Capítulo 1

Introducción

Este capítulo sirve de introducción a la presente Memoria de Proyecto Fin de Carrera. En él se describen los objetivos generales del proyecto, así como el contexto donde se enmarca. Por último, se describe como se estructura el presente documento.

1.1. Introducción

El principal objetivo de este Proyecto de Fin de Carrera es implementar un conjunto de generadores de código que permitan transformar modelos orientados a características para una línea de productos software en una implementación de dichos diseños para la plataforma .NET utilizando para ello las facilidades que ofrecen las clases parciales del lenguaje C# basadas en el Patrón Slicer. A continuación intentaremos introducir de forma breve al lector en estos conceptos.

El objetivo de la *Línea de Producto Software* [**?**], de un segmento particular de mercado, es reutilizar una base común de tecnología y proporcionar productos personalizados, acordes a las necesidades del cliente de manera eficiente, eficaz y a un coste razonable.

El uso de las líneas de producto software permite la reducción de costes de desarrollo por la reutilización de la tecnología en los distintos sistemas, a mayor cantidad de productos a desarrollar mayor rentabilidad respecto a los sistemas creados individualmente. Ofrece alta calidad en el producto resultante porque se realizan pruebas de los componentes de la plataforma en diferentes tipos de producto para ayudar a detectar y corregir errores. Reduce el tiempo de creación debido a la reutilización de los componentes ya existentes para cada nuevo producto y reduce también el esfuerzo requerido por el mantenimiento ya que cuando se cambia algo de un componente de la plataforma, ese cambio se propaga a todos los componentes que lo empleen, y de esta forma se reduce el esfuerzo de aprender cómo funciona cada elemento individualmente.

En contraposición a la flexibilidad que ofrece el desarrollo de software individual, específico para cliente pero que supone grandes costes, las líneas de producto software delimitan las variaciones de sus productos a un conjunto prefijado y optimizan, por tanto, los procesos para dichas variaciones.

La línea de productos software se puede extrapolar a otros ámbitos de producción. Un ejemplo clásico de línea de productos es la fabricación de automóviles, donde se ofrece al cliente un modelo base al cual puede añadir aquellos extras que así desee, personalizando el vehículo y adaptándolo a sus necesidades. De esta forma partiendo del mismo modelo y de unas variaciones adicionales preestablecidas, y diseñadas de tal forma que se adaptan perfectamente al modelo seleccionado, se puede obtener gran cantidad de variaciones en el modelo final de manera automática.

En el ámbito del desarrollo software, las empresas ya no se centran en la creación de un producto específico para un cliente (por ejemplo, diseñar y construir un portal para la Universidad de Cantabria), sino en un domino (por ejemplo, diseñar y construir un portal para universidades). Los principales desafíos a los que se enfrentan las empresas son: delimitar dicho dominio, identificar las distintas variaciones que se van a permitir y desarrollar la infraestructura que permita realizar los productos a bajo coste sin reducir la calidad.

El proceso de desarrollo de la línea de productos software se divide en dos procesos [**?**]: Ingeniería de Dominio e Ingeniería de la Aplicación. Por un lado la *Ingeniería del Dominio* se encarga de la construcción de la plataforma mediante la delimitación del conjunto de aplicaciones para las que está creada, además de definir y construir qué características serán reusables y cuales específicas para cada uno de los productos que se desean fabricar.

Por otra parte, la *Ingeniería de la Aplicación* se encarga de la creación de los productos para clientes concretos. Partiendo de la plataforma creada en la fase de Ingeniería de Dominio, y reutilizando tantos componentes como fuera necesario, se crea una especialización del producto base acorde a los requisitos del cliente.

Una de las técnicas más utilizadas para realizar dicho análisis es el *diseño orientado a características*[**?**], que intenta encapsular porciones coherentes de funcionalidad proporcionadas por una aplicación en módulos independientes llamados características. Lo que convirte la obtención de diferentes versiones de una misma aplicación combinando diferentes conjuntos de características en una tarea sencilla. Los *diseños orientados a características* deben asegurar que el resultado de la composición de un conjunto de características produce como resultado una aplicación correcta y segura.

Tal como se ha descrito al inicio de este apartado, el objetivo del presente Proyecto Fin de Carrera consiste en el desarrollo e implementación de unos generadores de código que permitan la tranformación del diseño de los modelos en una implementación en código C# de dichos diseños, para ello se usarán las prestaciones que ofrecen el uso de las clases parciales del

lenguaje C# basadas en el patrón Slicer.

Las *clases parciales* permiten a los desarrolladores fragmentar la implementación de una clase en un conjunto de ficheros, cada uno de los cuales contiene una porción, o incremento, de una funcionalidad de la clase. Sin embargo, no ofrecen ningún mecanismo para agrupar o encapsular características, por lo que no es posible ocultar clases y métodos que pertenecen a una característica específica de aquellas clases y métodos que pertenecen a características independientes. Además, permiten añadir nuevos atributos y métodos a existentes clases parciales pero no permite sobreescribir o extender métodos ya existentes.

Para solventar dichos problemas, el profesor Pablo Sánchez, dentro del Departamento de Matemáticas, Estadística y Computación, ha desarrollado un patrón de diseño llamado *Patrón Slicer* [**?**] que parte de la siguiente idea: todos los problemas que se pretenden solucionar tienen origen en el hecho de no poner tener métodos con el mismo nombre en distintas clases parciales, hay que evitar dicha situación. Estos fragmentos de clases parciales, son combinados en tiempo de compilación para crear una única clase que auna todas las características seleccionadas inicialmente por el cliente.

Por ejemplo, supongamos que un cliente quiere un vehículo con varias características adicionales entre las que se encuentran: aire acondicionado, sensor de lluvia, medidor de temperatura en grados Celsius y GPS integrado en idioma español e inglés. La base de nuestro producto final será el vehículo, al cual iremos añadiendo las distintas características requeridas por el cliente. Hay algunas peculiaridades, la clase del medidor de temperatura puede estar a su vez fragmentada en varios componentes (temperatura en Celsius, temperatura en Farentheit) y de los cuales en el modelo final solo usaremos uno de ellos, el de temperatura en Celsius. Lo mismo ocurre con el selector de idiomas para el GPS, solo se elegirá el idioma español e inglés. De esta forma, el producto final juntará todas estas características dentro de un mismo elemento que será el vehículo entregado al usuario final atendiendo a sus requisitos.

El objetivo de este Proyecto Fin de Carrera es implementar generadores de código que abordarán tanto la implementación de la familia de productos software cubierta por la línea de productos, como la configuración de productos concretos pertenecientes a dicha familia utilizando las prestaciones de las clases parciales en C# y el Patrón Slicer. Con esto esperamos haber aclarado el primer párrafo de esta sección al lector no familiarizado con las líneas de productos software, clases parciales en lenguaje C# y/o el Patrón Slicer.

Tras esta introducción, el resto del presente capítulo se estructura como sigue: La Sección [] proporciona []. Por último, la Sección 1.3 describe la estructura general del presente documento.

1.2. Planificación del proyecto

1.3. Estructura del Documento

Capítulo 2

Antecedentes

Este capítulo trata de describir a grandes rasgos las técnicas, tecnologías y herramientas utilizadas para el desarrollo del presente Proyecto Fin de Carrera. En primer lugar, se introducirá el caso de estudio que se utilizará de forma recurrente a lo largo del proyecto, que consiste en una línea de productos software para software de control de hogares inteligentes. Para ello se describen en primer lugar diversos conceptos relacionados el dominio del proyecto, como son las líneas de productos software, la tecnología TENTE, el diseño orientado a características, el patrón slicer y la generación de código.

2.1. Caso de Estudio: Software para Hogares Inteligentes

El objetivo último del presente proyecto es la construcción de una línea de productos software sobre la plataforma .NET para hogares automatizados y/o inteligentes.

El objetivo de estos hogares es aumentar la comodidad y seguridad de sus habitantes, así como hacer un uso más eficiente de la energía consumida. Los ejemplos más comunes de tareas automatizadas dentro de un hogar inteligente son el control de las luces, ventanas, puertas, persianas, aparatos de frío/calor, así como otros dispositivos, que forman parte de un hogar. Un hogar inteligente también busca incrementar la seguridad de sus habitantes mediante sistemas automatizados de vigilancia y alerta de potenciales situaciones de riesgo. Por ejemplo, el sistema debería encargarse de detección de humos o de la existencia de ventanas abiertas cuando se abandona el hogar.

El funcionamiento de un hogar inteligente se basa en el siguiente esquema: (1) el sistema lee datos o recibe datos de una serie de sensores; (2) se procesan dichos datos; y (3) se activan los actuadores para realizar las acciones que correspondan en función de los datos recibidos de los sensores.

Todos los sensores y actuadores se comunican a través de un dispositivo especial denominado puerta de enlace (*Gateway*, en inglés). Dicho disposi-

tivo se encarga de coordinar de forma adecuada los diferentes dispositivos existentes en el hogar, de acuerdo a los parámetros y preferencias especificados por los habitantes del mismo. Los habitantes del hogar se comunicarán con la puerta de enlace a través de una interfaz gráfica. Este proyecto tiene como objetivo el desarrollo de un hogar inteligente como una línea de productos software, con un número variable de plantas y habitaciones. El número de habitaciones por planta es también variable. La línea de productos deberá ofrecer varios servicios, que podrán ser opcionalmente incluidos en la instalación del software para un un hogar determinado. Dichos servicios se clasifican en funciones básicas y complejas, las cuales describimos a continuación.

Funciones básicas

1. *Control automático de luces:* Los habitantes del hogar deben ser capaces de encender, apagar y ajustar la intensidad de las diferentes luces de la casa. El número de luces por habitación es variable. El ajuste debe realizarse especificando un valor de intensidad.

2. *Control automático de ventanas:* Los residentes tienen que ser capaces de controlar las ventanas automáticamente. De tal modo que puedan indicar la apertura de una ventana desde las interfaces de usuario disponibles.

3. *Control automático de persianas:* Los habitantes podrán subir y bajar las persianas de las ventanas de manera automática.

4. *Control automático de temperatura:* El usuario será capaz de ajustar la temperatura de la casa. La temperatura se medirá siempre en grados celsius.

Funciones complejas

1. *Control inteligente de energía:* Esta funcionalidad trata de coordinar el uso de ventanas y aparatos de frío/calor para regular la temperatura interna de la casa de manera que se haga un uso más eficiente de la energía. Por ejemplo, si se recibe la orden de calentar la casa, a la vez que se activan los radiadores se cerrarán las ventanas para evitar las pérdidas de calor.

2. *Presencia simulada:* Para evitar posibles robos, cuando los habitantes abandonen la casa por un periodo largo de tiempo, se deberá poder simular la presencia de personas en las casas. Hay dos opciones de simulación (no exclusivas):

a) *Simulación de las luces:* Las luces se deberán apagar y encender para simular la presencia de habitantes en la casa.

b) *Simulación de persianas:* Las persianas se deberán subir y bajar automática para simular la presencia de individuos dentro de la casa.

Todas estas funciones son opcionales. Las personas interesadas en adquirir el sistema podrán incluir en una instalación concreta de este software el número de funciones que ellos deseen. La siguiente sección profundiza en el concepto *línea de productos software*.

2.2. Líneas de Producto Software

El objetivo de una *línea de producto software* [**?**, **?**] es crear una infraestructura adecuada a partir de la cual se puedan derivar, de forma tan automática como sea posible, producto concretos pertenecientes a una familia de producto software. Una familia de producto software es un conjunto de aplicaciones software similares, lo que implica que comparten una serie de características comunes, pero que también presentan variaciones entre ellos.

Un ejemplo clásico de familia de producto software es el software que se encuentra instalado por defecto en un teléfono móvil. Dicho software contiene una serie de características comunes, tales como agenda, recepción de llamadas, envío de mensajes de texto, etc. No obstante, dependiendo de las capacidades y la gama del producto, éste puede presentar diversas funcionalidades opcionales, tales como envío de correos electrónicos, posibilidad de conectarse a Internet mediante red inalámbrica, radio, etc.

La idea de una línea de producto software es proporcionar una forma automática y sistemática de construir producto concretos dentro de una familia de producto software mediante la simple especificación de qué características deseamos incluir dentro de dicho producto. Esto representa una alternativa al enfoque tradicional de desarrollo software, el cual se basaba simplemente en seleccionar el producto más parecido dentro de la familia al que queremos construir y adaptarlo manualmente.

El proceso de creación de líneas de producto software conlleva dos fases: *Ingeniería del Dominio* (en inglés, *Domain Engineering*) e *Ingeniería de Aplicación* (en inglés, *Application Engineering*) (la figura 2.1 ilustra el proceso para ambas fases). La *Ingeniería del Dominio* tiene como objetivo la creación de la infraestructura o arquitectura de la línea de producto, la cual permitirá la rápida, o incluso automática, construcción de sistemas software específicos pertenecientes a la familia de producto. La *Ingeniería de Aplicación* utiliza la infraestructura creada anteriormente para crear aplicaciones específicas adaptadas a las necesidades de cada usuario en concreto.

En la fase de Ingeniería del Dominio, el primer paso a realizar es un análisis de qué características de la familia de producto son variables y por

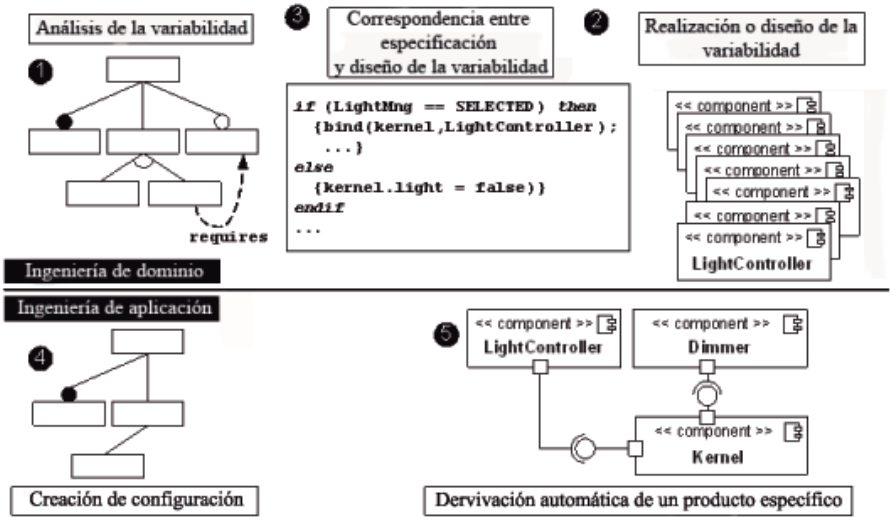

Figura 2.1: Proceso de Desarrollo de una línea de producto software

qué, y cómo son variables. Esta parte es la que se conoce como *Análisis o Especificación de la Variabilidad* (figura 2.1, punto 1). A continuación, se ha de diseñar una arquitectura o marco de trabajo para la familia de producto software que permita soportar dichas variaciones. Esta actividad se conoce como *Realización o Diseño de la Variabilidad* (figura 2.1, punto 2). El siguiente paso es establecer una serie de reglas que especifiquen cómo hay que instanciar la arquitectura previamente creada de acuerdo con las características seleccionadas por cada cliente. Esta fase es la que se conoce como *Correspondencia entre Especificación y Diseño de la Variabilidad* (figura 2.1, punto 3).

En la fase de Ingeniería de Aplicación, se crea una *Configuración*, se trata de una selección de características que un usuario desea incluir en su producto (figura 2.1, punto 4). En el caso ideal, usando esta configuración, se debe poder ejecutar las reglas de correspondencia entre especificación y diseño de la variabilidad para que la arquitectura creada en la fase de Ingeniería del Dominio se adaptase automáticamente generando un producto concreto específico acorde a las necesidades concretas del usuario (figura 2.1, punto 5). En el caso no ideal, dichas reglas de correspondencia deberán ejecutarse a mano, lo cual suele ser un proceso tedioso, largo, repetitivo y propenso a errores.

La siguiente sección proporciona una breve pero completa descripción sobre la tecnología TENTE, encargada de definir una serie de modelos y transformaciones de modelo a código para el desarrollo y configuración de líneas de producto software.

2.3. TENTE

TENTE [?, ?] es una moderna metodología para el desarrollo de líneas de productos software desarrollada en el contexto del proyecto AMPLE[1]. TENTE integra diversos avances para el desarrollo de líneas de productos software, tales como avanzadas técnicas de modularización y desarrollo software dirigido por modelos.

Las técnicas avanzadas de modularización permiten el encapsulamiento en módulos bien definidos y fácilmente componibles de las diferentes características de una familia de productos software, lo cual simplifica el proceso de construcción de productos específicos. Dicha modularización de características se realiza desde la fase arquitectónica, usando mecanismos específicos del lenguaje de modelado UML [?]. Después, mediante el uso de generadores de código, a partir del diseño arquitectónico de una familia de productos software se genera el esqueleto de su implementación. Dicha implementación se realiza en el lenguaje CaesarJ [?], una extensión de Java que incluye potentes mecanismos para soportar la separación y composición de características. Dichos esqueletos se completan manualmente, obteniéndose al final un conjunto de módulos software, o piezas, cuya composición da lugar a productos software concretos[2]. Esta fase constituye la definición de la infraestructura desde la cual se derivarán los productos concretos.

Para la derivación de productos concretos desde la infraestructura descrita en el párrafo anterior, TENTE usa un innovador lenguaje, denominado VML [?, ?], basado en transformaciones de modelo a modelo de orden superior. Dicho lenguaje sirve para especificar qué acciones hay que realizar sobre un modelo que describe la familia completa de productos software para adaptarlo a las necesidades del cliente. Posteriormente, dada una lista con las características que el cliente desea incluir o excluir de su producto concreto, VML es capaz de transformar automáticamente el modelo de la familia de productos software para adaptarlo a las necesidades de dicho cliente.

Acto seguido, se utiliza dicho modelo de un producto concreto como entrada para un generador automático de código, que creará el código necesario para componer los módulos o piezas software que se constituyeron durante la creación de la infraestructura de la línea de productos software.

Esta metodología posee diversas ventajas:

1. Gracias al uso de técnicas orientadas a características, como el operador *merge* de UML y el lenguaje CaesarJ, se facilita la modularización y composición de características, lo que facilita no sólo el proceso de

[1] www.ample-project.net

[2] El nombre de la metodología proviene del célebre juego de construcción TENTE, versión española del popular Lego.

obtención de productos concretos, sino también la reutilización y evolución de dichas características [**?**].

2. Gracias al uso de técnicas dirigidas por modelos, se automatiza gran parte del proceso, evitando tareas repetitivas, largas, tediosas y monótonas, usualmente propensas a errores.

3. Gracias al uso de lenguajes avanzados (como VML) y tecnologías estándares de modelado (como UML) se evita que los desarrolladores, tanto de la infraestructura como de los productos concretos, tengan que poseer cierta experiencia en el uso de técnicas de desarrollo software dirigido por modelos tales como creación de transformaciones de modelo a modelo en lenguajes como ATL [**?**] o Epsilon [**?**].

No obstante, a pesar de sus bondades, se han encontrado diversas dificultades a la hora de transferir esta metodología a las empresas de desarrollo software situadas en el entorno cántabro. Las principales dificultades provienen de dos fuentes distintas pero relacionadas, descritas a continuación.

Problema 1: Resistencia a cambiar el lenguaje de implementación habitual

En primer lugar, y éste no es un solo un problema encontrado en el entorno de Cantabria, TENTE está basado en el uso del lenguaje CaesarJ como lenguaje de implementación. Podría usarse, con el coste asociado de tener que escribir nuevos generadores de código, lenguajes similares a CaesarJ tales como ObjectTeams [**?**]. En cualquier caso, hace falta un lenguaje con fuerte soporte para la modularización y composición de características, y en especial, con soporte para *clases virtuales* [**?**]. La mayoría de las empresas son bastante reticentes a cambiar su lenguaje habitual de programación, debido fundamentalmente a dos motivos:

1. El coste de aprendizaje que ello supone, ya que los programadores han de familiarizarse con nuevos conceptos y técnicas de codificación software.

2. El uso de un nuevo lenguaje de programación podría dejar obsoletas muchas herramientas, como suites para la ejecución de casos de prueba, que la empresa podría tener asociadas al anterior lenguaje de programación.

Además, en el caso de lenguajes de reciente creación como CaesarJ u ObjectTeams, aunque los compiladores están completamente desarrollados y son bastante estables, las facilidades auxiliares asociadas a un lenguaje de programación maduro tipo Java o C# no están a menudo disponibles. Por

ejemplo, CaesarJ no soporta compilación incremental por el momento, por lo que un ligero cambio en el nombre de un atributo obliga a recompilar el proyecto completo.

Problema 2: Obligatoriedad de usar la plataforma .NET

Debido a diferentes razones estratégicas de negocio, la mayoría de las empresas de desarrollo software en Cantabria trabajan casi en exclusiva sobre la plataforma .NET [**?**], siendo además bastante reticentes a considerar el cambio a otras plataformas. La mayoría de los lenguajes con fuerte soporte para orientación a características, como suele ser el caso habitual en los lenguajes académicos, están basados en Java.

Por tanto, TENTE es una tecnología llena de ventajas, pero quizás excesivamente innovadora para el momento actual. Por dicho motivo, se decidió crear una nueva metodología, basada en TENTE, pero que resultase más fácilmente transferible a la industria. Para favorecer dicha transferencia, se tomaron dos decisiones estratégicas:

1. El lenguaje de programación usado en el nivel de implementación tenía que ser un lenguaje comercial estándar, tipo Java, C# o C++. Esto evitaría los problemas derivados de obligar a las empresas a aprender un nuevo lenguaje y estilo de programación, y sobre todo, a cambiar sus entornos de desarrollo.

2. El nuevo lenguaje de programación de la metodología debía ser C# [**?**], al ser éste el lenguaje bandera de la plataforma .NET, y al desarrollar las empresas del entorno de nuestra universidad casi en exclusiva software para dicha plataforma.

Tal nueva metodología recibiría el nombre de TENTE.NET (versión para la plataforma .NET de la metodología TENTE), y está actualmente en fase de desarrollo. El primer paso para la adaptación de la metodología TENTE a la plataforma .NET era encontrar un mecanismo para simular las facilidades ofrecidas por diseños orientados a características dentro del lenguaje C#. Tomando las ideas de Laguna et al [**?**] como base, se pensó inicialmente que las *clases parciales* proporcionadas por C# podían ser un mecanismo adecuado para alcanzar cierto grado de desarrollo orientado a características. Sin embargo, un posterior estudio realizado por López et al [**?**] demostraba que las clases parciales de C# poseían más limitaciones de las inicialmente previstas por Laguna et al para la implementación de diseños software orientados a características.

Por tanto, el siguiente paso hacia la construcción de la variante para .NET de la metodología TENTE era la solución de tales limitaciones. La siguiente sección profundiza en el concepto expuesto brevemente en la presente sección, el diseño orientado a características.

2.4. Diseño Orientado a Características con UML

El *diseño orientado a características* [**?**] es un paradigma para la construcción, adaptación y síntesis de sistemas software a gran escala. Una *característica* es una unidad de funcionalidad de un sistema software que satisface un requisito, representa una decisión de diseño y proporciona una opción de configuración potencial. La idea básica del diseño orientado a características es descomponer un sistema software en módulos agrupando las características que ofrece. El objetivo de la descomposición es la construcción de software bien estructurado que puede ser adaptado a las necesidades del usuario y al entorno de la aplicación.

Típicamente, a partir de un conjunto de características, se pueden generar multitud de sistemas software compartiendo características comunes y diferenciándose en otras. El conjunto de los sistemas software generados a partir de un conjunto de características, comentado en la sección 2.2, corresponde al concepto de *línea de productos software*.

Para estudiar las ventajas del diseño orientado a características nos basaremos en las facilidades proporcionadas por CaesarJ [**?**], e ilustraremos los ejemplos con diagramas UML. CaesarJ es un lenguaje de programación orientado a características basado en Java, para ello trabaja con el concepto de familia de clases. Una *familia de clases* es una unidad de encapsulamiento que sirve para agrupar clases relacionadas. Las familias de clases reciben un tratamiento similar al de las clases, soportando relaciones de herencia entre ellas.

Asimismo, CaesarJ también introduce el concepto de clases virtuales. Una *clase virtual* es una clase perteneciente a una familia de clases y que es susceptible de ser heredada y sobreescrita por familias de clases que hereden de la familia de clases que contiene dicha clase virtual. La figura 2.2 ilustra esta situación. Las familias de clases se representan mediante paquetes UML, y herencia entre clases, mediante relaciones *merge*. Cuando una familia de clases hereda de otra, hereda implícitamente todas sus clases virtuales. Si la primera contiene clases virtuales con el mismo nombre que la familia de clases que hereda, entonces la clase virtual de la familia de clases hija hereda de la clase virtual con el mismo nombre de la familia de clases padre. Por ejemplo, en la figura 2.2, la clase Mult de la familia de clases Eval heredaría implícitamente de la clase Mult de la familia de clases Basic. En cada familia de clases hija, se pueden añadir por tanto nuevos atributos y métodos a las clases virtuales de las familias de clases padre. Además, gracias al avanzado sistema de tipos de CaesarJ, se pueden añadir nuevas relaciones de herencia.

Además, las referencias entre clases se actualizan automáticamente. Por ejemplo, en el caso de la figura 2.2, aunque no se haga explícitamente, cualquier referencia a una clase del tipo Expression dentro de la familia de clases Eval se referirá a la clase virtual Expression de la familia de clases Eval y no a la clase virtual de mismo nombre de la familia de clases Expressions. De esta

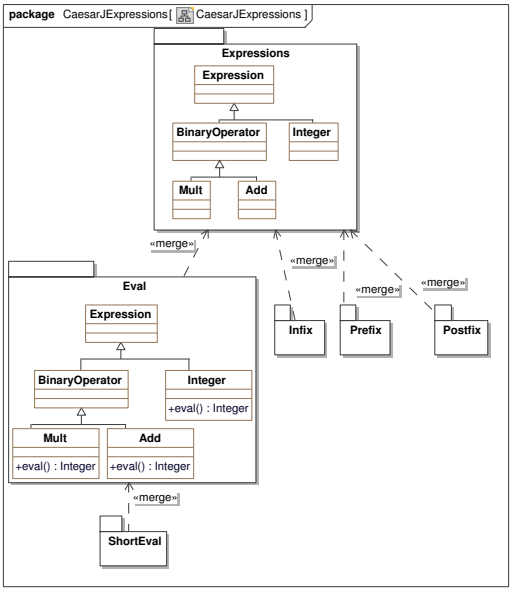

Figura 2.2: Diseño para resolver el problema de las expresiones con CaesarJ

forma, las referencias están siempre actualizadas a su versión más extendida.

Para implementar una línea de productos software, cada característica se considera como una familia de clases. Dentro de cada familia de clases, cada característica se diseña usando las técnicas tradicionales de la orientación a objetos, tal como se muestra en la figura 2.2.

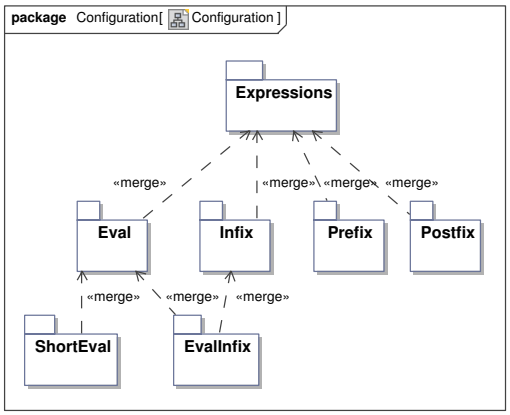

Figura 2.3: Composición de las características Eval e Infix en nuevo producto con CaesarJ

Para realizar una configuración, es decir, para crear un producto concreto por composición de características, simplemente hay que crear una nueva

familia de clases que herede de las familias de clases que correspondan a las características seleccionadas. La figura 2.3 muestra como se crearía un producto nuevo mediante la composición de las características Eval e Infix.

Con el fin de solventar las limitaciones expuestas en la sección 2.3 y apoyándose en el diseño orientado a características (sección 2.4), en la siguiente sección se describirá el patrón para resolver dichas limitaciones, denominado patrón slicer.

2.5. Slicer Pattern

Antes de proceder a la explicación del Patrón Slicer, es conveniente presentar el concepto de *clase parcial* de manera breve al lector. Las clases parciales [**?**] permiten dividir la implementación de una clase en varios archivos de código fuente. Cada fragmento representa una parte de la funcionalidad global de la clase. Todos estos fragmentos se combinan en tiempo de compilación para crear una única clase, la cual contiene toda la funcionalidad especificada en las clases parciales. Por lo tanto, las clases parciales C# parecen un mecanismo adecuado para implementar características, tal como ha sido identificado por diversos autores [**?**, **?**], dado que cada incremento en funcionalidad perteneciente a una característica se podría encapsular en una clase parcial separada. Sin embargo, las clases parciales tiene un serio inconveniente para ser utilizadas como un mecanismo de apoyo a la orientación de características: no son compatibles con la extensión de métodos ni la reescritura. El Patrón Slicer surge como patrón para solución de esta limitación.

El problema a solucionar por el patrón slicer tiene su origen en el hecho de que no se puede disponer de métodos con el mismo nombre en diferentes clases parciales.

La instanciación del patrón slicer [**?**] en los métodos regulares, consiste en añadir un prefijo a cada método que se corresponda con el nombre de la característica a la que cada método pertenece. La figura 2.4 ilustra cómo el caso de estudio de Hogar Inteligente ha sido refactorizado siguiendo dicha idea. En esta figura, las características han sido representadas como rectángulos conteniendo clases. Cada rectángulo ha sido etiquetado con el nombre de la característica que representa.

Usando esta estrategia se puede comprobar cómo, por ejemplo, las versiones del método thermometerChanged correspondientes a las características HeaterMng y SmartEnergyMng han sido transformadas en heaterMng_thermometerChanged y smartEnergyMng_thermometerChanged respectivamente y por tanto pueden co-existir sin que sus nombres colisionen. Es más, el método smartEnergyMng_thermometerChanged puede extender del método heaterMng_thermometerChanged. Se dispone por tanto de varias versiones parciales de un mismo método.

Para generar un producto específico es necesario que, a nivel de Ingeniería

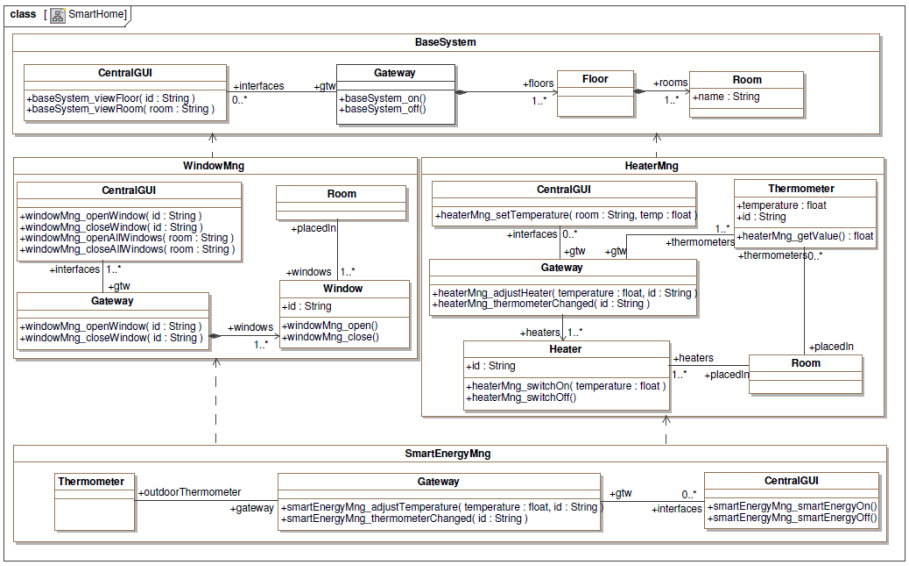

Figura 2.4: Proceso de Desarrollo de una Línea de Productos Software

de Aplicación, se cree la denominada "versión limpia"del método thermometerChanged, es decir, sin el prefijo. Mientras que las versiones de dicho método creadas en el nivel de Ingeniería del Dominio que han sido prefijadas con el nombre de la característica a la que cada método pertenece pasarán a ser denominadas "versiones sucias"de dicho método. Además, para asegurar que se invoca la versión correcta del método, no se deberían poder invocar los métodos heaterMng_thermometerChanged y smartEnergyMng_thermometerChanged directamente y por dicha razón todas las versiones sucias de los métodos son privadas. De dicha forma los objetos de las demás clases sólo podrán invocar a la versión limpia del método y dicha versión, redireccionará la llamada a las versiones sucias de los métodos correspondientes de acuerdo a las características seleccionadas.

Sin embargo este patrón no puede ser utilizado para los constructores de la clase ya que los constructores deben tener un nombre específico y por tanto, no se pueden renombrar. Por tanto, la instanciación del patrón slicer en los constructores debe realizarse de manera diferente a la instanciación del resto de métodos. De esta forma, cada clase parcial X correspondiente a una caracterísica F tendrá un método privado llamado <F>_init_<X> que contendrá el fragmento de la lógica del constructor correspondiente a la característica F.

El listing 2.1 muestra cómo se aplica dicha técnica. Se puede apreciar cómo la lógica del constructor para *Gateway* ha sido encapsulada en un método llamado baseSystem_initGateway (listing 2.1 líneas 03-06). La misma técnica ha sido usada en el listing 2.1 líneas 10-12 con la característica WindowMng. El siguiente paso es encontrar un mecanismo que permita componer

dichos fragmentos de acuerdo a una selección de características dada, de esta forma, como se puede apreciar en el listing 2.1 líneas 16-20, el constructor para la clase Gateway es creado con las características seleccionadas por el usuario final, en el caso analizado, por ejemplo, solo se ha seleccionado la característica WindowMng.

```
File BaseSystem/Gateway.cs
-----------------------------------------------------
01 public partial class Gateway {
02      ...
03      private void BaseSystem_initGateway() {
04          this.floors = new List<Floors>();
05          this.interfaces = new List<CentralGUI>();
06      }
07 }

File WindowMng/Gateway.cs
-----------------------------------------------------
08 public partial class Gateway {
09      ...
10      private windowMng_initGateway() {
11          this.windows = new List<Window>();
12      }
13 }

File MyHouse/Gateway.cs
-----------------------------------------------------
14 public partial class Gateway {
15      ...
16      public Gateway() {
17          // WindowMng has been selected
18          baseSystem_initGateway();
19          windowMng_initGateway();
20      }
21 }
```

Listing 2.1: Código para el constructor de la clase Gateway usando clases parciales y patrón slicer

Concluyendo con el patrón slicer, tanto los métodos regulares como con los constructores pueden ser extendidos y reescritos usando dicho patrón y solucionando así las limitaciones ofrecidas por el uso de clases parciales [**?**] como mecanismo de soporte orientado a características. Utilizando todo lo expuesto en las secciones anteriores, en la siguiente sección se expone a grandes rasgos el funcionamiento de los generadores de código que serán empleados durante la fase de Ingeniería del Dominio del presente proyecto.

2.6. Generación de Código con Epsilon

Epsilon [**?**] es una plataforma para la construcción de lenguajes consistentes e interoperables para las tareas de gestión de modelado tales como transformación de modelos, generación de código, comparación de modelos, técnicas merge, refactorización y validación. El presente proyecto se ha centrado en la generación de código y por tanto en los lenguajes *Epsilon Ge-*

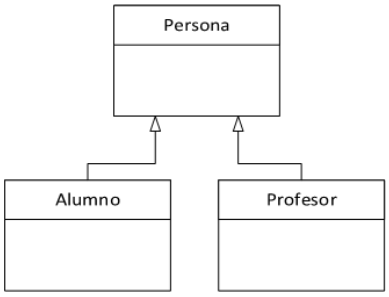

Figura 2.5: Ejemplo de modelo de entrada para generación de código con EGL

neration Language (EGL) y *Epsilon Object Language* (EOL) proporcionados por Epsilon, que se analizan con más detalle a continuación.

Epsilon Generation Language(EGL)

EGL es un *plug-in* para Eclipse que proporciona un lenguaje apropiado para las transformaciones modelo a texto (*model-to-text-transformation*, abreviado M2T). EGL puede ser usado para transformar modelos en varios tipos de artefactos de carácter textual, incluyendo códigos ejecutables (por ejemplo Java), informes (por ejemplo en HTML), imágenes (por ejemplo usando DOT), especificaciones formales (por ejemplo el lenguaje Z), o incluso aplicaciones completas generadas con múltiples lenguajes (por ejemplo HTML, Javascript y CSS).

EGL es un generador de código basado en plantillas; es decir, que la parte programada se asemeja al contenido que se va a generar, proporcionan además diversas características que simplifican y dan soporte a la generación de texto desde modelos. La figura 2.5 muestra un ejemplo sencillo del modelo de entrada para un programa EGL, donde se pueden observar tres clases: Persona, Alumno y Profesor. Supongamos que se quiere obtener el nombre de las clases, para ello se debería generar un código similar al del listing 2.2 donde aparece un bucle (líneas 1-3) que recorre todas las clases contenidas en el modelo de entrada y en cada una de ellas (línea 2) se genera el texto *El modelo contiene la clase: <nombre de la clase>*. De esta forma el resultado para el ejemplo de la figura 2.5, después de haber sido tratado mediante EGL tal como muestra el listing 2.2, queda expuesto en el listing 2.3.

```
1 [% for (c in Class.all) { %]
2     El modelo contiene la clase: [%=c.name%]
3 [% } %]
```

Listing 2.2: Generación del nombre de cada Class contenida en un modelo de entrada

```
1 El modelo contiene la clase: Persona
2 El modelo contiene la clase: Alumno
3 El modelo contiene la clase: Profesor
```

Listing 2.3: Resultado de la generación del nombre de cada Class contenida en un modelo de entrada

EGL no solo nos permite generar estos sencillos ejemplos, es una herramienta mucho más potente que permite la generación de funciones creadas específicamente por el usuario para facilitar la obtención de datos de los modelos de entrada. Tal como se aprecia en el listing 2.4 se puede programar una operación que implemente la cabecera de una clase Java y de esta forma invocar dicha función tantas veces como sea necesario en aquellos programas EGL que lo necesiten. En el listing 2.4 (línea 4) la operación *visibility* indica la visibilidad de *self* siendo en este caso la visibilidad de la clase tal como muestra la línea 3. Mientras que la operación *name*, al igual que en el listing 2.2, se refiere al nombre del elemento actual, en este caso el nombre de la clase.

```
1 [%=c.declaration()%]
2 [% @template
3 operation Class declaration() { %]
4     [%=self.visibility] class [%=self.name%] {}
5 [% } %]
```

Listing 2.4: Uso de una operación que especifica el texto generado para una declaración de una clase Java

El tipo Template proporciona tres utilidades básicas al usuario:

1. Una Template puede invocar a otras Templates y por tanto pueden ser compartidas y reutilizadas entre programas EGL. En el listing 2.5 (línea 2) se aprecia cómo desde un programa EGL se llama a otro programa EGL.

2. El tipo Template permite al usuario definir el destino del texto generado (por ejemplo a una base de datos o ficheros para un proyecto en Visual Studio). En el listing 2.5 (línea 3) se presenta que el fichero de salida será un fichero de extensión .txt.

3. Y por último, proporciona un conjunto de operaciones que se usan para controlar el destino del texto generado. En el listing 2.5 (línea 3) se muestra cómo se elige el fichero destino para almacenar el texto generado con la plantilla *ClassNames.egl*.

```
1 [%
2 var t : Template = TemplateFactory.load("ClassNames.egl");
3 t.generate("Output.txt");
4 %]
```

Listing 2.5: Almacenar el nombre de cada Class en disco

Epsilon Object Language(EOL)

El principal objetivo de EOL es proporcionar un conjunto rehusable operaciones que son comunes en la gestión de modelos. El listing 2.6 muestra un ejemplo de operaciones EOL. No es un lenguaje orientado a objetos en el sentido de que no define clases de sí mismo pero sin embargo necesita gestionar objetos de los tipos definidos externamente a el (listing 2.6 líneas 3 y 7, en este caso objetos de tipo Integer). La línea 1 del listing 2.6 presenta un ejemplo de llamada a operaciones EOL, donde el primer elemento es un 1, de tipo Integer, que llama a la operación add1() y devuelve, tal como se aprecia en la línea 4, el valor de *self*+1 es decir 2. A continuación, se vuelve a tomar el valor 2, de tipo Integer, y se realiza la llamada a la operación add2() que devuelve el valor de *self*+2 es decir 4. Y dicho contenido se imprime por pantalla (instrucción println()).

```
1 1.add1().add2().println();
2
3 operation Integer add1() : Integer {
4   return self + 1;
5 }
6
7 operation Integer add2() : Integer {
8   return self + 2;
9 }
```

Listing 2.6: Ejemplo de operación EOL

De esta forma, EOL permite encapsular aquellas operaciones EGL que se vayan a utilizar a lo largo de las respectivas plantillas para evitar la repetición innecesaria de código.

2.7. Sumario

Durante este capítulo se han descrito los conceptos necesarios para comprender el ámbito y el alcance de este proyecto. Se ha descrito el caso de estudio que se utilizará a lo largo del documento además de qué es una línea de productos software y la tecnología TENTE utilizada para su desarrollo, el diseño orientado a características UML, la respuesta a las limitaciones de las clases parciales en C# mediante el uso del patrón slicer y la generación de código con Epsilon.

En el siguiente capítulo profundizaremos acerca de la fase Ingeniería de Dominio del proyecto.

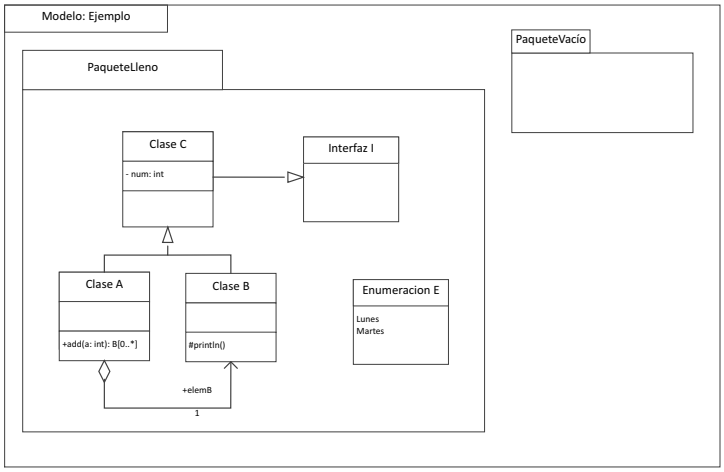

Figura 3.1: Ejemplo de Modelo UML simplificado

Capítulo 3

Ingeniería del Dominio

En este capítulo se describe la fase de *Ingeniería del Dominio* (en inglés, *Domain Engineering*) de nuestra línea de productos software. Dentro de dicha fase analizaremos cómo se transforman los elementos del modelo a código C#, profundizaremos en el desarrollo e implementación de los generadores de código junto con la explicación de varios ejemplos y concluiremos con la fase de pruebas de dicha etapa del proyecto.

3.1. Transformaciones de Modelo UML a C#

El primer paso para la transformación de modelo a código (*model-to-text*, M2T) es la identificación de los distintos elementos que nos podemos encontrar en un diagrama de clases UML y hallar el equivalente en código (C# en nuestro caso). Hay que tener en cuenta que la transformación no es trivial y es necesario procesar la información obtenida del modelo para

Elemento en UML	**Elemento en C#**
Modelo	Namespace del proyecto. Los namespaces permiten agrupar entidades tales como paquetes, clases, objetos y funciones bajo el mismo nombre. De esta forma, se pueden tener varios namespaces en el mismo proyecto que son independientes entre sí.
Paquete	Puede ser un directorio, con el mismo nombre que dicho paquete en el modelo, y que contenga tantos ficheros como clases o interfaces almacene en su interior. O un directorio vacío, con el mismo nombre que dicho paquete en el modelo, por cada paquete vacío que haya en el modelo UML.
Atributo	Cada atributo será tratado como una propiedad de C# con los correspondientes métodos getter y setter. Si el atributo tiene visibilidad *protected* o es *static* no poseerá los métodos getter y setter.
Parámetro	Después de identificar si es de retorno de una operación o de entrada a la misma se aplica la transformación al tipo de dato correspondiente.
Clase	Clase parcial C# siguiendo el patrón slicer, es decir <nombre del paquete al que pertenece>_<nombre de la clase>. Al crear la clase, añadir también los métodos de utilidad.
Clase Enumerada	Clase enumerada C#.
Interfaz	Interfaz C# implementada siguiendo el patrón slicer.
Operación	Método C# privado y renombrado siguiendo el patrón slicer de la forma <nombre del paquete al que pertenece>_<nombre de la operación>, para evitar posibles conflictos, todos los métodos serán virtuales. Los métodos *protected* no se cambiarán a privados para respetar la visibilidad requerida inicialmente por el usuario.
Constructor	Cada clase parcial correspondiente a una característica tendrá un método privado llamado <nombre del paquete al que pertenece>_init<nombre de la clase> que contendrá la porción de constructor que corresponde a la característica.
Asociación	*Asociación simple:* Se añade el atributo, simple o colección, de tipo Class a la clase destino. *Asociación bidireccional:* Dependiendo del tipo de bidireccionalidad (one to one, one to many o many to many) se añaden los atributos y métodos adicionales necesarios para implementar el correcto funcionamiento de dicha asociación.
Generalización	*Herencia simple:* Una clase hereda de otra clase. *Herencia múltiple:* Una clase hereda de varias clases y se debe realizar la transformación correspondiente mediante la creación de interfaces ya que en C# no se permite la herencia múltiple de varias clases pero sí de varias interfaces.

Cuadro 3.1: Transformación de elementos del modelo UML a código C#

generar el código adecuadamente. En la tabla 3.1 se encuentra un resumen de dicho proceso. A continuación se procede al análisis más detallado de cada uno de los elementos de dicha tabla, para ello nos apoyaremos en la figura 3.1.

```
File PaqueteLleno/B.cs
-----------------------------------------------------------
01 namespace Ejemplo{
02     partial class B: C{
03          ...
04          private virtual void B_initB ( ) {}
05          protected virtual PaqueteLleno_println ( ) {}
06     }
07 }

File PaqueteLleno/A.cs
-----------------------------------------------------------
08 namespace Ejemplo{
09     partial class A: C{
10          private B elemB;
11          public B elemB {
12              get { return this.elemB; }
13              set { this.elemB= value; }
14          }
15          ...
16          private virtual void A_initA ( ) {}
17          private virtual ISet<B> PaqueteLleno_add (int a) { }
18     }
19 }

File PaqueteLleno/C.cs
-----------------------------------------------------------
20 namespace Ejemplo{
21      partial class C: I{
22          private int num;
23          public int num {
24              get { return this.num; }
25              set { this.num= value; }
26          }
27          ...
28          private virtual void C_initC ( ) {}
29      }
30 }

File PaqueteLleno/I.cs
-----------------------------------------------------------
31 namespace Ejemplo{
32     partial interface I{
33          public virtual override bool Equals (Object o);
34          public virtual override int CompareTo (Object o);
35          public virtual override int GetHashCode ( );
36          public virtual override Type GetType ( );
37          public virtual override string ToString( );
38          private virtual void I_initI ( ) {}
39     }
40 }

File PaqueteLleno/E.cs
-----------------------------------------------------------
41 namespace Ejemplo{
42     enum E {
```

```
43          Lunes,
44          Martes,
45      };
46  }
```

Listing 3.1: Código generado para las clases y la interfaz del modelo de la figura 3.1

El modelo de la figura 3.1 es Ejemplo y por tanto cada clase del proyecto debería comenzar definiendo el namespace del modelo en cuestión mediante la línea de código C# tal como se aprecia en las líneas 1, 8, 20, 31 y 41 del listing 3.1.

En la figura 3.1 hay dos paquetes PaqueteLleno y PaqueteVacío, por tanto, en el directorio destino donde se generan los ficheros del modelo deberán aparecer dos carpetas con dichos nombres. La carpeta PaqueteLleno contendrá en su interior cuatro ficheros denominados A.cs, B.cs, C.cs, I.cs y E.cs, uno por cada clase, clase enumerada (listing 3.1 41-46) o interfaz que se encuentra en su interior, mientras que la carpeta PaqueteVacío no contendrá ningún archivo en su interior.

Tal como se aprecia en la figura 3.1, la clase A del paquete PaqueteLleno, tiene un atributo llamado num por lo que se genera una propiedad con sus respectivos métodos getter y setter tal como se muestra en el listing 3.1 en las líneas 22-26.

La figura 3.1 presenta la clase A con una operación add que tiene un parámetro a: int y retorna una colección de elementos de tipo B (listing 3.1 línea 17). De la misma forma, la clase B tiene una operación println de carácter *protected* y por tanto su visibilidad no se transforma en *private* (listing 3.1 línea 5). Con este ejemplo quedan ilustrados los puntos de operación y parámetro descritos en la tabla 3.1.

Para la generación de clases e interfaces de la figura 3.1, el resultado sería el mostrado en las líneas 2, 9, 21 y 32 del listing 3.1. Se aprecia también las herencias correspondientes.

Aunque no esté reflejado en el modelo UML añadimos a cada clase, o interfaz, del modelo añadimos un constructor (listing 3.1 líneas 4, 16, 28 y 38) y unos métodos de utilidad (listing 3.1 líneas 33-37).

Por último, la asociación simple de las clases A y B se traduce con las líneas de código descritas en las líneas 10-14 del listing 3.1.

Con esto queda explicado más detalladamente la transformación de modelo UML a código C# descrita en la tabla 3.1. Se han omitido la herencia múltiple y las asociaciones bidireccionales por su complejidad. En la siguiente sección se profundizará en la implementación y creación de las transformaciones de modelo UML a código C#.

3.2. Generadores de Código C#

Tras explicar a grandes rasgos las transformaciones de modelo a código en la sección **??**, se procede a comentar la implementación de los generadores de código correspondientes. La figura 3.2 muestra la jerarquía de los generadores de código implementados.

El punto de partida es el generador de código llamado ProjectCreation, a partir de él se invoca a la plantilla ClassFilesCreation que determina si un elemento es:

- Una clase o interfaz, en tal caso llamaría a su vez al generador de código ClassCreation.
- Una clase con herencia múltiple, en este caso se requiere un tratamiento especial que veremos más adelante en detalle, en tal caso llamaría a su vez a los generadores de código ClassCreationMultipleHierarchyCase, ChildClassHierarchyCase e InterfaceCreation.

Una vez se ha determinado el tipo de elemento de los descritos en la enumeración anterior, se procede a generar el fichero fuente correspondiente a dicho elemento. Para ello, y en un orden secuencial estricto, se hacen llamadas a las plantillas descritas en forma de árbol en la figura 3.2. Por ejemplo, para la generación de una clase el proceso sería tal como se muestra a continuación:

1. ProjectCreation
2. ClassFilesCreation, se determina que es una clase.
3. ClassCreation, se indica la ruta para guardar el fichero que se va a generar.
4. ModelCreation, genera el namespace del proyecto.
5. ClassDeclaration, genera la declaración de la clase parcial.
6. PropertiesGeneration, genera las propiedades de la clase.
7. MethodsCreation, genera los métodos de la clase.
8. UtilityMethodsCreation, genera los métodos de utilidad de la clase.
9. Una vez generados todos los elementos de la clase, se genera el fichero fuente correspondiente.

Después de haber generado todos los paquetes, clases e interfaces del proyecto se deben generar los ficheros necesarios para que dicho proyecto pueda ser abierto con el programa Visual Studio sin problemas. Para ello es necesario generar una serie de ficheros específicos de dicha plataforma, los ficheros se describen a continuación:

- SlnFileCreation, genera el fichero .sln que contiene información del entorno del sistema.
- CsprojectFilesCreation, genera el fichero .csproj correspondiente para incluir los directorios creados en el proyecto Visual Studio.
- AssemblyInfoFileCreation, genera el fichero .cs que contiene la información general sobre las directivas de ensamblado.

Aunque no aparezca en el diagrama, todos los generadores de código utilizan además el fichero *Operations.eol*, que es un fichero EOL que dispone de operaciones básicas que se utilizan recurrentemente en el resto de generadores, por nombrar algunos ejemplos, dicho fichero contiene funciones del estilo:

- *abstract*, retorna el texto abstract cuando el elemento sobre el que está siendo utilizado es abstracto.
- *typeCollection*, retorna el tipo de colección correspondiente al elemento sobre el que se está actuando.
- *hierarchy*, retorna las clases de las que hereda el elemento en cuestión.

Adicionalmente a los generadores de código EGL y EOL, en los sucesivos párrafos se habla en detalle de los Java Tools que se han elaborado para hacer más gráfico y sencillo al usuario el proceso de ejecución de los generadores de código.

Java Tool

Un Java Tool es una porción de código java que tras ser empaquetada como plug-in se puede incorporar en el directorio de instalación de Epsilon y utilizar las funciones implementadas en las plantillas de generación de código creadas. Se han desarrollado cinco Java Tools en total:

- pluginCreateDirectories, genera los directorios en el sistema.
- pluginCreateMessageWindow, genera la ventana con los errores y/o información oportunos (figura 3.5).
- pluginCreateSelectDirectoryWindow, genera un entorno de exploración por el equipo para elegir el destino de la generación del proyecto (figura 3.4).
- pluginCreateWindowAndShowMessages, genera una ventana al final de la ejecución con información para el usuario (figura 3.6).

- pluginWriteInFile, genera el fichero que contendrá la información para mostrar al usuario al final la ejecución, también implementa funciones para borrar ficheros y/o directorios en el caso de que fuera oportuno.

- pluginChooseModel, genera la ventana de selección de modelo para aquellos proyectos que tienen varios (figura 3.3).

Para utilizar los Java Tools en los generadores de código es necesario implementar funciones EOL para cada una de las funciones descritas ellos. El listing 3.2 muestra un ejemplo de cómo se realiza dicho proceso, en las líneas 1-4 se describe la creación de la función a utilizar en los generadores de código, en la línea 5 se importa dicho fichero EOL para hacer uso de las funciones implementadas en el generador ProjectCreation, mientras que la línea 7 muestra la forma de uso de la función writeInFile.

```
File Operations.eol
-----------------------------------------------------------
01 operation writeInFile(path:String, message:String) {
02     var sampleTool = new Native("pluginWriteInFile.WriteInFile");
03     sampleTool.writeInFile(path, message);
04}

File ProjectCreation.egl
-----------------------------------------------------------
05 import "Operations.eol"
06 ...
07 writeInFile(path+"\\log.txt", message);
08 ...
```

Listing 3.2: Uso de un Java Tool en los generadores de código

Una vez explicado el funcionamiento de los generadores de código y los java tools implementados, en las siguientes secciones se procederá a explicar de forma detallada varios ejemplos de generadores. Primero uno sencillo que sea fácil de comprender para el lector no experto en el tema y después dos casos más complejos.

3.3. Ejemplo de Generación de Código C#: Caso Sencillo

Para introducir al lector en la implementación de los generadores de código, vamos a analizar en detalle uno de los generadores de código más sencillos: MethodsCreation, el fichero fuente aparece en el listing 3.3. Vamos a proceder al análisis detallado del mismo:

- Líneas 1-3, generadores de código que utiliza y fichero Operations.eol que contiene las funciones básicas comunes a los generadores de código.

- Línea 4, descripción de la función que retornará el texto generado.

- Línea 6, texto correspondiente al constructor de la clase de la forma <nombre del paquete>_init<nombre de la clase>.
- Líneas 8-28, tratamos una a una todas las operaciones descritas en elemento actual (clase o interfaz).
- Líneas 9-22, en cada operación recorremos todos y cada uno de los parámetros.
- Líneas 11-13, si la operación no tiene definido un tipo, es decir, si el usuario ha obviado especificar si la función devuelve una colección, un entero, un elemento de una clase, etc, por defecto se trata como una operación void (operación que no retorna ningún valor).
- Línea 15, si la operación tiene un tipo de retorno definido, comprobamos si dicho parámetro es de retorno.
- Línea 16, si el parámetro es de retorno pero no está definido vuelve a ser tratada como una operación void.
- Línea 18, si el parámetro es de retorno y tiene un tipo definido se trata de una operación que sí retorna un valor.
- Línea 24, si la operación que está siendo analizada retorna un valor, se añade a la lista de operaciones que devuelven un valor.
- Línea 26, si la operación que está siendo analizada no retorna un valor, se añade a la lista de operaciones que no devuelven un valor.
- Línea 29-38, añadir al string resultado la información correspondiente a los métodos de la clase actual que no retornan ningún valor (métodos void).
- Línea 31, si el método no tiene un nombre definido, se otorga un nombre por defecto.
- Línea 35, se realizan llamadas a los generadores de código para obtener los parámetros de la función.
- Línea 39-48, de manera análoga a las operaciones que no retornan ningún valor, se procede a añadir al string resultado los métodos de la clase que sí retornan un valor.
- Línea 49, se retorna el string con todos los métodos de la clase o interfaz actual.

```
01 [%import "ReturnParameterCreation.egl";
02 import "ParametersCreation.egl";
03 import "../Operations.eol";
04 operation Element classMethods(currentPackage: String, path: String)
                    : String {
05  ...
06  opers=private()+void()+currentPackage+"_init"
        +self.firstToUpperCase()+"␣(␣)␣{}\n\t\t";
07  ...
08  for (oper in self.getOperations()){
09      for (par in oper.ownedParameter){
10          ...
11          if (oper.type==null){
12              isReturn=false;
13          }else{
14              if (par.direction.toString().equals("return")){
15                  if (not par.type.name.isDefined()){
16                      isReturn=false;
17                  }else{
18                      isReturn=true;
19                  }//if-par-type
20              }//if-par-direction
21          }//if-oper-type
22      }//for-parameters
23      if (isReturn){
24          operations_return.add(oper);
25      }else{
26          operations_void.add(oper);
27      }
28  }//for-operations
29  for (oper in operations_void) {
30      if (oper.name==""){
31          methodname="method_"+iter;
32      }else{
33          methodname=oper.name;
34      }
35      opers=opers+oper.visibility()+oper.abstract()+oper.esStatic()
              +virtual()+void()+currentPackage+"_"+methodname
              +"␣("+oper.parameters(currentPackage, path)+")␣{}\n\t\t";
36      // Increase the iterator
37      iter=iter+1;
38  }
39  for (oper in operations_return) {
40      if (oper.name==""){
41          methodname="method_"+iter;
42      }else{
43          methodname=oper.name;
44      }
45      opers=opers+oper.visibility()+oper.abstract()+oper.esStatic()
             +virtual()+oper.returnParameter(currentPackage, path)
             +"␣"+currentPackage+"_"+methodname
             +"␣("+oper.parameters(currentPackage, path)
             +")␣{}\n\t\t";
46      // Increase the iterator
47      iter=iter+1;
48  }
49  return opers;
50 }%]
```

Listing 3.3: Implementación del generador de código MethodsCreation

Un vez explicado un ejemplo sencillo, la siguiente sección 3.4 explica ejemplos más complejos que quedan a la curiosidad del lector.

3.4. Ejemplo de Generación de Código C#: Caso Complejo

Antes de comenzar a exponer los ejemplos más complejo, vamos a introducir aquellos conceptos que pasamos por algo en la sección 3.1 dada su complejidad: las relaciones bidireccionales y la herencia múltiple.

Relaciones Bidireccionales

Las relaciones bidireccionales son aquellas en las que ambas clases relacionadas disponen de atributos de la clase opuesta. Existen tres tipos de relaciones bidireccionales:

- *Bidireccionalidad one to one*, en las que cada clase recibe un elemento de la clase opuesta (figura 3.7, Mujer-Marido).
- *Bidireccionalidad one to many*, en la que una de las clases recibe un elemento de la clase destino y la otra recibe una colección de elementos de la clase opuesta (figura 3.7, Alumno-Curso).
- *Bidireccionalidad many to many*, en la que ambas clases reciben colecciones de la clase opuesta (figura 3.7, Trabajadores-Proyectos).

Llegados a este punto, basándonos en los ejemplos de la figura 3.7 nos surgen algunas preguntas:

1. ¿Si una mujer A tiene un esposo B y ese esposo B tiene una mujer que no sea A?
2. ¿Si un alumno tiene asignado un curso pero ese curso no tiene a dicho alumno en su lista de alumnado?
3. ¿Si un trabajador tiene asignado un proyecto pero en el listado de trabajadores dicho trabajador no aparece?

¿Existe alguna forma de poner solución a estos problemas?, la respuesta es sí, a la hora de tratar este tipo de relaciones bidireccionales en los generadores de código correspondientes no basta con generar las propiedades tal como hemos estado haciendo hasta ahora sino que debemos implementar las propiedades a generar de una forma cuidadosa y exhaustiva para que no se produzcan este tipo de incoherencias, incluso recurriendo a la creación de métodos adicionales. Veamos a grandes rasgos en la tabla 3.3 cómo sería la implementación de la relación bidireccional one to one: El siguiente paso sería pasar dicha tabla a código C# y comprobar que funciona correctamente. Se sigue un proceso análogo para el resto de relaciones bidireccionales.

	Mujer soltera	**Mujer casada**
Hombre soltero	Se casan el hombre soltero y la mujer soltera.	La mujer casada se divorcia. El antiguo marido de la mujer divorciada queda soltero. La mujer divorciada y el hombre soltero se casan.
Hombre casado	El hombre casado se divorcia. La antigua mujer del marido divorciado queda soltera. Se casan la mujer soltera y el hombre divorciado.	El hombre casado se divorcia. La antigua mujer del marido divorciado queda soltera. La mujer casada se divorcia. El antiguo marido de la mujer divorciada queda soltero. Se casan la mujer divorciada y el hombre divorciado.

Cuadro 3.2: Solución para evitar incoherencias en el código C# en la bidireccionalidad one to one

Herencia múltiple

Los modelos UML admiten la herencia múltiple de clases pero lenguajes como C# no, por lo que hay que transformar el modelo inicial y adaptarlo para funcione correctamente. Podemos encontrarnos con un conjunto de clases como el descrito en la figura 3.8 donde se puede apreciar que la clase Profesor Universitario presenta herencia múltiple de las clases Profesor e Investigador, por tanto, debemos procesar la información de forma que el código generado funcione de la forma esperada en C#, las modificaciones que debemos realizar son:

- Por cada una de las clases padre, Profesor e Investigador, se generan sendas interfaces Interfaz_Profesor e Interfaz_Investigador.
- Las clases padre solo contienen los métodos de la clase y heredan de sus respectivas interfaces.
- Las interfaces correspondientes a las clases padre tienen los métodos y las propiedades de la respectiva clase.
- La clase hija, Profesor Universitario, hereda ahora de las interfaces Interfaz_Profesor e Interfaz_Investigador en lugar de heredar de las clases Profesor e Investigador como ocurría en el modelo inicial.
- La clase hija incorpora, además de sus propiedades y métodos, posee tantas colecciones como clases padre tenga, en el ejemplo profesores e investigadores.

Una vez concluidos los generadores de código, se debe proceder a comprobar que funciona como se espera, en eso consiste la fase de pruebas que se describe en la siguiente sección.

3.5. Pruebas con EUnit

EUnit es un sistema de pruebas [**?**] unitarias que proporciona assertions para la comparación de modelos, archivos y directorios. Una *assertion* es un predicado, verdadero o falso, colocado en un programa para indicar que el desarrollador cree que el predicado es siempre cierto en ese lugar. Las pruebas pueden reutilizarse con diferentes conjuntos de modelos y datos de entrada, y las diferencias entre los modelos esperados y los reales pueden ser visualizadas gráficamente.

Al comenzar la fase de pruebas me encontré con el problema inicial de que EUnit no tenía implementada la comparación de fragmentos de texto en los ficheros generados que era precisamente la manera de comprobar que los generadores de código funcionaban correctamente. De tal forma procedí a realizar una petición en el foro de la plataforma e incorporaron una nueva assertion denominada assertLineWithMatch que permitía comprobar si un fichero disponía de un determinado fragmento de texto o no (líneas 1-4 del listing 3.4). También era necesario comprobar que los ficheros y directorios se creaban correctamente por lo que la assertion assertEqualDirectories es válida para tal fin (líneas 6-9 del listing 3.4). Y por último, se debe comprobar que las plantillas lanzan las excepciones oportunas para los casos no válidos, mediante la combinación de las instrucciones assertError y runTarget, se puede comprobar si el fichero deseado lanza o no una excepción (líneas 11-16 del listing 3.4). Con estas assertions se procede a realizar todos los casos de prueba descritos en la tabla 3.3 con resultados satisfactorios.

```
01 @test
02 operation classWithNameAndWithoutType() {
03    assertLineWithMatch(path+"Data\\src\\BasicGraph\\Edge.cs",
                          "partial␣class␣Edge");
04 }
05 ...
06 @test
07 operation emptyPackage() {
08    assertEqualDirectories(path+"Data\\src\\PaqueteVacio",
                             path+"\\Data\\src\\PaqueteVacio");
09 }
10 ...
11 @test
12 operation thowsExceptions() {
13    ...
14    assertError(runTarget(pathTemplates+'\\ParametersCreation.egl'));
15    ...
16 }
```

Listing 3.4: Pruebas de los generadores de código con EUnit

	Casos válidos	Casos no válidos
Clase	Clase con nombre. Clase tipo abstract. Clase sin tipo. Clase que hereda de una o varias clases. Clase que hereda de una o varias interfaces. Clase que hereda de clases e interfaces. Clase sin propiedades. Clase sin métodos. Clase sin propiedades ni métodos. Clase con propiedades. Clase con métodos. Clase con propiedades y métodos.	Clase fuera de un paquete. Clase sin nombre. Clase enumerada.
Paquete	Paquete con nombre. Paquete con clases e interfaces en su interior. Paquete vacío. Paquete dentro de otro paquete (recursividad).	Paquete sin nombre.
Clase Enumerada	Clase enumerada con nombre. Clase enumerada con literales. Clase enumerada vacía.	Clase enumerada sin nombre.
Interfaz	Interfaz con nombre. Interfaz sin métodos. Interfaz con métodos.	Interfaz sin nombre. Interfaz fuera de paquete.
Propiedad	Propiedad con nombre. Propiedad sin nombre (se debe poner uno por defecto). Propiedad estática (no lleva métodos getter ni setter). Propiedad protected (no lleva métodos getter ni setter). Propiedad no estática (lleva métodos getter ni setter). Propiedad es una colección. Propiedad es una asociación simple. Propiedad es una asociación bidireccional one to one. Propiedad es una asociación bidireccional one to many. Propiedad es una asociación bidireccional many to many.	Propiedad sin tipo. Asociaciones sin multiplicidad.
Método	Método con nombre. Método sin tipo (se debe poner void por defecto). Método sin tipo (se debe poner void por defecto) y sin parámetros. Método sin tipo (se debe poner void por defecto) y con parámetros. Método void sin parámetros. Método void con parámetros. Método retorna tipo primitivo sin	Método sin nombre.

3.6. Sumario

Durante este capítulo se han descrito la fase de *Ingeniería del Dominio* de nuestra línea de productos software. Dentro de dicha fase se ha analizado cómo se transforman los elementos del modelo a código C#, el desarrollo e implementación de los generadores de código junto con la explicación de varios ejemplos y se ha concluído con la fase de pruebas.

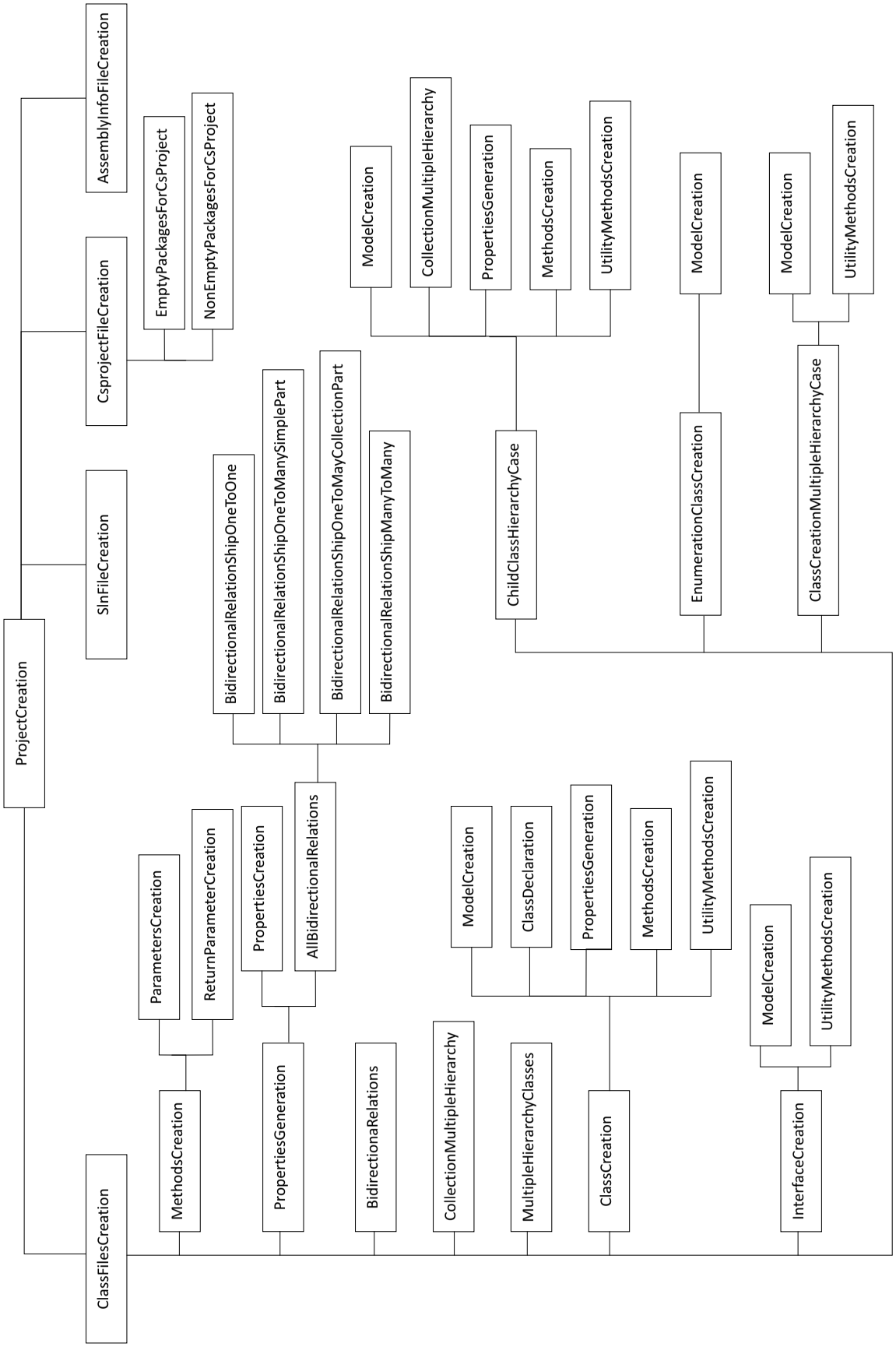

Figura 3.2: Jerarquía de los generadores de código implementados en la fase de Ingeniería de Dominio

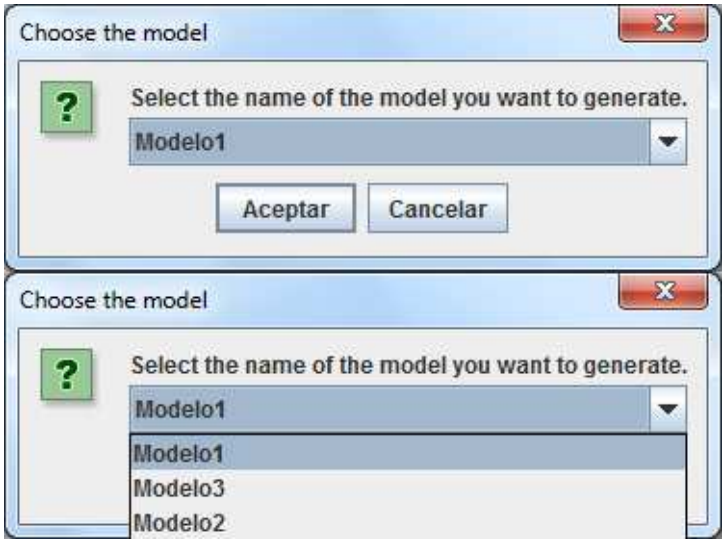

Figura 3.3: Ventana de selección de modelo

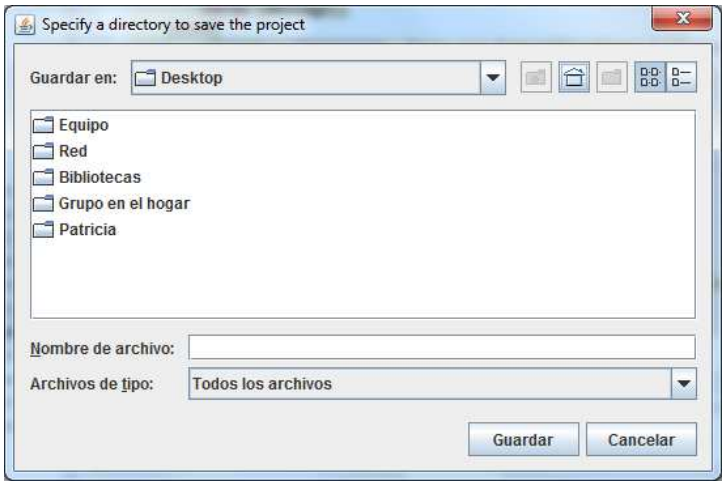

Figura 3.4: Selección de directorio donde emplazar el proyecto a generar

Figura 3.5: Ventana de error

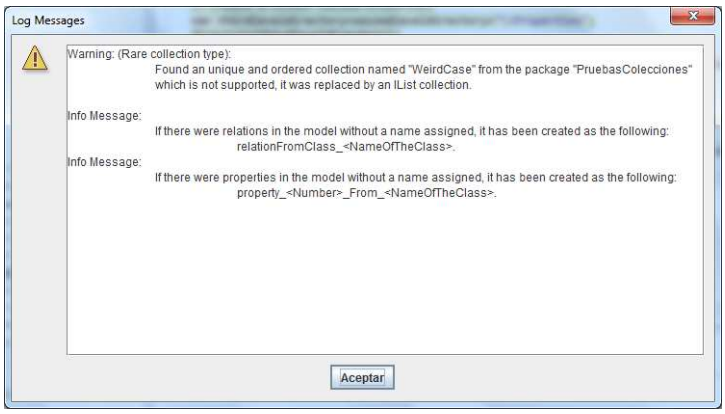

Figura 3.6: Ventana al final de la generación de código

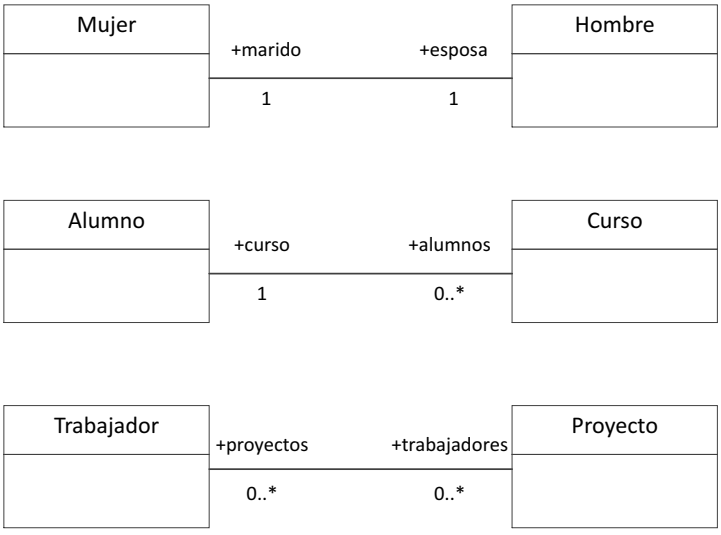

Figura 3.7: Tipos de relaciones bidireccionales

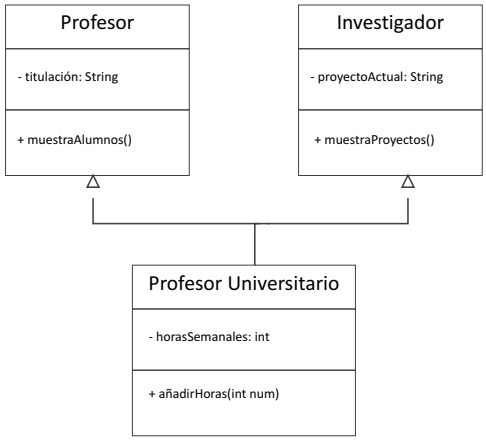

Figura 3.8: Tipos de relaciones bidireccionales

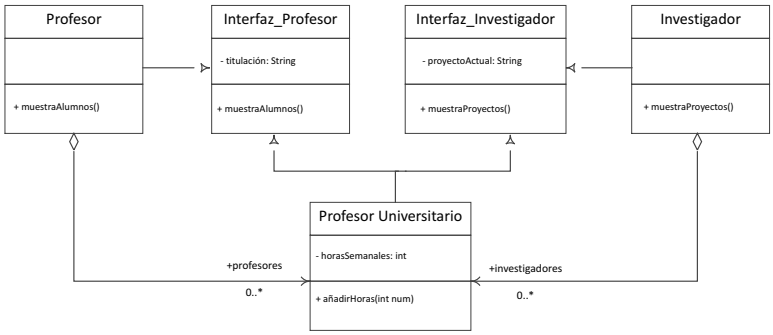

Figura 3.9: Tipos de relaciones bidireccionales

Capítulo 4

Ingeniería de Aplicación

En este capítulo se describe la fase de *Ingeniería de Aplicación* (en inglés, *Application Engineering*) de nuestra línea de productos software. Dentro de dicha fase analizaremos el algoritmo necesario para la generación del producto específico, profundizaremos en el desarrollo e implementación de los generadores de código necesarios para tal fin y concluiremos con la fase de pruebas de dicha etapa del proyecto.

4.1. Algoritmo para la implementación del producto específico

En el capítulo 3, se ha descrito el desarrollo de la generación de código para todos los elementos del modelo de entrada. En la presente sección de describirá de forma detalla el proceso de generación de código para un producto específico, para ello nos apoyaremos en la figura **??**.

El primer paso es crear un *profile* que nos permita diferenciar las características del modelo de la selección de características requeridas para elaborar el producto específico. Un profile es un mecanismo genérico de extensión que permite personalizar los modelos UML para un dominio particular. En nuestro caso aplicamos dicho profile a la metaclase paquete tal como se aprecia en la figura **??** y denominamos al profile CasoConcreto, se ha sombreado en el modelo aquel paquete al que le ha sido aplicado el profile CasoConcreto para poder diferenciarlo visualmente del resto de paquetes del modelo.

Partiendo del paquete que ha sido caracterizado con el profile, es necesario calcular todas las rutas del modelo que son requeridas, en la figura **??** se aprecia cómo hay una única ruta formada por los paquetes: Cinco → Cuatro → Uno.

Una vez reconocida la ruta, o conjunto de rutas, se procede a identificar el total de clases a implementar, en qué paquetes (de entre los de la ruta seleccionada) aparecen y las respectivas operaciones que implementan. En

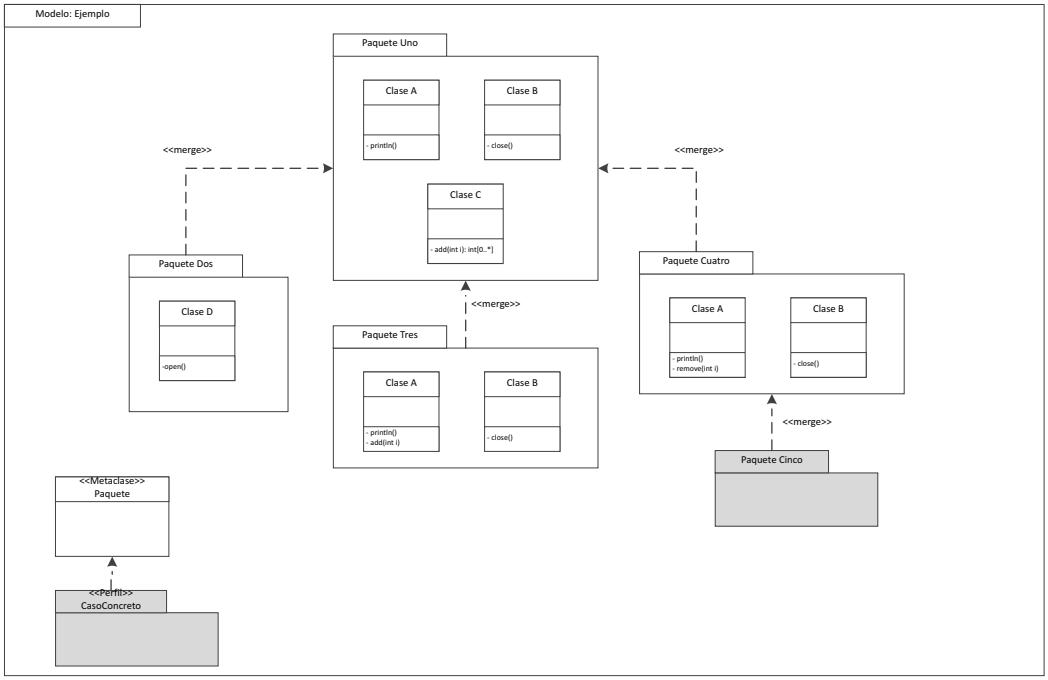

Figura 4.1: Ejemplo de apoyo para la explicación del algoritmo

el ejemplo de la figura **??** tenemos tres clases a implementar: A, B y C.

- Clase A, está implementada en el paquete Uno y Cuatro.
- Clase B, está implementada en el paquete Uno y Cuatro.
- Clase C, está implementada en el paquete Uno.

Nos encontramos así con dos conflictos, puesto que la misma clase (en nuestro caso la clase A y la B) está implementadas en dos paquetes diferentes. En estos casos, debemos buscar las operaciones que dicha clase implementa en ambos paquetes, de esta forma tenemos que:

- Clase A, implementa las operaciones println() y remove (int i) en el paquete Cuatro, mientras que en el paquete Uno implementa la operación println().
- Clase B, implementa la operación close() en el paquete Cuatro, mientras que en el paquete Uno implementa la operación open().

En este punto tenemos un conflicto, en la operación println() de la clase A, ya que tenemos dos versiones del método la implementada en el paquete Cuatro y la implementada en el paquete Uno, nos quedamos con la versión más profunda del método es decir, con la versión implementada en el paquete Cuatro.

Para la generación del constructor de las clases basta con comprobar el paquete más profundo donde está implementada dicha clase, en nuestro ejemplo:

- Clase A, implementada en el paquete Cuatro.
- Clase B, implementada en el paquete Cuatro.
- Clase C, implementada en el paquete Uno.

```
File Cinco/A.cs
-------------------------------------------------------
01 namespace My_Ejemplo{
02     public class A{
03        public virtual void A ( ) {
04             Cuatro_A_initA ( );
05        }
06        public virtual println ( ) {
07                Cuatro_println ( );
08        }
09        public virtual remove ( int i ) {
10             Cuatro_remove ( i );
11        }
12     }
13 }

File Cinco/B.cs
-------------------------------------------------------
14 namespace My_Ejemplo{
15     public class B{
15          public virtual void B_initB ( ) {
16              Cuatro_B_initB ( );
17          }
18          public virtual close ( ) {
19              Cuatro_close ( );
20          }
21     }
22 }

File Cinco/C.cs
-------------------------------------------------------
23 namespace My_Ejemplo{
24     public class C{
25          public virtual void C_initC ( ) {
26              Uno_C_initC ( );
27          }
28          public virtual ISet<int> add ( int i) {
29              Uno_add ( i );
30          }
31     }
32 }
```

Listing 4.1: Código generado para el producto específico seleccionado en la figura **??**

Procedemos entonces a generar el código para la versión clean de tanto la clase como los métodos tal como se aprecia en el listing 4.1.

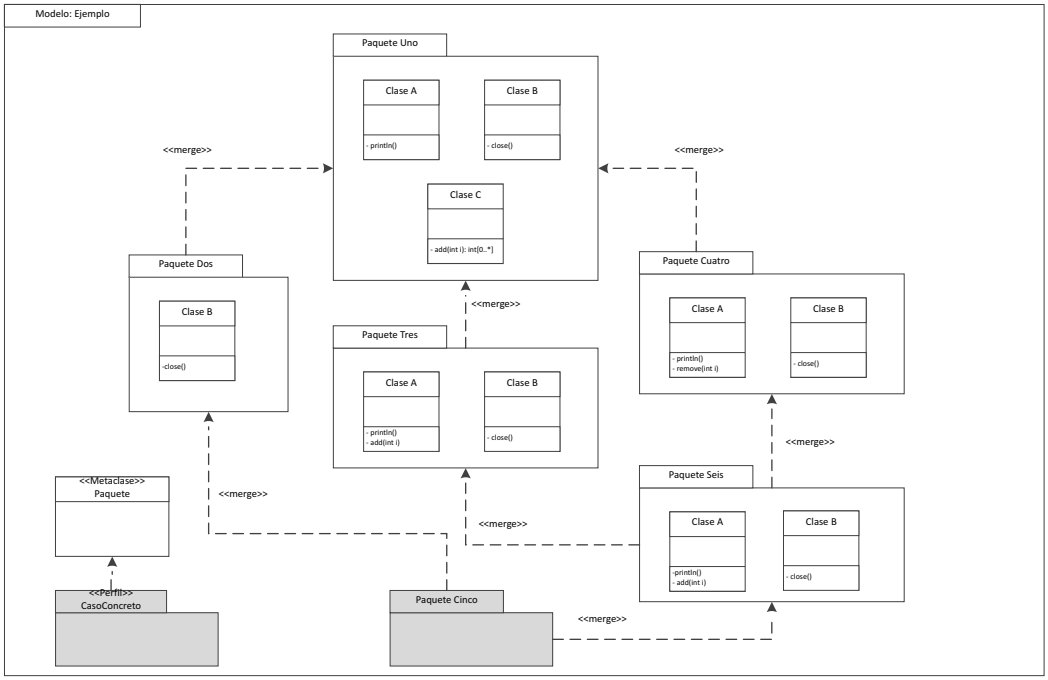

Figura 4.2: Ejemplo complejo de apoyo para la explicación del algoritmo

Renombramos el modelo aplicado añadiendo un "My_"(líneas 1, 14 y 23) para denotar que ese es el modelo que contiene la implementación específica del producto seleccionado.

Tal como comentamos anteriormente, la llamada interna de los constructores se corresponde con la de aquel paquete más profundo en la selección específica que implemente dicha clase (líneas 4, 16 y 26).

Y tras resolver los conflictos con aquellos métodos implementados en varios paquetes (en nuestro ejemplo el método print() de la clase A) obtenemos el código descrito en el listing 4.1.

Analizamos ahora un ejemplo más complejo de generación de código de un producto específico, supongamos ahora el ejemplo descrito en la figura 4.2.

En este caso tenemos tres rutas: Cinco → Seis → Tres → Uno, Cinco → Seis → Cuatro → Uno y Cinco → Dos → Uno.

Debemos por tanto analizar los caminos independientes, en este caso hay dos, por un lado el camino Cinco → Dos → Uno y por otro el formado por Cinco → Seis (que se bifurca a su vez en dos).

A la hora de extraer la implementación interna de los métodos, debemos incluir ambas ramas puesto que son independientes entre sí y por tanto sus versiones de los métodos no entran en conflicto. De esta forma, y tal como se describe en el listing 4.2, para la clase B tanto el método close() como el constructor se realizan llamadas internas a ambas versiones de dicho paquete

(líneas 15-22).

Para la implementación interna de la clase A, procedemos de manera análoga al ejemplo **??**, es decir, la clase A implementará los métodos print(), add(int i), remove() y el constructor (listing 4.2, líneas 3-14). Como los caminos Cinco → Seis → Tres → Uno y Cinco → Seis → Cuatro → Uno son dependientes entre sí, debemos seleccionar una vez más las versiones más profundas de cada método que entre en conflicto.

```
File Cinco/A.cs
-------------------------------------------------------
01 namespace My_Ejemplo{
02     public class A{
03          public virtual void A ( ) {
04              Seis_A_initA ( );
05          }
06          public virtual println ( ) {
07              Seis_println ( );
08          }
09          public virtual add ( int i ) {
10              Seis_add ( i );
11          }
12          public virtual remove ( int i ) {
13              Cuatro_remove ( i );
14          }
15     }
16 }

File Cinco/B.cs
-------------------------------------------------------
14 namespace My_Ejemplo{
15     public class B{
15          public virtual void B_initB ( ) {
16              Dos_B_initB ( );
17              Seis_B_initB ( );
18          }
19          public virtual close ( ) {
20              Dos_close ( );
21              Seis_close ( );
22          }
23     }
24 }

File Cinco/C.cs
-------------------------------------------------------
25 namespace My_Ejemplo{
26     public class C{
27          public virtual void C_initC ( ) {
28              Uno_C_initC ( );
29          }
30          public virtual ISet<int> add ( int i) {
31              Uno_add ( i );
32          }
33     }
34 }
```

Listing 4.2: Código generado para el producto específico seleccionado en la figura 4.2

Al igual que como se ha comentado hasta ahora, también podemos incluir más de un paquete con profile para generar varios productos específicos,

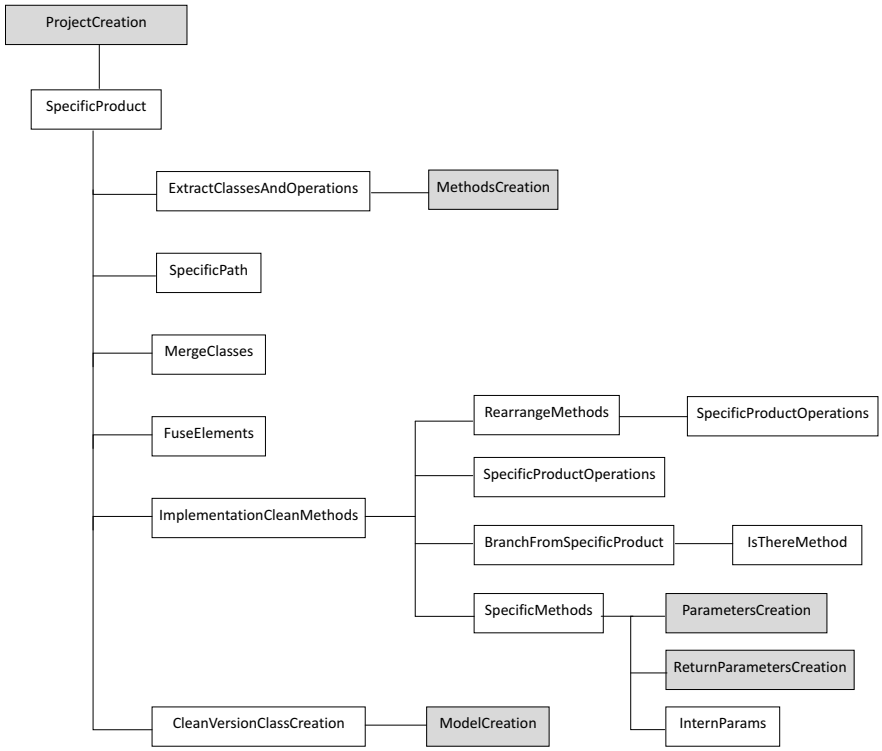

Figura 4.3: Jerarquía de los generadores de código implementados en la fase de Ingeniería de Aplicación

para cada uno de ellos se creará la implementación específica seleccionada, las soluciones son obviamente independientes entre sí.

Así pues queda explicado en detalle el proceso de generación de código para un producto específico del modelo UML a código C#. En la siguiente sección se profundizará en la implementación y generación de código C#.

4.2. Generadores de Código C#

Tras explicar en detalle el algoritmo necesario para obtener las versiones clean de las clases y métodos en la sección **??**, se procede a comentar la implementación de los generadores de código correspondientes. La figura 4.3 muestra la jerarquía de los generadores de código implementados.

El punto de partida es idéntico al utilizado para la fase de *Ingeniería del Dominio*; es decir, el generador de código llamado ProjectCreation, a partir del cual se invoca a la plantilla SpecificProduct que genera los ficheros fuente correspondientes a la implementación del caso concreto determinado por el profile del modelo. Para ello se hacen llamadas a las plantillas descritas en forma de árbol en la figura 4.3. Se procede a explicar brevemente el

funcionamiento de cada una de ellas:

- MergeClasses, genera un listado de los paquetes del modelo y los paquetes directamente relacionados con los mismos.
- SpecificPath, genera un conjunto que incluye aquellos paquetes que necesitan ser implementado para el camino específico seleccionado.
- ExtractClassesAndOperations, genera un listado con cada paquete, por el camino específico, y sus correspondientes clases y versiones clean de las operaciones que implementa.
- FuseElements, fusiona las operaciones de las clases, con el mismo nombre, que están en paquetes diferentes.
- ImplementationCleanMethods, genera la implementación final de los métodos clean (con sus correspondientes implementaciones internas).
- BranchFromSpecificProduct, extrae las versiones más profundas de cada método en una rama concreta.
- IsThereMethod, indica si un método está implementado en el paquete dado.
- RearrangeMethods, selecciona de todas las ramas del proyecto específico aquellas versiones de los métodos más profundas y elimina las redundancias mediante el análisis de la alcanzabilidad.
- SpecificMethods, genera la implementación para la versión clean de cada clase y la implementación interna de sus métodos.
- InternParams, genera los parámetros para las llamadas internas de los métodos.
- CleanVersionClassCreation, una vez creados los elementos a generar, esta plantilla genera la estructura completa del fichero resultante.
- SpecificProductOperations, implementa funciones de uso recurrente a lo largo del resto de plantillas para simplificar el proceso. Entre las funciones implementadas está la accesibilidad entre dos paquetes o la generación de los paquetes para el caso específico a analizar.

Una vez explicado el funcionamiento de los generadores de código, en la siguiente sección se procederá a explicar la fase de pruebas de esta etapa del proyecto.

Casos válidos	Casos no válidos
Un profile y un único camino. Un profile y varios caminos independientes. Un profile y varios caminos dependientes. Varios profiles y varios caminos independientes. Varios profiles y varios caminos dependientes.	Paquetes recursivos.

Cuadro 4.1: Casos de prueba para la fase de Ingeniería de la Aplicación

4.3. Pruebas

A diferencia de la fase de Ingeniería del Dominio, en esta fase no se utiliza EUnit para realizar las pruebas ya que la flexibilidad de dicha herramienta no se ajusta a las comprobaciones que deben realizarse. De tal forma se procede a crear todos los casos descritos en el cuadro 4.1 como modelos independientes y se prueba uno a uno si el código generado para los productos específicos se corresponde con el esperado.

4.4. Sumario

Durante este capítulo se han descrito la fase de *Ingeniería de Aplicación* de nuestra línea de productos software. Dentro de dicha fase se ha analizado el algoritmo necesario para la generación del producto específico, se ha profundizado también en el desarrollo e implementación de los generadores de código necesarios para tal fin y se ha concluído con la fase de pruebas de dicha etapa del proyecto.