidad Nacional de la Plata. Argentina, 2010.

[58] OMG. *OMG: Unified Modeling Language (UML) Version 2.0.* Document number ormsc/2001-07-01. (2001) Disponible en: `http://www.uml.org/`.

[59] OMG. *OMG: Model Driven Architecture (MDA).*, 2001. Document number ormsc/2001-07-01. (2001).

[60] OMG. *OMG. Catalog of OMG Modeling and Metadata Specifications.*, 2003. OMG. Catalog of OMG Modeling and Metadata Specifications.

[61] OMG. *OMG: Meta Object Facility (MOF) Version 2.4.1*, 2011. Documents Associated With Meta Object Facility (MOF) Version 2.4.1 Release Date: August 2011.

[62] Pastor Oscar and Molina Juan Carlos. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling.* Springer, 2007.

[63] M. S. Paternó F., Mancini C. Concurtasktrees: A diagrammatic notation for specifying task models. *The Handbook Of Task Analysis For HCI*, pages 483 – 501, 2004.

[64] Pérez Jose Manuel; Ruiz Francisco; Piattini Mario. Model driven engineering aplicado a business process management. informe técnico uclm-tsiafit-002. Technical report, Universidad de Castilla-La Mancha. España, 2007.

[65] Quintero Juan Bernando. Marco de referencia para la evaluación de herramientas de transformación de modelos reporte técnico atl - atlas transformation language. Technical report, UNIVERSIDAD EAFIT. Medellín. Colombia, 2006.

[66] Quintero Juan Bernando & Anaya Raquel. Mda y el papel de los modelos en el proceso de desarrollo de software. *Revista EIA*, (8):131–146, 2007.

[67] Quintero Juan Bernardo; Duitama Muñoz Jhon Freddy. Reflexiones acerca de la adopción de enfoques centrados en modelos en el desarrollo de software. *Ingeniería y Universidad*, 15(1):219–243, 2011.

[68] J. Richley. *GMF: Beyond the Wizards*, 2007. GMF: Beyond the Wizards. Disponible en: `http://onjava.com/pub/a/onjava/2007/07/11/gmf-beyond-the-wizards.html?page=1`.

[69] Rose, LouisM. and Herrmannsdoerfer, Markus and Williams, JamesR. and Kolovos, DimitriosS. and Garcés, Kelly and Paige, RichardF. and Polack, FionaA.C. A comparison of model migration tools. In D. Petriu, N. Rouquette, and y. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin Heidelberg, 2010.

[70] Rubel, D. and Wren, J. and Clayberg, E. *The Eclipse Graphical Editing Framework (GEF)*. Eclipse Series. Pearson Education, 2011.

[71] Rumbaugh, J. and Booch, G. and Jacobson, I. and Rodríguez, H.C. and de la Fuente Alarcón, M. and Martínez, Ó.S. *El Lenguaje unificado de modelado: manual de referencia*. Fuera de colección Out of series. Pearson Educación, 2007.

[72] Ruscio, Davide and Lämmel, Ralf and Pierantonio, Alfonso. Automated co-evolution of gmf editor models. In B. Malloy, S. Staab, and M. Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin Heidelberg, 2011.

[73] Sánchez, Cuadrado Jesús. García, Molina Jesús. Building domain–specific languages for model–driven development. *IEEE Computer Society*, (0740-745907):2–9, 2007.

[74] Seehusen, Fredrik and Stolen, Ketil. An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2011.

[75] The Eclipse Foundation. *GEF/Developer FAQ*, 2012. GEF/Developer FAQ Disponible en: `http://wiki.eclipse.org/GEF_Developer_FAQ`.

[76] The Eclipse Foundation. *Graphical Editing Framework Programmer's Guide*, 2012. Eclipse documentation - Current Release Disponible en: `http://help.eclipse.org/juno/index.jsp`.

[77] The Eclipse Foundation. *Eclipse Modeling Framework Project (EMF)*, 2013. Eclipse Modeling Framework Project (EMF) Disponible en: `http://www.eclipse.org/modeling/emf/`.

[78] The Eclipse Foundation. *Eclipse Modeling Project (EMP)*, 2013. Eclipse Modeling Project (EMP) Disponible en: `http://www.eclipse.org/modeling/`.

[79] The Eclipse Foundation. *EMF Core*, 2013. Disponible en: `http://www.eclipse.org/modeling/emf/?project=emf#emf`.

[80] The Eclipse Foundation. *GEF (Graphical Editing Framework)*, 2013. GEF (Graphical Editing Framework) Disponible en: `http://www.eclipse.org/gef/`.

[81] The Eclipse Foundation. *Graphical Modeling Framework*, 2013. Graphical Modeling Framework/Models/GMFMap.

[82] The Eclipse Foundation. *Graphical Modeling Project (GMP)*, 2013. Graphical Modeling Project (GMP) Disponible en: `http://www.eclipse.org/modeling/gmp/?project=gmf-tooling#gmf-tooling`.

[83] UsiXML Consortium. *UsiXML, USer Interface eXtensible Markup Language. Reference Manual*, 2007. versión 1.8 Disponible en: `http://www.isys.ucl.ac.be/bchi`.

[84] J. Vanderdonckt. Model-driven engineering of user interfaces: Promises, successes, failures, and challenge. *5th Annual Romanian Conf. on Human–Computer Interaction ROCHI'2008*, pages 1 – 10, 2008.

[85] Villegas, Maria Lili. Metodología para el desarrollo de sistemas interactivos usables ciaf+hci a partir de la integración de ciaf y mpiu+a. Master's thesis, Universidad EAFIT. Colombia, 2012.

[86] Völter, M. and Stahl, T. and Bettin, J. and Haase, A. and Helsen, S. and Czarnecki, K. and von Stockfleth, B. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. John Wiley & Sons, 2006.

[87] Wienands, Christoph and Golm, Michael. Anatomy of a visual domain-specific language project in an industrial context. In A. Schrr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 453–467. Springer Berlin Heidelberg, 2009.