FACULTAD DE CIENCIAS

UNIVESIDAD DE CANTABRIA

Proyecto Fin de Carrera

Desarrollo de un entorno dirigido por modelos para el desarrollo de líneas de productos software para la plataforma .NET

(Development of a model-driven development enviroment for software product lines on the .NET platform)

Para acceder al Título de
INGENIERO EN INFORMÁTICA

Autor: Patricia Abascal Fernández
Julio 2013

FACULTAD DE CIENCIAS

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por:	Patricia Abascal Fernández
Director del PFC:	Pablo Sánchez Barreiro
Título:	Desarrollo de un entorno dirigido por modelos para el desarrollo de líneas de productos software para la plataforma .NET
Title:	Development of a model-driven development enviroment for software product lines on the .NET platform

Presentado a examen el día:

para acceder al Título de
INGENIERO EN INFORMÁTICA

Composición del Tribunal:

Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

II

Agradecimientos

A mi madre y mi abuela por haberme apoyado en todas mis decisiones académicas y por estar siempre conmigo durante todos estos añoos.

A mi director Pablo Sánchez por ofrecerme la oportunidad de realizar este proyecto y por guiarme y aconsejarme en todo su desarrollo.

A todos los profesores que desde pequeña han sabido transmitirme sus conocimientos y en especial a los profesores de Ingeniería Informática de la Universidad de Cantabria.

Al Ministerio de Educación por darme la oportunidad de estudiar la carrera que siempre quise gracias a las becas que me han otorgado.

Resumen

Dentro del Departamento de Matemáticas, Estadística y Computación se han desarrollado con anterioridad una serie de técnicas para la implementación y configuración de líneas de productos software para la plataforma .NET basándose en las clases parciales de C#. Dichas técnicas se condensan en el denominado *Slicer Pattern*. No obstante, la aplicación de dicho patrón de forma manual implica una serie de tareas manuales y repetitivas.

El objetivo de presente proyecto fin de carrera es desarrollar una serie de generadores de código que permitan automatizar la aplicación del *Slicer Pattern*. Ello reduciría los tiempos de desarrollo; y , por tanto, el coste. Además, al automatizarse el proceso se evita la introducción de errores debidos a la intervención humana. Esto contribuye a aumentar la calidad del producto final y a reducir los tiempos y costes de desarrollo; ya que el tiempo necesario para detectar y corregir estos potenciales errores desaparece.

Para alcanzar dicho objetivo, este proyecto fin de carrera ha desarrollado una serie de generadores de código que transforman modelos de diseño, en UML 2.0, de una línea de productos software en una implementación en C# basada en el *Slicer Pattern*. Dichos generadores de código se han implementado usando EGL (*Epsilon Generation Language*), el lenguaje de transformación modelo a código de la suite de herramientas para la manipulación de modelos *Epsilon*.

Palabras Clave

Línea de Productos Software, Generación de Código, Desarrollo Software Orientado a Características, Clases Parciales C#, Patrón Slicer, .NET, Epsilon, Te.NET, TENTE.

Preface

During the last years, a family of languages, tools and techniques for the development and configuration of software product lines for the .NET platform has been released in the Departamento de Matemáticas, Estadística y Computación of the Universidad de Cantabria. One of the most emergent techniques is what is known as the *Slicer Pattern*, which allows developers to achieve some degree of support for feature-oriented programming in C#. However, the manual instantiation of this pattern has associated a considerable effort, as it implies a large number of repetitive, laborious and error-prone tasks.

This capstone projects aims to overcome this limitation by means of the development of a set of code generators that automate the instantiation of the *Slicer Pattern*. This will reduce development efforts, and, therefore, development time and cost. Moreover, this automation avoids errors due to human intervention, which contributes to increase the product quality and to reduce development time and costs even more; since the time required to detect and correct these errors is saved.

Thus, this project has developed several code generators which transform design models, in UML 2.0, of a software product line into a C#-based implementation, based on the *Slicer Pattern*, of such software product line. These code generators have been implemented using EGL (*Epsilon Generation Language*), the model to text transformation language of *Epsilon* suite for model driven software development.

Keywords

Software Product Line, Code Generation, Feature-Oriented Software Development, C# Partial Classes, Slicer Pattern, .NET, Epsilon, Te.NET, TENTE.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este capítulo sirve de introducción a la presente Memoria de Proyecto Fin de Carrera. Para ello, en primer lugar se describe el contexto general donde se enmarca dicho proyecto y que da lugar al mismo. Se describe luego, a grandes rasgos, el proyecto para la metodología Te.Net, proyecto general de amplio alcance donde se inscribe el presente proyecto. A continuación, se exponen los objetivos principales del proyecto. Por último, se describe cómo se estructura el presente documento.

Índice

1.1. Contexto del Proyecto

Las *Líneas de Productos Software* **??** se han popularizado en la última década como un adecuado paradigma para el desarrollo, mantenimiento y gestión de proyectos software altamente similares. Un ejemplo de ello sería el producto Partenón[1] desarrollada por ISBAN (*Ingeniería del Software Bancario*) el cual proporciona un sistema software de gestión bancaria a las diferentes entidades del Grupo Santander en Europa y Estados Unidos. Partenón es un *corebanking* orientado al cliente que unifica el entorno operativo y de control, trasladando a cada entidad bancaria las prácticas que desean ser aplicadas de forma global a todas las entidades del Grupo Santander.

No obstante, Partenón en sí no es un producto operativo. Partenón necesita ser personalizado y adecuado a las necesidades particulares de cada entidad. Por ejemplo, dependiendo del país donde opere cada entidad, habrá que

[1] http://www.isban.es/partenon.html (Accedido el 17/06/2013)

adaptar partes de dicha aplicación para que cumpla con el marco legal de dicho país. De igual forma, dependiendo de si se trata de une entidad de banca on-line, de una entidad física o de una entidad de banca privada, se hará necesario instalar y desplegar una u otra serie de servicios.

Si este proceso se realizara siguiendo un enfoque tradicional, necesitaríamos desarrollar cada producto solicitado de forma específica e independiente del resto. A partir de un conjunto de elementos software reutilizables, procederíamos a crear nuevas versiones de dichos elementos, de forma que se adapten a las necesidades del producto concreto que deseamos crear. Por tanto, la personalización del producto requiere un tiempo y esfuerzo de desarrollo no despreciable, lo que constituye un cierto coste. Además, a medida que crecen los productos concretos creados a partir del núcleo Partenón, nos encontraríamos con más y más versiones, ligeramente diferentes entre ellas, de los mismos elementos software. Llegado un cierto punto, la gestión de estas múltiples versiones podría volverse un problema inabordable.

La finalidad última de una línea de productos software es evitar estos problemas mediante la aplicación de una idea basada en parte en las líneas de montaje industrial para la producción en masa. El objetivo de un línea de productos software es crear una infraestructura base común adecuada a partir de la cual se puedan derivar, tan automáticamente como sea posible, productos concretos pertenecientes a una familia de productos software, donde cada producto está personalizado de acuerdo a las necesidades de cada cliente particular concreto.

En relación con las líneas de productos software, y dentro del proyecto europeo *AMPLE (Aspect-oriented Model-Driven Product Line Engineering)* [2], el profesor Pablo Sánchez, actualmente adscrito a la Universidad de Cantabria, junto con otros investigadores, creó la metodología TENTE **??**. La principal novedad de dicha metodología era la de integrar diversos avances que se habían ido realizando en los paradigmas del desarrollo software orientado a características y el desarrollo software dirigido por modelos.

La *orientación a características* **?** tiene como objetivo encapsular porciones coherentes de la funcionalidad proporcionadas por una aplicación en módulos independientes llamados *características*. La orientación a características eleva el nivel al cual se agrupa la funcionalidad de un sistema del concepto de clase al concepto de *conjunto* o *familia de clases*, las cuales se añaden o eliminan de una aplicación como piezas o módulos indivisibles. De esta forma, las variaciones existentes en una familia de productos software se encapsulan en *características*, las cuales se añaden o eliminan de un producto concreto de acuerdo a las necesidades de cada cliente en particular.

El *desarrollo software dirigido por modelos*, entre otros objetivos, pretende automatizar partes del proceso de desarrollo de un producto software mediante la automatización de ciertos pasos del mismo, ya sea mediante

[2] www.ample-project.net

técnicas de análisis de modelos, simulación, transformación de un modelo en otro o generación de código desde los modelos. En el caso de las líneas de productos software, el desarrollo software dirigido por modelos se suele utilizar principalmente para automatizar el proceso de configuración o personalización de un producto concreto a partir de un conjunto común de elementos software reutilizables **?**.

Utilizando técnicas de desarrollo software dirigido por modelos, se puede generar el código necesario para la inclusión, exclusión configuración de las características una familia de productos software, a partir de modelos de alto nivel que especifican qué características deben estar presentes un determinado producto **?**.

Una vez definida la metodología TENTE, se decidió proceder a su transferencia tecnológica. Fue en este punto donde se encontró un obstáculo que en principio parecía insalvable. TENTE está basado en la utilización de un lenguaje orientados a características que soporte el concepto de familia de clases, al estilo de *CaesarJ* **?** u *ObjectTeams* **?**. Desafortundamente, la mayor parte de las empresas del sector de desarrollo software se mostraron bastante reticentes a a adoptar como lenguaje de programación un lenguaje de este estilo debido fundamentalmente a dos motivos: (1) el coste de aprendizaje por parte de sus programadores de los nuevos conceptos utilizados por este tipo de lenguajes; y, (2) la utilización de un nuevo lenguaje podría dejar obsoletas muchas de las herramientas de la empresa como las *suites* para ejecución de pruebas, al estilo de *JUnit* **?**.

Por tanto, se decidió que si se quería transferir la metodología TENTE a la industria software, debíamos rediseñar dicha metodología para que utilizase como lenguaje de programación un lenguaje orientado a objetos convencional, al estilo de Java o C#, en lugar de CaesarJ. Dada la preferencia de la mayoría de las empresas del sector industrial cántabro por la plataforma .Net **?**, se decidió elegir como lenguaje destino C#, el lenguaje de programación insignia de la plataforma .Net. La siguiente sección describe el estado actual de la metodología .Net.

1.2. La metodología Te.NET

Tal como se ha comentado en la sección anterior, la metodología Te.Net se trata de una variante de la tecnología TENTE. A diferencia de TENTE, la cual obliga a utilizar como lenguaje de programación final un lenguaje orientado a características que soporte el concepto de *familia de clases*, al estilo de *CaesarJ* **?** u *ObjectTeams* **?**, Te.NEt utiliza como lenguaje de programación destino un lenguaje convencional orientado a objetos, más concretamente C#.

El primer paso a realizar para llevar a cabo este rediseño de la metodología TENTE era analizar cómo se podía dar soporte a la orientación a

aspectos en un lenguaje de programación orientado a objetos como C#. Tras realizar una buscar opciones en el estado del arte actual, se encontró un prometedor trabajo **?** en el cual se proponía la utilización de las clases parciales de C# como mecanismos para dar soporte a la orientación características.

Por tanto, se decidió evaluar dicho trabajo en profundidad con objeto de verificar las ideas propuestas en el mismo. Los experimentos realizados **?** revelaron diferentes debilidades de las clases parciales como mecanismo para la implementación de líneas de productos software.

Para solventar los problemas detectados, se creó, como resultado de otro Proyecto Fin de Carrera presentado en esta misma Facultad, un patrón de diseño denominado *Slicer Pattern* **?**. Dentro de dicho Proyecto Fin de Carrera se implementó una línea de productos software para el desarrollo de software de gestión de hogares inteligentes.

Una vez que se había solventado el problema de cómo soportar la orientación a características en C#, la siguiente tarea a realizar era la de adaptar los generadores de códigos originales para que soportasen la generación de código en C# en lugar de CaesarJ. Esta tarea constituye el objetivo principal de este proyecto, el cual se detalla en la siguiente sección.

1.3. Motivación y Objetivos

Tal como se ha explicado en la sección anterior, la metodología Te.Net surge como solución para transferir la metodología TENTE a la industria software; en particular, para aquellas pequeñas y medianas empresas que no pueden permitirse, o son reticentes, la inversión necesaria para incorporar un nuevo lenguaje de programación a sus entornos de desarrollo. Dicha metodología está actualmente en desarrollo, siendo el siguiente paso a realizar la adaptación de los antiguos generadores de código de la metodología TENTE para que soporten los nuevos principios y lenguajes introducidos por la versión Te.Net.

El objetivo de este Proyecto de Fin de Carrera consiste precisamente en implementar dichos generadores de código. Dichos generadores de código deberán soportar dos fases diferentes del desarrollo de una línea de productos software. Por una parte, a partir de un modelo de diseño UML 2.0 orientado a características **??**, se deberán desarrollo generadores de código que transformen dicho modelo en una implementación en C#, utilizando como mecanismo para la gestión de la variabilidad el *Slicer Pattern*. Dicho código generado constituirá lo que se conoce como la implementación de referencia de una línea de productos software.

Por otra parte, se deberán crear generadores de código que permitan, a partir de una especificación de qué características deben estar presentes un producto concreto, generar el código necesario para personalizar la implementación de referencia anteriormente creada, de forma que se pueda

obtener un producto concreto adecuado a las necesidades particulares de un cliente concreto.

Por exigencias de los usuarios finales de este producto, el código generado deberá ser editable como un proyecto de Visual Studio 2010. Además, de acuerdo a principios generales de la metodología Te.Net, dichos generadores deberán ser implementados en el lenguaje EGL (*Epsilon Generation Language*) **?**, el lenguaje de transformación de modelo a código de la *suite* de desarrollo software dirigido por modelos Epsilon.

1.4. Estructura del Documento

Tras este capítulo introductorio, el resto del documento se estructura como sigue. El Capítulo 2 describe brevemente los conceptos necesarios para poder entender la presente memoria, y que no se pueden presuponer conocidos por el lector, tales como qué es una *Línea de Producto Software* o en qué consiste el *Slicer Pattern*. El Capítulo 3 explica el proceso de desarrollo de uno de los generadores de código creados, concretamente el que actúa durante la fase de la fase de *Ingeniería del Dominio* del desarrollo de una línea de productos software. El Capítulo 4 describe el desarrollo del generador de código que actúan durante la fase de configuración de una línea de productos software, la *Ingeniería de Aplicaciones*. Dicho capítulo también comenta brevemente las acciones realizadas para el despliegue de la aplicación. Por último, el Capítulo **??** sirve de sumario y cierre a esta memoria de Proyecto Fin de Carrera, proporcionando también las conclusiones extraídas tras su realización, así como posibles trabajos futuros.

Capítulo 2

Antecedentes y Planificación

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. A continuación, se describen diversos conceptos relacionados el dominio del proyecto, como son las líneas de productos software, el desarrollo software orientado a características, la metodología TENTE, el *Slicer Pattern*, y las técnicas de generación de código. Se concluye con la planificación que se ha llevado a cabo para la creación del Proyecto.

Índice

2.1. Caso de Estudio: Software para Hogares Inteligentes

Como caso de estudio a lo largo de este proyecto, se utilizará una línea de productos software para el desarrollo de software de gestión para hogares automatizados y/o inteligentes.

El objetivo final de estos hogares inteligentes 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 cerrar las ventanas abiertas cuando abandonan un hogar sus habitantes.

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 necesarios 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 dispositivo 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.

El software para la gestión de estos hogares inteligentes deberá ofrecer varios servicios, los cuales podrán ser opcionalmente incluidos en el software para un un hogar inteligentes 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.

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 producto Partenón, para software bancario, comentado en la introducción a este documento (ver Sección 1.1). Dicho producto representa una familia de productos destinados a la gestión bancaria. Partenón en sí no puede ser desplegado como una aplicación, sino que necesita ser configurado de acuerdo a una serie de características concretas demandadas por cada cliente que require una instalación de Partenón.

La idea de una línea de producto software es proporcionar una forma automática y sistemática de construir productos concretos dentro de una familia de producto software mediante la simple especificación de qué características deseamos incluir dentro de dicho producto. Esto representa una

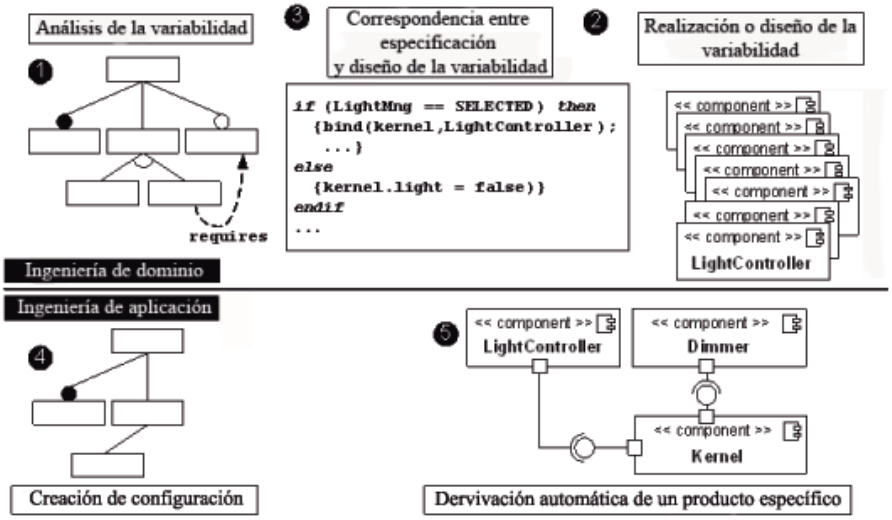

Figura 2.1: Proceso de Desarrollo de una línea de producto software

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 se estructura dos fases: (1) *Ingeniería del Dominio* (en inglés, *Domain Engineering*); y (2) *Ingeniería de Aplicación* (en inglés, *Application Engineering*) (ver Figura 2.1). La *Ingeniería del Dominio* tiene como objetivo la creación de la infraestructura o arquitectura de referencia de la línea de productos software. Esta arquitectura de referencia debe permitir la rápida, o incluso automática, construcción de sistemas software específicos pertenecientes a la familia de productos software. 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 qué son variables. Esta parte es la que se conoce como *Análisis o Especificación de la Variabilidad* (Figura 2.1, etiqueta 1).

A continuación, se ha de diseñar una arquitectura de referencia para la familia de producto software que permita soportar dicha variabilidad. Esta actividad se conoce como *Realización o Diseño de la Variabilidad* (Figura 2.1, etiqueta 2).

El siguiente paso es establecer una serie de reglas que especifiquen cómo hay que instanciar o configurar 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, etiqueta 3).

Tras completar la fase de Ingeniería del Dominio, disponemos de una especie de línea de montaje, la cual podemos utilizar para construir productos concretos de forma más o menos automatizada.

En la fase de Ingeniería de Aplicación, se crean productos concretos utilizando la infraestructura previamente creada. Para ello, el primer paso es crear una *configuración*, que no es más que una selección de características que un usuario desea incluir en su producto concereto (Figura 2.1, etiqueta 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 adapte automáticamente; generando un producto concreto específico acorde a las necesidades concretas del usuario (Figura 2.1, etiqueta 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 describe el paradigma de desarrollo software orientada a características, el cual está íntimamente ligado al diseño e implementación de líneas de productos software.

2.3. 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 coherente funcionalidad de un sistema software. Una características proporciona una opción de configuración potencial, ya que dicha característica debe poder ser incluida o excluida del producto software, dando lugar a productos software con diferentes funcionalidades. Por ejemplo, en el caso del software de gestión de hogares inteligentes, toda la funcionalidad relacionada con la gestión automática de luces, sería considerada como una característica.

La idea básica del diseño orientado a características es descomponer un sistema software en módulos bien definidos, donde cada módulo encapsula una característica que el sistema ofrece. El objetivo de la descomposición es la construcción de software bien estructurado que puede ser adaptado a las necesidades del usuario y el entorno, mediante la selección y composición de las características adecuadas.

Por tanto, a partir de un conjunto de características, se pueden generar multitud de sistemas software compartiendo características comunes y diferenciándose en otras, lo que hace que este paradigma sea especialmente adecuado para el diseño e implementación de *línea de productos software*.

Para ilustrar como funciona el paradigma orientado a características, utilizaremos un diseño UML (ver Figura 2.2) orientado a características ilustraremos de nuestro caso de estudio, el software de gestión de hogares

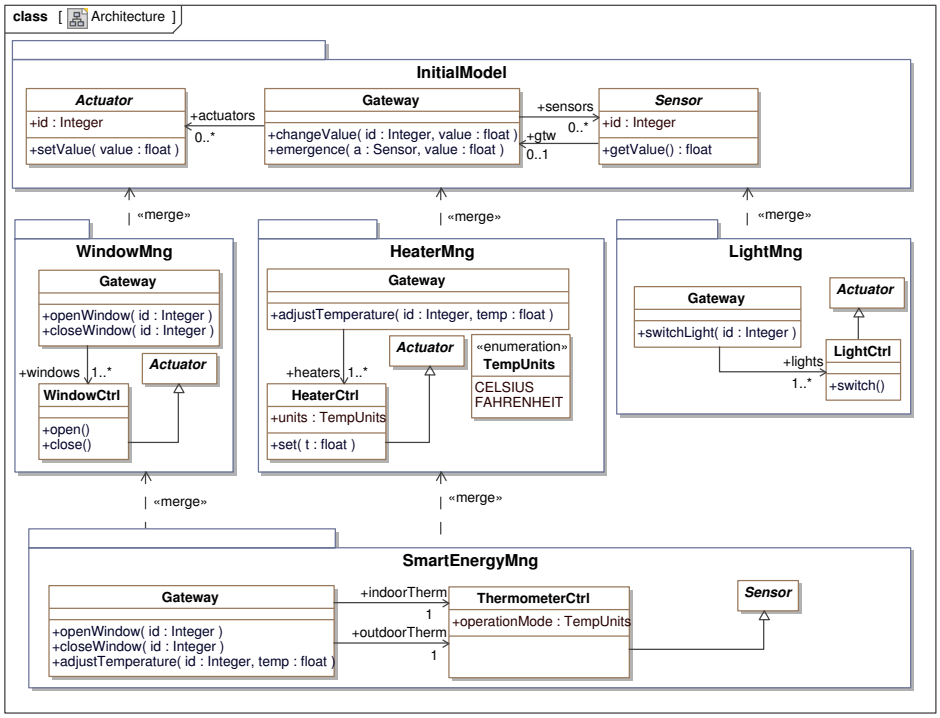

Figura 2.2: Diseño orientado a características del software para hogares inteligentes

inteligentes 2.1. Por razones de claridad, se ha simplificado dicho modelo de diseño, eliminando ciertas operaciones de la clase *Gateway* y suprimiendo todas las clases relacionadas con la interfaz gráfica de usuario.

Dos conceptos comúnmente utilizados por los lenguajes orientados a características son el concepto de *familia de clases* y de *clase virtual*.

Una *familia de clases* es un nuevo tipo de módulo que se utiliza para encapsular y gestionar como una unidad de composición un conjunto de clases pertenecientes a una misma característica. Por ejemplo, en nuestro caso de estudio, todas las clases que estén relacionadas con la gestión automáticas de luces, deberían estar encapsuladas en una misma familia de clases. Estas familias de clases se representan en el modelo UML de la Figura 2.2 como paquetes que contienen clases. Por ejemplo, el paquete *LightMng* representa la familia de clases que se correspondería con la característica *gestión de luces automáticas*.

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, al estilo de los métodos virtuales de una clase en el paradigma orientado a objetos (**?**).

La herencia entre familias de clases se representa en nuestro modelo UML

mediante relaciones *merge*, de acuerdo a la técnica expuesta por **?**. Cuando una familia de clases hereda de otra, hereda implícitamente todas sus clases virtuales. Todas las clases de una familia de clases son implícitamente clases virtuales.

Si una clase estuviese presente tanto en una familia de clases padre como en una familia de clases hija, como por ejemplo, Gateway en LightMng y BaseSystem en la Figura 2.2, el resultado que se produce es el mismo que se produciría si la clase virtual perteneciente a la familia de clases hija, heredase de la clase virtual perteneciente a la familia de clases padre. Por ejemplo, en nuestro caso, la aplicación funcionaría como si clase Gateway de LightMng heredase de la clase Gateway de BaseSystem.

De este modo, una clase virtual de una familia de clases hija, puede añadir nuevos atributos y métodos a las clases virtuales de las familias de clases padre. De igual forma, se pueden sobreescribir métodos de las virtuales de las familias de clases padre. Señalar que la herencia múltiple entre familias de clases sí suele estar permitida y soportada en los lenguajes de programación orientados a características.

Para implementar una línea de productos software, cada característica se tiende a manipular 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.

Por último, comentar que no todas las características pueden ser correctamente implementadas como familia de clases. Por ejemplo, en nuestro ejemplo, una característica del sistema es el modo de operación de los aparatos de frío/calor, los cuales pueden funcionar en grados Celsius o Fahrenheit. Este tipo de variabilidad se implementa mejor introduciendo un parámetro de configuración en las clases que corresponda, que creando familias de clases separadas para cada opción. No obstante, los lenguajes de programación orientados a características no impiden la utilización de los mecanismos tradicionales de gestión de la variabilidad de los lenguajes orientados a objetos, como la parametrización o los patrones de diseño (**?**).

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. Por ejemplo, si quisiéramos crear un producto concreto que tuviese las características LightMng y WindowMng, crearíamos una familia de clases, la cual estaría vacía, y la que le damos como nombre, por ejemplo, PatriciaHouse. A continuación, hacemos que PatriciaHouse herede de LightMng y WindowMng. De esta forma, ambas características se combinan en la familia de clases hija, la cual representaría un producto concreto con dichas características. Por tanto, el proceso de composición de características, se realiza mediante herencia entre familias de clases.

La siguiente sección proporciona una breve descripción sobre la metodología TENTE, una metodología orientada a características y dirigida por

modelos para el desarrollo y configuración de líneas de productos software.

2.4. 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 han de completar manualmente, obteniéndose al final un conjunto de módulos software, o piezas, cuya composición da lugar a productos software concretos[2]. Dichos módulos constituyen lo que se conoce como la implementación de referencia de la línea de productos software.

Para la derivación de productos concretos desde la infraestructura descrita en el párrafo anterior, TENTE usa un innovador lenguaje, denominado VML (*Variability Management Language*) **??** para la especificación de las reglas que indican cómo se ha de configurar una arquitectura de referencia de acuerdo con la selección de características de cada cliente en particular.

Posteriormente, a nivel de Ingeniería de Aplicaciones (ver Figura 2.1), se crea un modelo de configuración, el cual debe contener una lista con las características que el cliente desea incluir o excluir de su producto concreto. Utilizando este modelo de configuración como entrada, VML es capaz de ejecutar las reglas de derivación anteriormente especificadas para crear de forma automática el modelo de la arquitectónico del producto deseado por el cliente.

A continuación, 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 pertenecientes a la

[1] www.ample-project.net

[2] El nombre de la metodología proviene del célebre juego de construcción TENTE, versión española del popular Lego, el cual permite realizar diferentes construcciones mediante el ensamblado de una serie de piezas predefinidas

implementación de referencia 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 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.

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 pertenecientes al tejido industrial cántabro.

Tal como se ha comentado anteriormente, TENTE está diseñado, para utilizar como lenguaje de programación *CaesarJ*. No obstante, la mayoría de las empresas son bastante reticentes a cambiar su lenguaje habitual de programación, por la razones ya expuestas (ver Sección 1.1).

Fue por ello, junto con la preferencia de la mayoría de las empresas de desarrollo software cántabras por la plataforma .NET, por lo que se decidió desarrollar la metodología Te.Net (ver Sección 1.2).

El primero paso, tal como se ha comentado previamente, fue diseñar un mecanismo que permitiese simular las ventajas de los lenguajes orientados a características en C#. Dicho mecanismo, conocido como el *Slicer Pattern*, se describe en la siguiente sección.

2.5. Slicer Pattern

El mecanismo utilizado por la metodología Te.Net para gestionar la variabilidad de una línea de productos a nivel de código es el *Slicer Pattern* **?**. Dicho patrón se basa fuertemente en el concepto de clases parciales existente en C#. Por tanto, antes de proceder a la descripción de dicho patrón, introduciremos al lector en el concepto de clase parcial. Más concretamente, nos centraremos en el concepto de clase parcial proporcionado por C#.

2.5.1. Clases Parciales C#

Las clases parciales de C# **?** 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 **??**. La idea es que cada incremento en funcionalidad perteneciente a una característica se podría encapsular en una clase parcial separada. Cuando un cliente solicita un producto con una serie de características concretas, se combinarían (compilarían) las clases parciales correspondientes a esas características. Esto daría lugar a un producto que contendría única y exclusivamente las características seleccionadas.

Ilustramos esta idea con un ejemplo basado en el caso de estudio del presente proyecto, expuesto en la Figura 2.2. La Figura 2.3 muestra un ejemplo donde las clases parciales se aplican a la implementación de la clase Gateway. La implementación de esta clase para las caracterísiticas BaseSystem y LightMng ha sido separada en dos ficheros (Figura 2.3 líneas 0-9 y líneas 10-15) con el mismo nombre, pero emplazados en distintos directorios (BaseSystem/Gateway.cs y LightMng/Gateway.cs).

La clase Gateway para la característica BaseSystem (Figura 2.3 líneas 0-9) contiene colecciones para sensores y actuadores (líneas 2 y 3), al igual que los métodos changeValue, emergence (líneas 5-6), acorde al modelo de la Figura 2.2. La clase Gateway para la característica LightMng (Figura 2.3 líneas 10-15) contiene la colección ligthCtrl (línea 12) y el método switchLigth (línea 14).

La Figura 2.3 (líneas 16-28) muestra el fichero de construcción o compilación (SmartHome.csproj) para este proyecto. Ese fichero indica que las clases parciales para las características BaseSystem y LightMng están incluidas en la unidad de compilación; pero las correspondientes clases parciales para las características HeaterMng, WindowMng y SmartEnergyMng deben ser excluidas. Por tanto, el compilador generará una clase Gateway con la funcionalidad para controlar las luces pero no para controlar las ventanas o las temperaturas.

Inicialmente, las clases parciales de C# parecen un mecanismo adecuado para dar soporte a la orientación a características, al permitir dividir una clase en varias porciones, cada una de ellas correspondientes con una característica del producto a implementar.

Sin embargo, de acuerdo a una serie de experimentos realizados por **?** y **?**, las clases parciales poseen una serie inconvenientes que necesitan ser resueltos para que puedan ser utilizadas para implementar características. El principal problema es que utilizando clases parciales, no podemos ni sobreescribir ni extender métodos ya existentes dentro de una clase parcial. Ilustramos este problema con un ejemplo.

De acuerdo al modelo de la Figura 2.2 cuando se implementa la característica LightMng, debemos añadir una nueva colección, llamada LightCtrl, para almacenar objetos de tipo LightCtrl para la clase Gateway. Sin embargo, debemos también extender el contructor de la clase Gateway para inicializar apropiadamente dicha colección.

Siguiendo esta argumentación, se intenta escribir el código descrito en la

```
File BaseSystem/Gateway.cs
-----------------------------------------------------------
0 namespace SmartHome {
1    public partial class Gateway {
2       protected IList<Sensor> sensors;
3       protected IList<Actuator> actuators;
4
5       private void changeValue(Int id, float value) {...}
6       private void emergence(Sensor a, float value) {...}
7    }
8 }

File LightMng/Gateway.cs
-----------------------------------------------------------
10 namespace SmartHome {
11     public partial class Gateway {
12        private ISet<LigthCtrl> ligthCtrl;
13
14        private void switchLight(Int id) {...}
15 }

File SmartHome.csproj
-----------------------------------------------------------
16 </Project>
17 ...
18 <ItemGroup>
19 <Compile Include="BaseSystem\Gateway.cs" />
20 <Compile Include="LightMng\Gateway.cs" />
21 <!--
22 <Compile Include="HeaterMng\Gateway.cs" />
23 <Compile Include="WindowMng\Gateway.cs" />
24 <Compile Include="SmartEnergyMng\Gateway.cs" />
25 -->
26 ...
27 </ItemGroup>
28 </Project>
```

Figura 2.3: Implementación de la clase Gateway usando clases parciales

Figura 2.4. Dicho código contiene el constructor de la clase Gateway para la característica BaseSystem (Figura 2.4, líneas 3-6). Este constructor debería poder ser extendido en la clase parcial Gateway de la característica WindowMng tal como se muestra en la Figura 2.4, líneas 10-12.

Sin embargo, esto no es posible puesto que el compilador reporta un error indicando que el método Gateway() está duplicado y hay ambigüedad. Esto significa que no podemos separar la implementación de un método en varios archivos, ya que no se puede tener métodos con el mismo nombre en dos clases parciales distintas. Esto reduce las capacidades de extensión proporcionadas por las clases parciales a la adición de nuevos métodos y atributos,

```
File BaseSystem/Gateway.cs
-----------------------------------------------------
1 public partial class Gateway {
2 ...
3    public Gateway() {
4       this.floors = new List<Floor>();
5       this.interfaces = new List<CentralGUI>();
6    }
7 }
File LightMng/Gateway.cs
-----------------------------------------------------
8 public partial class Gateway {
9
10    protected IList<LightCtrl> lightCtrl;
11
12    public Gateway() {
13       this.lightCtrl = new List<LightCtrl>();
14    }
15 }
```

Figura 2.4: Implementación del constructor de la clase Gateway usando clases parciales

siendo imposible añadir funcionalidad o sobreescribir métodos existentes.

Por ejemplo, el caso de la característica SmartEnergyMng, la clase parcial Gateway debe sobreescribir el método adjustTemperature de la clase Gateway de la característica HeaterMng para comprobar si las ventanas deben cerrarse antes de cambiar la temperatura de los aparatos de frío calor. No obstante, como no podemos añadir un método adjustTemperature a la clase parcial Gateway de la característica SmartEnergyMng con el mismo nombre que el declarado en la característica HeaterMng, no hay forma de sobreescribir el método.

El *Slicer Pattern* surge como solución para resolver estas limitaciones. Dicho patrón se describe en la siguiente subsección.

2.5.2. *Slicer Pattern*

El *Slicer Pattern* se basa en la siguiente idea: dado que el problema es que no podíamos tener métodos con el mismo nombre en diferentes clases parciales, la solución consiste en añadir un prefijo a cada método, de forma que cada método tenga un nombre diferente. Dicho prefijo será el nombre de la características a la cual pertenece la clase parcial que contiene cada método.

La Figura 2.5 muestra un ejemplo de diseño para el hogar inteligente, el cual ha sido realizado siguiendo esta idea.

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

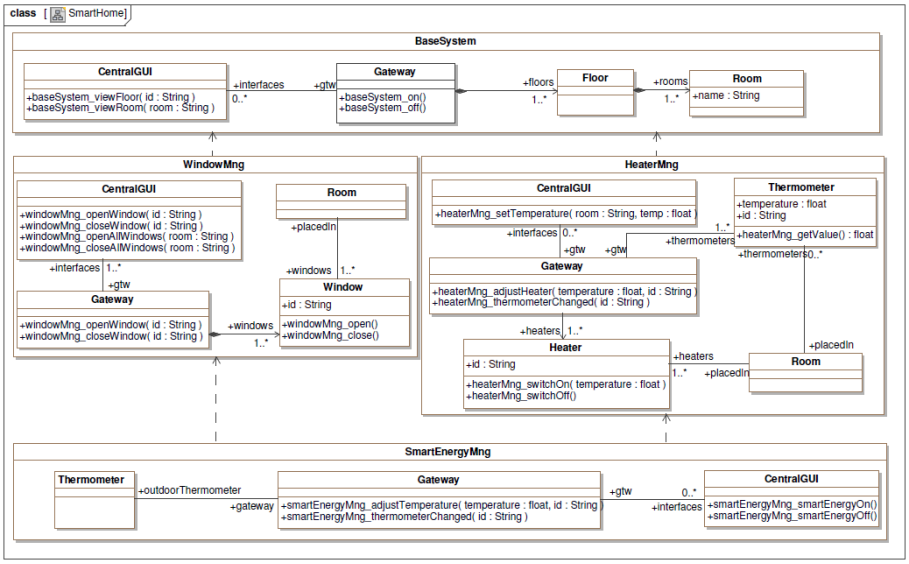

Figura 2.5: Proceso de Desarrollo de una Línea de Productos Software

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. A las versiones prefijadas de un método les denominaremos *versiones sucias* de dicho método.

Para generar un producto específico es necesario que, a nivel de Ingeniería de Aplicación, se compongan o combinen dichas versiones sucias. Para ello, cada vez que queramos configurar una nueva aplicación, debemos crear, por cada clase que deba ser incluida en el producto final, una nueva clase parcial, la cual se encargará de combinar o componer dichos métodos sucios de las clases parciales correspondientes a características. Dicha clase parcial contendría lo que denominaremos la *versión limpia* de los métodos a componer, es decir, las versiones de los métodos sin prefijar, que serían además públicos.

En nuestro caso, para configurar la clase *Gateway* con control inteligente de la energía, se crearía una nueva clase parcial *Gateway*. Dicho método contendría, por ejemplo, el método thermometerChanged (sin prefijo alguno).

A continuación, para componer los métodos, se hace que cada método limpio *delegue* en tantos métodos sucios como fuere necesario, de acuerdo a las características seleccionadas para el producto que se está construyendo. En nuestro caso, thermometerChanged delegaría en smartEnergy_thermometerChanged.

Para evitar que los métodos sucios heaterMng_thermometerChanged y smartEnergyMng_thermometerChanged puedan ser invocados directamente por objetos que no sean de la clase Gateway, todas las versiones sucias de un métodos

```
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 }
```

Figura 2.6: *Slicer Pattern* para constructores

serán privadas. De esta forma, sólo son invocables las versiones limpias de un método.

Esta idea, tal cual, no puede utilizarse para los constructores de la clases, ya que dichos constructores no se pueden ser renombrados, al tener que seguir un patrón específico para su nombre. Por tanto, necesitamos utilizar una solución alternativa.

Dicha solución se basa en hacer que la versión limpia de un constructor sólo se cree en el proceso de configuración de un producto software. En las clases parciales *sucias*, en lugar de constructores, tendremos métodos *especiales*, los cuales contendrán la lógica del constructor para dicha clase parcial. De esta forma, cada clase parcial X correspondiente a una característica F tendrá un método privado llamado <F>_init_<X> que contendrá el fragmento de la lógica del constructor para la clase X correspondiente a la característica F.

La Figura 2.6 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 (Figura 2.6 líneas 03-06). Se ha utilizado la misma técnica (Figura 2.6, líneas 10-12) para la característica WindowMng. Estos métodos *init* corresponderían a la *versión sucia* del constructor.

Para componer dichas *versiones sucias* de un constructor de acuerdo a una selección de características dada, se crea en la clase parcial que representa en producto a configurar el constructor de dicha clase. Dicho constructor constitute la *versión limpia* del constructor de la clase. Al igual que en el caso de los métodos regulares, dicho constructor delegará en tantos métodos *init* como sea necesario, de acuerdo a la selección de características realizada.

La Figura 2.6, líneas 16-20, muestra esta solución aplicada al constructor de la clase *Gateway*, entendiendo que sólo se ha seleccionado como característica a ser incluida en el producto final WindowMng.

Por tanto, a modo de resumen, se puede ver como utilizando el *Slicer Pattern* se pueden extender y sobreescribir métodos regulares y constructores, solventado las limitaciones inicialmente identificadas de las clases parciales en relación a la implementación de diseños orientados a características (**?**).

Por tanto, el objetivo de este proyecto es que todo el trabajo que es necesario realizar para instanciar este patrón, es decir, renombrado de los métodos para crear las versiones sucias, creación de las versiones limpias correspondientes, así como del código para delegar en los métodos que corresponda, sea generado automáticamente. Para ello es necesario crear una serie de generadores de código. La siguiente sección describe, a grandes rasgos, el funcionamiento de los lenguajes de generación de código.

2.6. Generación de Código con Epsilon

Epsilon **?** es una *suite* de lenguajes y herramientas para el desarrollo software dirigido por modelos. En este sentido, Epsilon ofreces lenguajes para la transformación de modelo a modelo, modelo a texto, navegación a través de modelos o generación de editores textuales y/o gráficos para modelos, entre otras características. Epsilon se distribuye actualmente como un plug-in para Eclipse. El presente proyecto se ha centrado en la generación de código, para lo cual se han utilizado los lenguajes *Epsilon Generation Language* (EGL) **?**, que es el lenguaje de generación de código proporcionado por Epsilon; y *Epsilon Object Language* (EOL) **?**, que es el lenguaje básico para la manipulación y navegación a través de modelos proporcionado por Epsilon. Describimos estos lenguajes en las siguiente subsecciones.

2.6.1. Epsilon Object Language(EOL)

El principal objetivo de EOL es proporcionar un lenguaje para la manipulación a nivel de código para la gestión de modelos. EOL proporciona

```
1 var bol = clase.hasParent();
2 bol.println();
3
4 operation Class hasParent (): Boolean{
5   var hasParent= false;
6   if (not self.generalization.isEmpty()){
7       hasParent=true;
8   }
9   return hasParent;
10 }
```

Figura 2.7: Ejemplo de operación EOL

facilidades para cargar un modelo, navegar a través de sus elementos o leer los valores de sus elementos, entre otras cracterísticas.

La Figura 2.7 muestra un ejemplo del funcionamiento de EOL. EOL no es un lenguaje orientado a objetos en el sentido de que no permite definir clases. Sin embargo, EOL permite gestionar objetos de tipos o clases externamente definidos, normalmente dentro del metamodelo o gramática que define un lenguaje de modelado.

En la Figura 2.7 línea 4, en este caso los objetos son de tipo Class. La línea 1 de la Figura 2.7 presenta un ejemplo de llamada a operaciones EOL, donde el elemento clase, de tipo Class, llama a la operación hasParent() y devuelve, tal como se aprecia en la línea 9, el valor de *hasParent* que será true o false dependiendo de si la clase a analizar presenta herencia, si es clase hija de otra clase, o no. Dicho contenido se imprime por pantalla tal como se aprecia en la línea 2 mediante la instrucción *println()*).

De esta forma, EOL permite definir funciones auxiliares para la manipulación o gestión de modelos. Estas funciones pueden ser utilizadas desde otros lenguajes de la suite Epsilon. En nuestro caso, estas funciones auxiliares se invocan desde el lenguaje de transformación modelo a texto EGL, el cual se describe a continuación.

2.6.2. Epsilon Generation Language(EGL)

EGL es un lenguaje de transformación modelo a texto (*model-to-text-transformation*). EGL puede ser tulizad para transformar modelos en diversos tipos de artefactos de carácter textual, incluyendo código (por ejemplo, Java), informes (por ejemplo, en HTML), imágenes (por ejemplo, gráficos SVG), especificaciones formales (por ejemplo, en el lenguaje Z), o incluso aplicaciones completas generadas con múltiples lenguajes (por ejemplo, HTML, Javascript, CSS, Java y SQL). En nuestro caso, utilizaremos EGL para la generación de código C# desde modelos UML.

EGL es un generador de código basado en plantillas. Dichas plantillas son similares a las utilizadas en la generación de páginas web dinámicas, al

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

Figura 2.8: Generación del nombre de cada Class contenida en un modelo de entrada

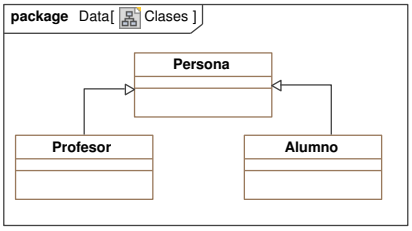

Figura 2.9: Ejemplo de modelo de entrada para generación de código con EGL

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

Figura 2.10: Resultado de la generación del nombre de cada Class contenida en un modelo de entrada

estilo de sistemas como JSP (*Java Server Pages*) **?** o PHP (*PHP Hypertext Preprocessor*) **?**.

Las Figuras 2.8, 2.9 y 2.10 muestran un ejemplo sencillo del funcionamiento de EGL. La Figura 2.8 contiene una sencilla plantilla que genera un listado de las clases que contiene un modelo UML. El texto situado dentro de los caracteres de escape `[%` y `%]` aparecer tal cual en la salida producida por la plantilla. El código situado entre dichos caracteres de escape es código EGL que gobierna el proceso de generación de código.

Por ejemplo, las líneas 1 y 3 declaran un bucle que recorre todas las clases contenidas en el modelo de entrada. En cada iteración, se procesa la línea 2, que genera automáticamente el texto fuera de los caracteres de escape ("El modelo contiene la clase: "). A continuación, se añade a la salida, por medio del operador =, el nombre de la clase *c*, que corresponda con la iteración realizada.

La Figura 2.10 muestra el resultado que produciría la plantilla de la Figura 2.8 para el modelo mostrado en la Figura 2.10.

Obviamente, EGL no sólo nos permite realizar estas sencillas operaciones. Por ejemplo, se pueden encapsular porciones reutilizables de código EGL en entidades bien definidos denominados *templates*. Un *template* tie-

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

Figura 2.11: Ejemplo de template en EGL

ne una funcionalidad similares a la de una *función* o *procedimiento* de un lenguaje de programación convencional.

La Figura 2.11 muestra un ejemplo de template que sirve para generar la cabecera de una clase Java. La cabecera del template (línea 2) especifica que el template de aplica a objetos del tipo *Class*, no precisando de ningún parámetro de entrada. La clase a la cual se aplica el template es un parámetro implícito de entrada a esta operación, a la cual se puede acceder a través del operador *self*. Utilizando este operador, la línea 4 genera la cabecera de una calse java, de acuerdo a la visibilidad *self.visibility* y nombre *self.name* de la clase sobre la cual se invoca esta operación. Siendo *c* un objeto de tipo *Class*, invocar este template sería tan fácil como escribir *c.getHeader()*.

Con este apartado, termina el proceso de formación y aprendizaje de los fundamentos, técnicas, lenguajes y herramientas necesarios para la realización del proyecto. Una vez adquirido este bagaje, se procedió a la definición de la planificación del proyecto.

2.7. Planificación

Como se ha comentado con anterioridad, el objetivo de este Proyecto Fin de Carrera era el desarrollar una serie de generadores de código que permitan automatizar la aplicación del *Slicer Pattern*. Este patrón se utiliza para la desarrollo de líneas de producto software, en las cuales se distinguen claramente, tal como se ha comentado dos fases: *Ingeniería del Dominio*, e *Ingeniería de Aplicaciones* (ver Figura 2.1).

Por tanto, el proceso de desarrollo del presente proyecto queda gobernado por dichas fases. La Figura 2.12 muestra dicho proceso de desarrollo.

Obviamente, la primera tarea (Figura 2.12, *T1A*), nada desdeñable, fue la de adquirir los conocimientos teóricos necesarios para la realización de todas las tareas posteriores. Ello implicaba adquirir los conocimientos relacionados con las *Líneas de Producto Software* (**??**) en general; y con el diseño orientado a modelos (**?**), las clases parciales (**?**) y el *Slicer Pattern* **?** en particular.

Dado que el proyecto se debía implementar con una herramienta para el desarrollo software dirigido por modelos, denominado *Epsilon* **?**, el siguiente paso (Figura 2.12, *T1B*) fue familiarizarse con dicha herramienta.

Para ello fue necesario adquirir ciertos conocimientos sobre EMF (*Eclip-*

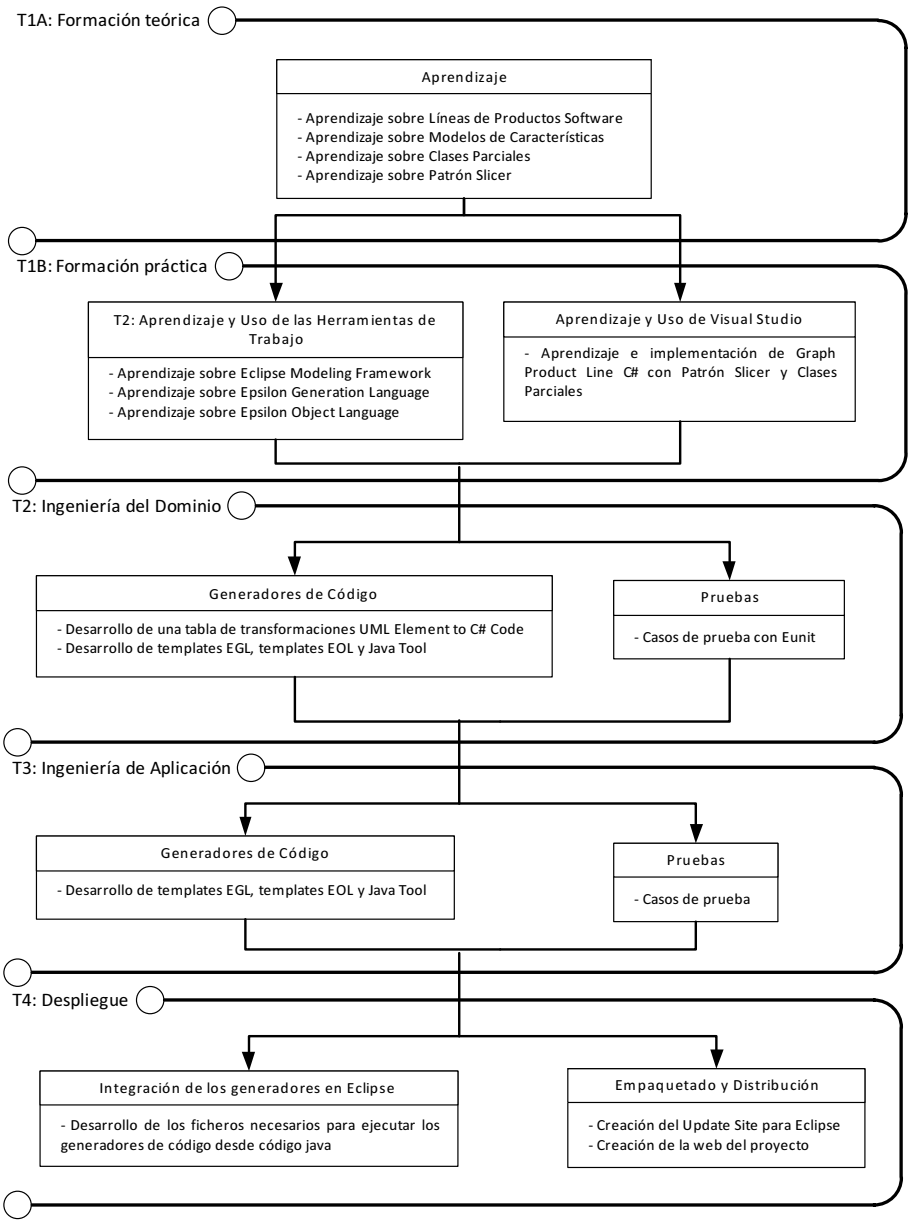

Figura 2.12: Proceso de desarrollo del Proyecto Fin de Carrera

se Modelling Framework) **?**, el lenguaje para la definición de lenguajes de modelado que constituye el corazón de Epsilon; así como con así como con EOL (*Epsilon Object Language*) (**?**), lenguaje que constituye la espina dorsal de Epsilon, y que es utilizado por la amplia mayoría de sus lenguajes y herramientas. A continuación, se debió adquirir destreza con el lenguajes a utilizar para la generación de código, EGL (*Epsilon Generation Language*) (**?**).

Además, por exigencias de los usuarios finales de este producto, el código generado debía ser editable como un proyecto de Visual Studio 2010, por lo que a continuación se procedió a familiarizarse con la utilización de dicha herramienta. Para ello, se implementó una línea de productos software sencilla utilizando las técnicas aprendidas hasta este punto, como el *Slicer Pattern*.

Tras esta tarea inicial de adquisición de conocimientos previos, el resto del proyecto se estructura de acuerdo a las dos fases de una línea de productos software.

La primera tarea tras la fase inicial de documentación fue la fase dedicada a la implementación de los generadores de código para la fase de *Ingeniería del Dominio* (Figura 2.12, *T2*). Dichos generadores de código necesarios debían, a partir de un modelo UML 2.0 orientado a características, generar un proyecto Visual Studio, en el lenguaje C# ,y que estuviera implementado acorde a las clases parciales C# y el *Slicer Pattern*. Para ello deberían generarse las clases parciales correspondientes a las cada características, así como los esqueletos las versiones sucias de los métodos. Dichos esqueletos, al igual que en la metodología original TENTE, deben ser completados a mano.

Para comprobar el correcto funcionamiento de estos generadores de código, se desarrollaron una serie de pruebas unitarias que se implementaron en *EUnit* (**?**), el lenguaje para la definición de pruebas unitarias de la suite *Epsilon*.

A continuación, la siguiente fase consistiría en el desarrollo de los generadores de código para la fase de *Ingeniería de Aplicaciones* (Figura 2.12, *T3*), para lo que se siguió un procedimiento similar al de la fase anterior. La función de estos generadores de código era la de crear las clases parciales destinadas a componer las versiones sucias de cada método, para lo cual se debían generar las versiones limpias de cada métodos, así como el código para que dichas versiones limpias delegasen en las versiones sucias que correspondiese.

Llegados a este punto, teníamos los generadores de código solicitados, por lo que sólo restaba proceder a su empaquetado y despliegue (Figura 2.12, *T4*). Este despliegue implicaba su integración dentro de la arquitectura de plugins de Eclipse.

Para ello primero se creó un plug-in para el entorno de desarrollo de Eclipse, de forma que éste tuviese elementos gráficos y la lógica necesaria para invocar a los generadores de código creados. A continuación, tanto los generadores de código cómo el plug-in creados debían empaquetarse y distribuirse de acuerdo a las políticas de desarrollo de plug-ins para Eclipse.

Tras dicha integración, se procedió a realizar una serie de pruebas de aceptación, destinadas a comprobar que el trabajo realizado satisfacía las necesidades de los usuarios finales que iban a utilizar el producto creado.

2.8. 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. A continuación, se ha especificado qué es una línea de productos software, el diseño orientado a características, la metodología TENTE, las limitaciones de las clases parciales en C#, cómo pueden resolverse estas limitaciones mediante el uso del *SlicerPattern* y el proceso de generación de código con Epsilon.

Capítulo 3

Ingeniería del Dominio

Este capítulo describe el proceso de desarrollo de los generadores de código para la fase de *Ingeniería del Dominio* dentro de la metodología Te.Net. Para ello primero describiremos las reglas que especifican cómo se transforman los elementos del modelo a código C#. A continuación, profundizaremos en el desarrollo e implementación de los generadores de código. Para ilustrar cómo funcionan los generadores de código, proporcionaremos un ejemplo sencillo. Concluiremos detallando cómo se han realizado las pruebas de los generadores de código creados.

Índice

3.1. Introducción

El primer paso a la hora de desarrollar un generador de código es establecer una serie de correspondencias entre los distintos tipos de elementos que pueden aparecer en los modelos que sirven como entrada y los elementos del lenguaje de programación destino. En nuestro caso, se trata de establecer una correspondencia entre elementos UML 2.0 y el lenguaje C#, teniendo en cuenta que los elementos de entrada como los de salida deben seguir un enfoque orientado a características. Para el caso concreto de la implementación, se debe hacer uso, de acuerdo a los objetivos iniciales del proyecto, del *Slicer Pattern*.

Una vez que las correspondencias estuviesen claramente definidas, el siguiente paso era implementar las plantillas de generación de código que debían establecer dichas correspondencias. El proceso de implementación de estas plantillas no es trivial. Ello se debe básicamente a que los proceso de generación de código, al generar texto, son secuenciales. Es decir, una vezx generado un elemento, no podemos volver atrás para modificarlo. Por ejemplo, si tras generar la cabecera de una clase descubriésemos que debemos añadir una relación de herencia a dicha clase, no podríamos realizar tal modificación. Por ello, cuando se implementan generadores de código, hay que prestar especial atención a su secuenciación.

Tras implementar los generadores de código, el siguiente paso es realizar las pruebas que correspondan para poder comprobar el correcto funcionamiento de los generados de código implementados. Para ello se debían diseñar un conjunto de pruebas de forma sistemática, y ejecutar dicho conjunto de pruebas para verificar el correcto funcionamiento de los generadores de código creados. La Sección 3.5 describe dicho proceso de diseño y ejecución de las pruebas.

Este capítulo describe este proceso de desarrollo. Más concretamente, la Sección 3.2 describe las correspondencias definidas ente elementos UML 2.0 y elementos de C#. La Sección 3.3 describe la secuenciación de las plantillas de generación de código, mientras que la Sección 3.4 muestra a modo de ejemplo, una sencilla plantilla de generación de código.

3.2. Transformaciones de Modelo UML a C#

Como hemos comentado, el primer paso para desarrollar una transformación de modelo a código es: (1) identificar los distintos casos o tipos de entrada con los que nos podemos encontrar; y (2) hallar un equivalente en el lenguaje destino (C# en nuestro caso). A continuación, mostramos los casos identificados (cómo título de cada subsección), y por cada caso, comentamos las equivalencias propuestas. Ciertas de estas reglas son específicas para líneas de productos software, mientras que otras, como la transformación de las asociaciones, son aplicables a cualquier transformación de UML 2.0 a C#. Cada regla de transformación la ilustramos utilizando el ejemplo de la Figura 2.2.

3.2.1. Modelo

Un modelo UML, es decir, el elemento raíz que contiene al resto de los elementos de un modelo UML, se transforma en el *namespace* para el proyecto C#. 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.

Recordar que para que varias clases parciales puedan ser combinadas, éstas deben pertenecer a un mismo *namespace*. Por tanto, se utiliza como nombre de dicho *namespace*, el nombre del modelo UML 2.0 que sirve de entrada a los generadores de código.

Además, al transformar el modelo UML 2.0, se crea un proyecto Visual Studio 2010, inicialmente vacío, con el mismo nombre que el modelo UML 2.0.

Para el caso de la Figura 2.2, el nombre del modelo, el cual no aparece en el diagrama, es *SmartHome*. Por tanto, se crearía un proyecto Visual Studio 2010, con *SmartHome* como nombre. Dentro de dicho proyecto, se crearía un *namespace* denominado *SmartHome*.

3.2.2. Paquete

Cada paquete UML 2.0 representa en nuestro caso una familia de clases, la cual encapsula una característica. Por tanto, por cada paquete UML 2.0, se crea una nueva carpeta o directorio, con el mismo nombre que el paquete, en el proyecto Visual Studio 2010 creado al transformar el modelo que contiene dicho paquete. En dicho directorio se colocarán todos los ficheros resultantes de transformar el contenido de dicho paquete.

Por ejemplo, para el caso de la Figura 2.2, durante la transformación del paquete WindowMng, se crearía una nueva carpeta dentro del proyecto Visual Studio 2010 generado, denominada WindowMng. Lo mismo se aplicaría al resto de los paquetes.

3.2.3. Tipos primitivos

Por cada tipo primitivo de UML 2.0, se establece una correspondencia con los tipos primitivos de C#. Por ejemplo, un *String* de UML 2.0 se transforma en *String* de C#. Un *boolean* de UML 2.0 se transforma en un *bool* de C#. Esta correspondencia es bastante directa y no presenta problemas más de allá de tener que renombrar algunos tipos.

3.2.4. Clases Enumeradas

Cada clase enumerada UML 2.0, se transforma en un enumerado de C#, con el mismo nombre que el enumerado UML 2.0. A continuación, se procesan los literales de la clase enumerado UML 2.0, añadiendo un literal con el mismo nombre al enumerado creado en C#.

Por ejemplo, la clase enumerada TempUnits de la característica HeaterMng se transformaría en un enumerado de C#, con nombre TempUnits, perteneciente al *namespace* HeaterMng, y con CELSIUS y FARENHEIT como literales. El Listado 3.1 muestra el código resultante de esta transformación.

```
01 namespace SmartHome{
02    enum TempUnits {
```

```
03         CELSIUS,
04         FARENHEIT
05     };
06 }
```

Listing 3.1: Código generado para la clase enumerada TempUnits

3.2.5. Clase

Por cada clase UML 2.0 encontrada dentro de un paquete, se genera una clase parcial sita en el directorio correspondiente al paquete al cual pertenece. El nombre de la clase parcial es el mismo que el de la clase UML 2.0.

Por ejemplo, para la clase *WindowCtrl*, del paquete *WindowMng*, se crearía una clase parcial pública, denominada *WindowCtrl*, y sita en la carpeta del proyecto *WindowMng*.

A continuación, se procesan los contenidos de dicha clase, tal como se describe a continuación.

3.2.6. Atributo

Cada atributo de una clase en UML 2.0 se transforma en una propiedad de C#, perteneciente a clase parcial correspondiente a la clase que posee el atributo en el modelo UML 2.0. Dicha propiedad tendrá siempre visibilidad *protegida* (*protected*), salvo que estuviese declarada como *privada* (*private*) en el modelo UML 2.0, en cuyo caso se mantendrá la visibilidad privada.

Si el atributo era público en el modelo UML, se le generarán métodos de acceso (*getter* y *setter*) a dicha propiedad. Si el atributo fuese de solo lectura o derivado, no se le generaría método de escritura (*setter*).

Si el atributo fuese estático (*static*), se generará como estático en el código C#, y no se le generarán métodos de acceso.

Si el tipo del atributo es un tipo primitivo y el atributo no es multivaluado, es decir, la cota superior de su multiplicidad es igual a 1, se utiliza como tipo su correspondiente en C#, de acuerdo las correspondencias comentadas en la Sección 3.2.3. Si el tipo fuese una clase u otro tipo no primitivo, el tipo será el nombre resultante de transformar dicho elemento no primitivo.

Por ejemplo, el atributo id de la clase Actuator, dentro de la característica BaseSystem, se transformaría en una propiedad llamada id, de la clase Actuator, sita en la carpeta BaseSystem, y perteneciente al *namespace* SmartHome. Como tipo para dicha propiedad, se utilizaría Int.

Para el caso del atributo units de la clase Actuator, dentro de la característica HeaterMng, se utilizaría como tipo TempUnits, ya que sería éste el nombre del enumerado resultante de transformar la clase enumerada que sirve como tipo de este atributo.

Si el atributo fuese un atributo un atributo multivaluado, es decir, la cota superior de su multiplicidad es superior a 1, el tipo de la propiedad generada

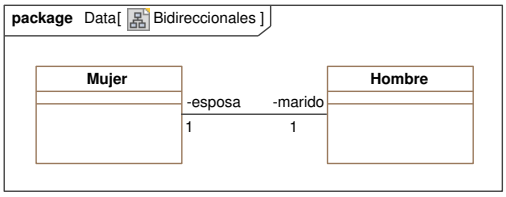

Figura 3.1: Ejemplo de asociación bidireccional

será una colección que use como tipo base el tipo del atributo. Dependiendo de ciertas propiedades del atributo isOrdered e isUnique, se deberá utilizar un tipo de colección u otro:

- Si el atributo tiene la propiedad isOrdered=false e isUnique=false estamos ante una colección que admite elementos repetidos y donde la posición es irrelevante. Se trata por tanto de una bolsa, que en C# se representan por medio de una ICollection.

- Si el atributo tiene la propiedad isOrdered=false e isUnique=true, se trata de un conjunto, ya que no hya repetidos y la posición es irrelevante. Escogemos por tanto el tipo de C#, ISet.

- Si el atributo tiene la propiedad isOrdered=true e isUnique=false se corresponde se trataría lista (IList), ya que son elementos en los que el orden es relevante y admite repeticiones.

- Si el atributo tiene la propiedad isOrdered=true e isUnique=true estamos ante un caso raro, poco utilizado dentro del mundo del desarrollo software, y para el que no se conocen equivalencias en lenguaje C#. Se trataría de una lista sin repeticiones. Se utiliza por tanto como colección una lista (IList) informando al usuario final, para que tome las medidas que considere adecuadas de cara al control de los duplicados.

3.2.7. Extremos de asociación

Las asociaciones en UML pueden ser de dos tipos:

Unidireccionales Sólo un extremo de la asociación aparece nombrado, y destacado con una flecha. Dicho extremo actúa como referencia entre clases. Por ejemplo, indoorTherm (Figura 2.2, paquete SmartEnergyMng) representa un referencia de la clase Gateway a la clase ThermometerCtrl. Dichas referencias pueden referirse a un solo objeto, como el caso de indoorTherm, o a una colección de ellos, como el caso ed actuators en la característica BaseSystem.

Bidireccionales Son los casos donde ambos extremos aparecen nombrados, pero no hay flechas en ninguno de los dos extremos. No se aprecian ejemplos de este tipo nuestro caso de estudio (Figura 2.2), por lo que se proporciona un ejemplo adicional (ver Figura **??**), el cual describimos a continuación. En este caso, un objeto de tipo Mujer posee una referencia marido a un objeto de tipo Hombre. A su vez, un objeto de tipo Hombre posee una referencia esposa a un objeto de tipo Mujer. Se espera que ambas referencias estén relacionadas. Es decir, si un objeto m de tipo Mujer tiene una referencia a un objeto h de tipo Hombre, de acuerdo la restricción de integridad impuesta por las asociaciones bidireccionales, el objeto h debe tener como valor para su referencia esposa el objeto m de tipo Mujer. Dicho de forma más fácil de entender, si h está casado con m, m debe de estar casado con h.

En cualquier caso, los extremos de asociación navegables en UML 2.0 representan referencias entre clases. Por tanto, un extremo de asociación tiene el mismo tratamiento que el de un atributo, siendo su tipo es una clase. De esta forma, cada extremo navegable de una asociación entre clases en UML 2.0 se transforma en una propiedad en C#, siguiendo un tratamiento similar al de los atributos. Al igual que en los atributos de las clases, si el extremo de asociación tiene multiplicidad superior igual a 1, se utiliza la clase referenciada como tipo de la propiedad. Si la multiplicidad fuese superior a 1, se utiliza una colección, siguiendo el mismo procedimiento que para los atributos, utilizando la clase referenciada como tipo base.

Además, para el caso de las asociaciones bidireccionales, se genera cierta lógica adicional, la cual está encargada de mantener la restricción de integridad impuesta por la bidireccionalidad.

Por ejemplo, sea un objeto m de tipo Mujer y un objeto h de tipo Hombre que no tienen asignadas sus respectivas referencias a marido y esposa, es decir, están *solteros*. supongamos ahora que asignamos al objeto m como valor para su atributo marido el objeto h, es decir, estamos casando a m con h. En este caso, debemos relacionar también al objeto h con el objeto m, es decir asignar el objeto m como valor para el atributo esposa del objeto h, para así cumplir con la restricción de integridad impuesta por la asociación bidireccional. La lógica generada en nuestro caso, se encarga de asegurar que se produce esta segunda asignación, y si m se casa con h, entonces h también se casa con m.

Si ahora decidiésemos cambiar la referencia marido del objeto m, y asignarle como valor otro objeto de tipo Hombre, sea h1, deberíamos: (1) poner la referencia esposa de h a null, es decir, divorciar a h de m, permitiendo que m se pueda volver a casar; (2) Asignar al objeto m como valor para la referencia marido el objeto h1, es decir, casar a m con h1; y, (3) asignar como valor para la referencia esposa de h1 el objeto m, de forma que m y h1 queden efectivamente casados. Resaltar que si h1 estuviese casado, habría que divorciarlo primero, tal como hemos hecho con h.

Toda esta lógica necesaria para mantener estas restricciones de integridad, tanto en el caso en el cual los extremos de asociación son de multiplicidad superior unitaria o multivaluada, se genera de forma automática por nuestra aplicación.

3.2.8. Operación

Cada operación de una clase en UML 2.0 se transforma en un método de C#, perteneciente a clase parcial correspondiente a la clase que posee la operación en el modelo UML 2.0. De acuerdo con los principios del *Slicer Pattern*, el nombre de dicho método será el nombre de la operación UML, prefijado con el nombre del paquete donde se encuentra contenida la clase que posee dicha operación. Además, dicho método tendrá siempre la visibilidad privada (*private*), de acuerdo a las exigencias del *Slicer Pattern*.

Por ejemplo, el método openWindow de la clase Gateway perteneciente a la característica SmartEnergyMng se renombraría como SmartEnergyMng_openWindow en la correspondiente clase parcial Gateway dentro de la carpeta del proyecto SmartEnergyMng.

Los parámetros de cada operación se transforman tal como se describe en la siguiente subsección.

3.2.9. Parámetro

Los parámetros de una operación se procesan de forma similar a los atributos. Se procesan de forma separada el parámetro de retorno, si lo hubiere, y los parámetros de entrada. Si el parámetro de retorno está presente, se obvia su nombre, y se añade su tipo como tipo de retorno de la operación a la cual pertenece. Si no hubiera parámetro de retorno especificado, se añade void como tipo de retorno. Por cada parámetro, se añade un nuevo parámetro a la operación.

Por ejemplo, para la operación openWindow, de la clase Gateway, en la característica WindowMng, se definiría void como tipo de retorno, ya que no dispone de un parámetro de retorno asociado. El parámetro de parámetro de entrada id, con tipo Integer, se añadiría al método SmartEnergyMng_openWindow.

3.2.10. Constructor

Para cada clase parcial generada , se genera un método privado tipo *init*, destinado a encapsular la lógica del constructor de dicha clase asociada a la característica que representa, de acuerdo a las directrices del *Slicer Pattern*.

Por ejemplo, para la clase parcial Gateway de la característica HeaterMng, se genera un método denominado HeaterMng_initGateway. De manera análoga, para la clase parcial Gateway de la característica WindowMng, se genera un método denominado WindowMng_initGateway, y así sucesivamente. Cada

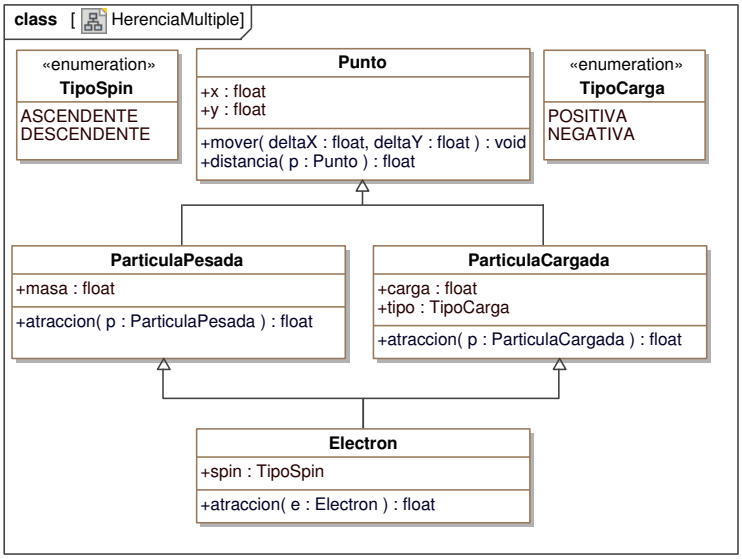

Figura 3.2: Ejemplo de herencia múltiple

método contiene la porción de la lógica del constructor que corresponde a la característica a la cual pertenece dicha clase parcial.

3.2.11. Interfaz

Las interfaces se procesan como si fuesen clases, salvo porque generan interfaces de C#, en lugar de clases.

3.2.12. Generalización

Las relaciones de generalización (o herencia) entre clases de UML 2.0 se transforman en relaciones de herencia entre clases de C#. No obstante, señalar que en UML 2.0 está permitida la herencia múltiple, mientras que en C# no lo está. Para soportar la herencia múltiple en C#, hacemos uso del *mixin pattern*, el cual describimos a continuación.

Herencia múltiple con el *mixin pattern*

El *mixin pattern* tiene como objetivo simular el funcionamiento de la herencia múltiple en lenguajes que carecen de tal. La Figura 3.2 muestra un ejemplo de herencia múltiple, donde una clase Electron hereda a la vez de las clases ParticulaPesada y ParticulaCargada, ya que un electrón es tanto una partícula cargada como una partícula pesada.

Los lenguajes de programación orientados a objetos modernos, como Java o C#, no soportan la herencia múltiple entre clases. No obstante, si

permiten que una clase herede de tantas interfaces como se desee. Por ello, la solución propuesta por el *mixin pattern* es como sigue:

1. Por cada una de las clases padre de las que se debe heredar, se crea una interfaz que contiene todos los métodos públicos de dicha clase. En nuestro caso, crearíamos interfaces para ParticulaPesada y ParticulaCargada, tal como se muestra e en la Figura 3.3.

2. A continuación, hacemos que cada clase padre original, herede de la correspondiente interfaz recientemente creada. Con ello conseguimos que la interfaz (e.g., IParticulaPesada) defina un *tipo de datos* reutilizable, mientras que la clase (e.g., ParticulaPesada) proporciona una implementación para dicho tipo.

3. La clase hija hereda ahora de las interfaces en lugar de las clases padre. Esta herencia múltiple si está permitida, y gracias a ella, la clase hija es del tipo de las interfaces heredadas. En nuestro caso, Electron es también de los tipos IParticulaCargada e IParticulaPesada (ver Figura 3.3).

4. Para reutilizar la implementación de dichos métodos proporcionada por las clases padre, se añade una referencia, denominada *mixin*, de la clase hija a cada cada clase padre. en nuestro caso, se añaden las referencias particulaPesadaMixed y particulaCargadaMixed, de la clase Electron a las clases particulaCargadaMixed.

5. A continuación, para cada método que la clase hija deba implementar como consecuencia de heredar de las interfaces que representan las clase padre, se hace que dicho método delegue en el método correspondiente de la clase padre. De esta forma, reutilizamos la implementación del método por *composición y delegación*.

Cada vez que nuestro generador de código debe procesar una clase con herencia múltiple, se ejecutan todos los pasos previamente descritos, de forma que se automatiza la aplicación del *mixin pattern*, lo que ahorra una gran cantidad de trabajo.

Una vez que teníamos claramente definidas las reglas para la transformación de los elementos UML 2.0 a código C#, el siguiente paso fue el de proceder a su implementación en EGL. La siguiente sección describe de forma superficial dicho proceso.

3.3. Generadores de Código C#

Para implementar los generadores de código, se procedió en encapsular cada una de las reglas descritas en la sección anterior en un *template* de EGL. Además, se crearon una serie de funciones auxiliares en EOL. Por

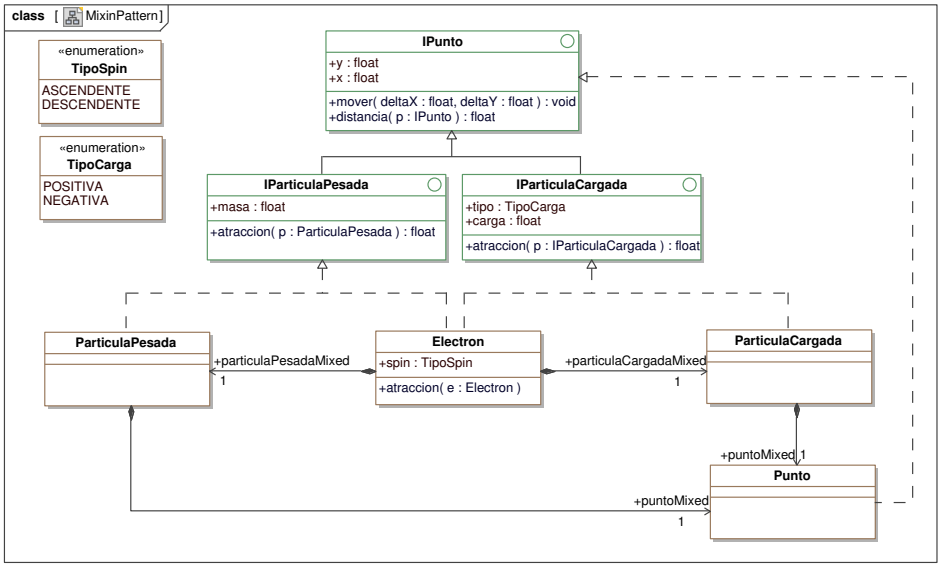

Figura 3.3: Herencia múltiple mediante el *mixin pattern*

ejemplo, se creó una función auxiliar para determinar el tipo de colección que debe ser utilizada para transformar un atributo multivaluado, es decir, con cota superior de su multiplicidad mayor que uno.

Uno de los mayores problemas que normalmente plantean los generadores de código es que la generación de código es secuencial, no permitiendo la vuelta a atrás. Por ejemplo, si generamos una clase y más tarde descubrimos que dicha clase debe ser modificada porque actúa como clase padre en una herencia múltiple, ya no podremos volver a abrir dicha clase para añadirle la relación de herencia con la interfaz que ha de crearse.

Por tanto, antes de generar una clase, debemos asegurarnos de que no va a necesitar ser modificada posteriormente. Ello implica que hay que tener especial cuidado a la hora de diseñar el orden en el cual se ejecutan las plantillas, o *templates* de generación de código. La Figura 3.4 muestra el orden de ejecución de las plantillas creadas en nuestro caso. Explicamos parte de dicha figura, aunque no la describiremos entera, por razones de espacio.

El punto de partida es el generador de código llamado ProjectCreation, encargado de procesar el elemento *modelo*, que constituye la raíz del proyecto, así como los *paquetes* que contiene dicho modelo, además de crear el proyecto *Visual Studio 2010* que constituye la salida del generador. Dicho *template* tiene , por tanto, dos tareas claramente diferenciadas: (1) por una parte, debe generar el código correspondiente a la arquitectura de referencia, lo que se hace a través de la plantilla ClassFilesCreation; y (2) por otra parte, debe generar todos los ficheros auxiliares y la estructura que constituyen un

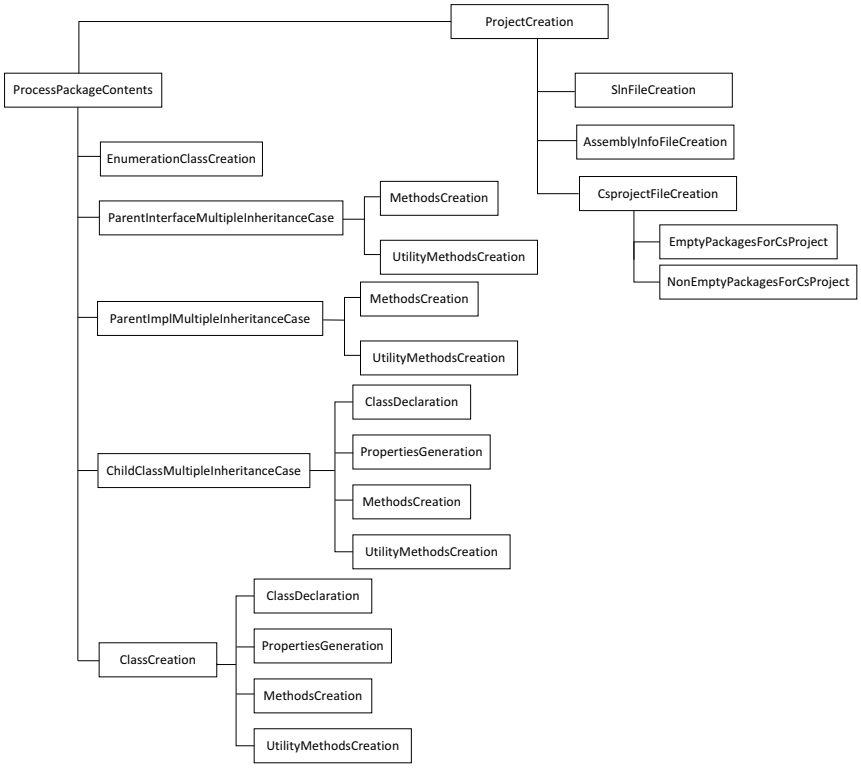

Figura 3.4: Orden de ejecución de las plantillas de generación de código

proyecto *Visual Studio 2010*, como el fichero de construcción (fichero *.csproj*) que indica que clases parciales deben compilarse cuando se construye el proyecto (ver Figura 2.3). Para generar estos ficheros auxiliares, se utilizan las plantillas SlnFileCreation, CsprojectFileCreation y AssemblyInfoFileCreation.

La plantilla ProcessPackageContents procesa por cada paquete, su contenido. Dependiendo del tipo de cada elemento, se realiza una acción diferente, tal como se describe a continuación.

Si se trata de una clase enumerada, se invoca el template EnumerationClassCreation, con dicho elemento como argumento.

Se procesan todas las clases con herencia múltiple, para aplicar el *mixin pattern*. Para ello se ejecutan las plantillas ParentImplMultipleInheritanceCase, que se encarga de procesar las clases padre involucradas en herencias múltiples; y ParentInterfaceMultipleInheritanceCase, que se encarga de crear las interfaces para estas clases padre. Ambas plantillas hacen uso de las plantillas MethodsCreation y UtilityMethodsCreation, encargadas de procesar los métodos de dichas clases e interfaces y de crear los métodos de infraestructura que fuesen necesarios, tal como Equals o CompareTo.

A continuación, se ejecuta la plantilla ChildClassMultipleInheritance, encargada de procesar una clase hija involucrada en herencia múltiple. Para ello

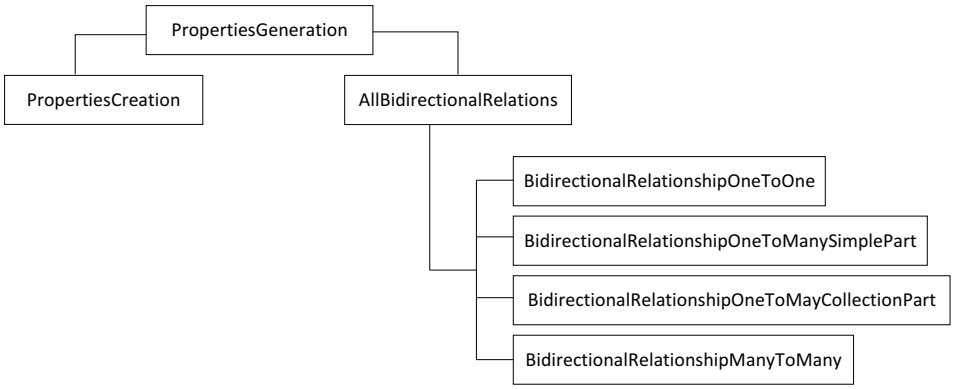

Figura 3.5: Orden de ejecución en la plantilla de generación de código PropertiesGeneration

se procesan el esqueleto de la clase (ClassDeclaration), sus atributos (PropertiesGeneration), sus métodos (MethodsCreation) y sus métodos de infraestructura (UtilityMethodsCreation).

Seguidamente, se procesan las clases no afectadas, como hijas o como padres, por herencia múltiple. Estas clases se procesan a través de la plantilla ClassCreation, que funciona igual que la plantilla ChildClassMultipleInheritance, a excepción de que no se genera el código de los delegados para los *mixins*.

Cada plantilla invocada hace uso a su vez de otras subplantillas, que por razones de claridad y espacio no detallamos. Por ejemplo, la plantilla PropertiesGeneration encargada de procesar atributos y extremos de asociación, hace uso de diversas plantillas para procesar los extremos pertenecientes a asociaciones doblemente navegables, tal como se indica en la Figura 3.5.

Por último, comentar que todas las plantillas utilizan diversas funciones auxiliares especificadas en EOL. Por ejemplo, existen funciones para determinar si una clase está involucrada en una herencia múltiple o para devolver todas las clases padre de una clase dada. Además, junto con las funciones auxiliares de EOL, se han creado una serie de funciones auxiliares en Java, invocables desde EOL, y EGL, lo que se conoce en argot Epsilon como una *Java Tool*, para poder manipular el sistema de ficheros. Esto era necesario, por ejemplo, para poder crear la estructura de carpetas del proyecto Visual Studio 2010.

Además, se crearon algunas *Java Tool* para poder mostrar cuadros de diálogo que permitiesen interactuar con el usuario durante el proceso de generación de código, ya fuese para mostrarle o requerirle información.

Las Figuras 3.6 y 3.7 muestra dos ejemplos de estos diálogos. El primero de ellos aparecería al utilizar como entrada para el generador de código un archivo UML que tuviese varios modelos. En este caso, el proceso de generación de código preguntaría al usuario cuál es el modelo a procesar, ya

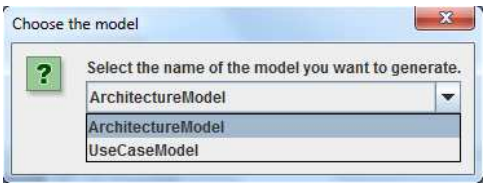

Figura 3.6: Selección de modelo en un proyecto con varios modelos

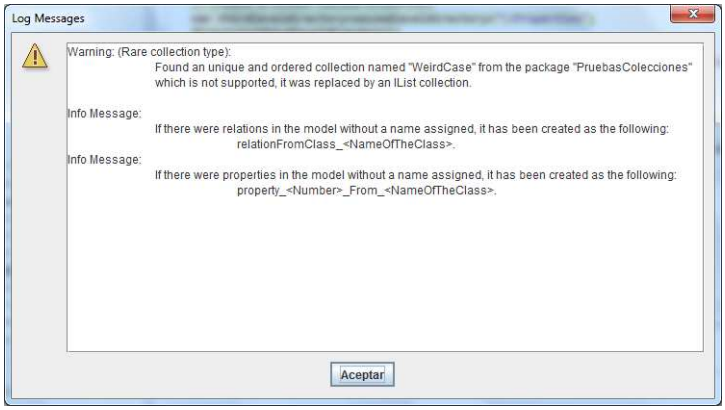

Figura 3.7: Log mostrado al finalizar el proceso

dentro de un fichero UML pueden coexistir modelos de diversa índole.

El segundo diálogo (Figura 3.7) muestra, al final del proceso de generación de código, las incidencias que hayan podido producirse durante dicho proceso.

La siguiente sección muestra, para el lector interesado, una plantilla de las descritas en esta sección a nivel de código.

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

Esta sección muestra, a modo de ejemplo, una de las plantillas de generación de código creadas en este Proyecto Fin de Carrera, así como un ejemplo de cómo se puede utilizar Java desde Epsilon.

3.4.1. Plantilla de generación de código

Con objeto de no distraer al lector con detalles irrelevantes, hemos escogido una de las plantillas más sencillas, no trivial, de entre las creadas. Dicha plantilla se muestra en el Listado 3.2. El objetivo de dicha plantilla es generar el bloque de código correspondiente a los esqueletos de los métodos contenidos para una clase C#.

```
00 [%
01 import "ReturnParameterCreation.egl";
02 import "ParametersCreation.egl";
03 import "../Operations.eol";
04 operation Classifier classMethods(package:String,path:String)
                    : String {
05
06  opers=self.generatePartialConstructor(package);
07
08  for (oper in self.getOperations()){
09      if (oper.type==null){
10          operations_return.add(oper);
11      }else{
12          operations_void.add(oper);
13      }//if
14  }//for-operations
15  for (oper in operations_void) {
16   name=oper.methodName();
17   opers=opers+oper.voidOperation(package,name,path);
18  }
19  for (oper in operations_return) {
20   name=oper.methodName();
21   opers=opers+oper.returnOperation(package,name,path);
22  }
23  return opers;
24 }%]
```

Listing 3.2: Implementación del generador de código MethodsCreation

La plantilla contiene una primera sección (líneas 1-3) donde se importan elementos,como funciones EOL, definidas en otros fichero. A continuación, se declara la cabecera de la plantilla, la cual, recordemos, puede ser invocada como una función. Esta función retorna el bloque de texto a generar, como un String, y, en principio, es aplicable a cualquier Classifier, de acuerdo con el sistema de tipos de UML 2.0. La función acepta dos parámetros que son necesarios para calcular ciertos elementos necesarios para la generación de código: (1) el nombre del paquete o característica donde se encuentra la clase que estamos procesando, necesario para poder prefijar el nombre de los métodos; y (2) la ruta donde se encuentra el fichero correspondiente a la clase parcial que estamos procesando.

Dicha función, al ejecutarse, genera en primer lugar el constructor para dicha clase, mediante la invocación a una subplantilla encargada de dicha tarea (línea 06). A continuación, se procesan todas las operaciones contenidas en el elemento UML que estamos procesando (líneas 8-22). Este elemento (Classifier) es el objeto que recibe la llamada, accesible a través de la palabra clave predefinida self.

Para procesar las operaciones, iteramos sobre ellas (línea 08). Para cada operación, comprobamos si dicha operación contiene o no parámetro de retorno (líneas 09-13). Aquellas operaciones que tienen parámetro de retorno definido (líneas 09-11) se añaden a la colección operations_return. Aquellas ope-

raciones que tienen parámetro de retorno definido (líneas 12-13) se añaden a la colección operations_void. Para estas últimas, se utiliza void como tipo de retorno. Una solución inteligente podría haber sido añadir void como tipo de retorno a las operaciones que no tuviesen tipo de retorno, pero, recordemos, no podemos modificar el modelo de entrada a los generadores de código.

Por último, se procesan ambas colecciones, operations_return y operations_void, invocando para ello las correspondientes subplantillas de generación de código (returnOperation y voidOperation), respectivamente (líneas 15-22). Finalmente, se retorna el texto generado, el cual se guardará en el fichero pertinente (línea 23).

3.4.2. Invocar código Java desde Epsilon

Para utilizar una *Java Tool* en Epsilon es necesario *envolver* las funciones Java en funciones EOL. El Listado 3.3 muestra un ejemplo de dicho proceso de envoltura.

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

Listing 3.3: Uso de un Java Tool en los generadores de código

Una vez explicado este sencillo ejemplo, damos por concluida la etapa de implementación de los generadores de código para la parte de *Ingeniería del Dominio*, el siguiente paso es la realización de las pruebas pertinentes para comprobar el correcto funcionamiento de los mismos, la siguiente sección 3.5 muestra cómo se desarrolla dicha etapa.

3.5. Pruebas

Una vez implementados los generadores de código, la siguiente tarea era comprobar su correcto funcionamiento. Para ello creamos, de forma sistemática, una serie de pruebas unitarias que permitiesen comprobar el correcto funcionamiento de los generadores de código para un exhaustivo conjunto de diferentes tipos de entrada. Estas pruebas unitarias se implementaron en *EUnit* **?**, el lenguaje de definición de pruebas de la suite Epsilon. El funcionamiento de *EUnit* es similar al de *JUnit*, pero aplicado a los lenguajes de la suite Epsilon, como EGL.

Para comprobar que el funcionamiento del generador de código es correcto, se diseña el caso de prueba y se crea la salida de ese caso de prueba de forma manual. A continuación, se ejecuta el caso de prueba y se comprueba que la salida generada coincide con la esperada, que es la creada manualmente. Al intentar implementar estas pruebas en *EUnit*, nos encontramos con el

	Casos válidos	Casos no válidos
Paquete	Paquete con nombre. Paquete con clases e interfaces en su interior. Paquete vacío. Paquete dentro de otro paquete (recursividad).	Paquete sin nombre.
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.

Cuadro 3.1: Casos de prueba para lo generadores de Ingeniería del Dominio

problema inicial de que este lenguaje no tenía implementada la comparación de fragmentos de texto en ficheros, que era precisamente la funcionalidad que nos hacía falta. No obstante, se cursó una petición a los creadores de Epsilon, los cuales, muy amablemente, incorporaron dicha funcionalidad a *EUnit*. De igual forma, añadieron otra funcionalidad para comprobar que dos directorios contenían los mismos archivos (assertEqualDirectories).

Para el diseño de los casos de prueba se siguió inicialmente un enfoque de caja negra, basado en una adaptación de la técnica de clases de equivalencia y análisis de valores límite al entorno de los modelos software. Una vez ejecutadas estas pruebas, se analizó la cobertura alcanzada, definiendo pruebas adicionales, ya de caja blanca, de forma que la cobertura alcanzada fuese del 100 %.

La Tabla 3.1 resume algunos de los casos de prueba ejecutados. Concretamente, se muestran los casos de prueba para analizar el comportamiento de las plantillas de generación de código para paquetes y clases.

```
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

3.6. Sumario

Durante este capítulo se ha descrito el proceso de desarrollo de los generadores de código para la fase de *Ingeniería del Dominio* de la metodología Te.Net. Para ello, primero se ha analizado cómo se transforman los elementos de un modelo UML 2.0 orientado a características en código C#. A continuación, se ha descrito de forma superficial la implementación de los generadores de código, mostrando el funcionamiento de una sencilla plantilla. Por último, se han descrito las pruebas realizadas.

Capítulo 4

Ingeniería de Aplicación

El capítulo anterior describió el funcionamiento y desarrollo de los generadores de código para la fase de *Ingeniería del Dominio*. Dichos generadores son los encargados de crear los esqueletos de la implementación de referencia, la cual debe completarse manualmente. Este capítulo describe el funcionamiento y desarrollo de los generadores de código para la fase de *Ingeniería de Aplicaciones*, los cuales tienen como objetivo adaptar la implementación de referencia a las necesidades de cada cliente.

Índice

4.1. Introducción

El primer paso para crear un producto concreto, de acuerdo con la metodología Te.Net (ver Sección 1.2) es crear una selección de aquellas características que se desea incluir en el producto. Cómo se crea dicha selección de características está fuera del ámbito de este proyecto fin de carrera. Referimos al lector interesado en tal asunto a otros proyectos fin de carrera presentados en esta misma Facultad sobre dicho tema ().

Una vez que se tiene una selección de características válida, utilizando dicha selección de características, se configura la arquitectura de referencia creada en la fase de Ingeniería del Dominio para crear un modelo arquitectónico concreto, adaptado a las necesidades del cliente, del producto que

queremos construir. Dicho modelo arquitectónico se obtiene de forma automática mediante la utilización del lenguaje *VML* (), de acuerdo con la metodología Te.Net (ver Sección 1.2).

4.2. Configuración de productos a nivel arquitectónico

La estrategia para crear un modelo arquitectónico concreto consiste, tal como se describió en la Sección 2.3, en crear un paquete vacío, el cual representa el producto a construir, y añadir relaciones *merge* a aquellas características que se desean incluir en el producto final. De esta forma, el contenido de los paquetes correspondiente a características que se deben incluir en el producto final, se combinan o componen en el paquete que representa el producto final.

Para distinguir el paquete que representa el producto final de los paquetes que representan características, se ha creado un perfil de UML 2.0 (**?**). Un perfil UML 2.0 es un mecanismo genérico de extensión que permite personalizar los modelos UML para un propósito particular, mediante la especificación de estereotipos y valores etiquetados que modifican la semántica original de los elementos del modelo UML 2.0.

En nuestro caso, el perfil contiene un solo estereotipo, denominado SpecificProduct, el cual se puede aplicar exclusivamente a paquetes UML tal como se puede apreciar en la Figura4.1. Además, por cada modelo UML 2.0 representando un producto concreto, sólo puede existir un paquete estereotipado de dicha forma. Esta última restricción se expresa por medio de OCL.

La Figura 4.1 muestra un ejemplo de creación de un producto concreto dentro de la línea de productos software para hogares inteligentes. En este caso, se trata de un producto donde se ha incluido exclusivamente la característica de SmartEnergyMng, lo que implica que deben seleccionarse además las características WindowMng y HeaterMng, ya que SmartEnergyMng necesita que ambas características estén instaladas en un producto final para poder funcionar. Dicha dependencia queda especificada de forma explícita a través de las relaciones *merge* existentes entre SmartEnergyMng y WindowMng y HeaterMng. Debido a dichas relaciones, es imposible crear un producto que incluya SmartEnergyMng pero no WindowMng o HeaterMng.

La siguiente sección describe como este modelo arquitectónico puede transformarse automáticamente en el código necesario para crear una implementación concreta y completamente funcional de un producto software concreto.

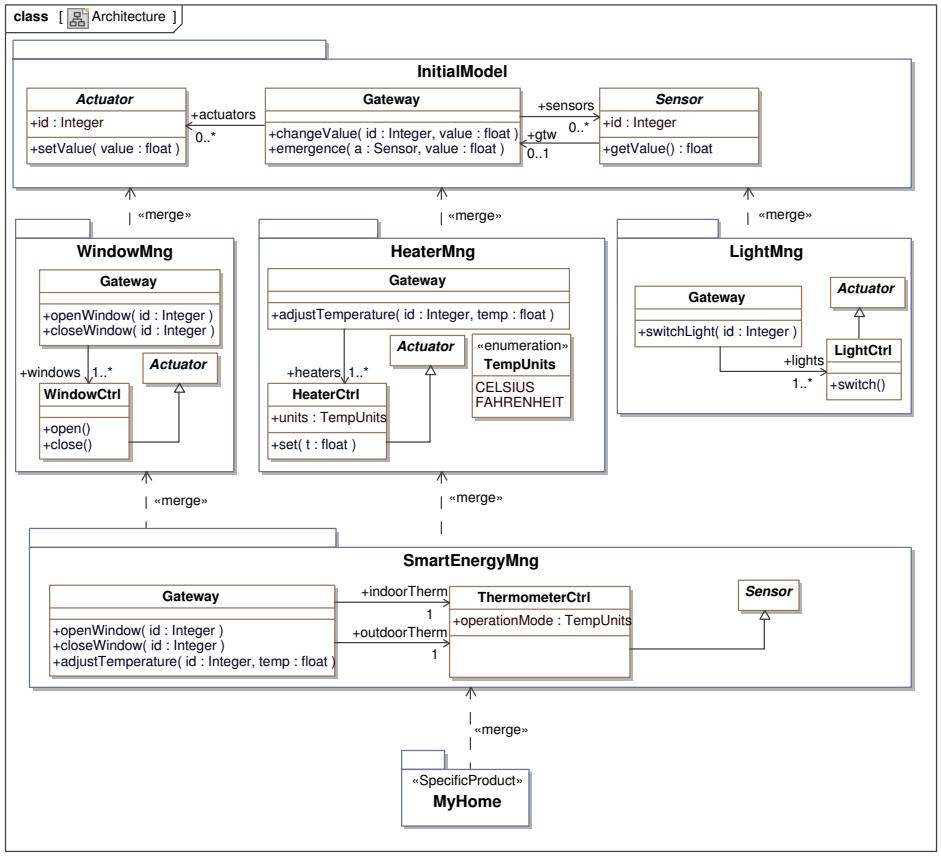

Figura 4.1: Configuración de un hogar inteligente completo

4.3. Algoritmo para la implementación del producto específico

Para obtener una implementación completamente funcional de un producto concreto, con unas características determinadas, de acuerdo con el *Slicer Pattern* (ver Sección 2.5), es necesario: (1) crear una clase parcial por cada clase que deba estar incluida en el producto final; (2) crear la *versión limpia* de cada constructor y cada método que deba estar incluido en el producto final; y (3) hacer que dichas versiones limpias deleguen en las *versiones sucias* que corresponda.

El primer paso en el proceso de transformación es crear un nuevo proyecto y una nueva carpeta que represente el producto final.

Para calcular todas las clases que deben estar incluidas en el producto final, recorremos el modelo desde el paquete que representa el producto concreto, y que será siempre un paquete *hoja*, hacia arriba, hasta llegar a la raíz, o raíces, del modelo orientado a características. Normalmente,

siempre hay un modelo raíz que contiene los elementos que son comunes a todos los productos. En nuestro caso de la Figura 4.1, dicho recorrido generaría dos caminos distintos: (1) SmartEnergyMng, WindowMng, BaseSystem; y (2) SmartEnergyMng, HeaterMng, BaseSystem.

Obviamente, una clase puede aparecer en más de un paquete. Por ejemplo, la clase Gateway aparece en todos los paquetes, a excepción del que representa el producto final, de la Figura 4.1. No obstante, cada clase que esté en un camino desde el paquete hoja al paquete raíz, solo debe incluirse una vez en el producto final, aunque ésta aparezca varias veces. Por cada clase distinta presente en algunos de los caminos del paquete hoja a la raíz, generamos una nueva clase parcial, que colocamos en la carpeta que representa el producto final

A continuación, para cada clase, debemos calcular todos los métodos limpios que debemos generar. Para ello, al igual que ocurría con las clases parciales, recorremos todos los caminos existentes de raíz a hoja. Para cada clase, por cada método distinto, es decir, con diferente signatura, creamos una versión limpia de dicho método dentro de la clase parcial incluida en el producto final. El proceso de generación del esqueleto del método se realiza reutilizando las plantillas de generación de código y facilidades creadas para la Ingeniería de Dominio.

Por último, quedaría por generar el código de cada método, de forma que este delegue en la versión sucia del método que corresponda. Es esta fase del algoritmo de generación de código la que entraña mayor dificultad, porque pueden darse diversos casos. Analizamos cada caso a continuación.

4.3.1. Caso 1: Sólo existe una *versión sucia* del método

Se trata del caso más simple. Sólo existe una *versión sucia* del método, por lo que hay que hacer es delegar en él. En el ejemplo de la Figura 4.1, para la clase Gateway, el método WindowCtrl.open solo está implementado en la característica WindowMng, por lo que el código generado para la *versión limpia* de dicho método simplemente contendría un delegado a la *versión sucia* windowMng_open de dicho método.

4.3.2. Caso 2: Existen varias *versiones sucias* independientes

En este caso, existen varias *versiones sucias* independientes del método. Por independientes entendemos que dichas versiones se encuentran en caminos distintos, y ninguna es *alcanzable* desde la otra. El ejemplo de la Figura 4.1 no contiene ninguno de estos casos, por lo que usamos el ejemplo de la Figura 4.2, extraído del mismo caso de estudio. Por razones de concisión y brevedad, en dicho ejemplo sólo aparecen aquellos detalles que son relevantes para explicar la situación que estamos tratando.

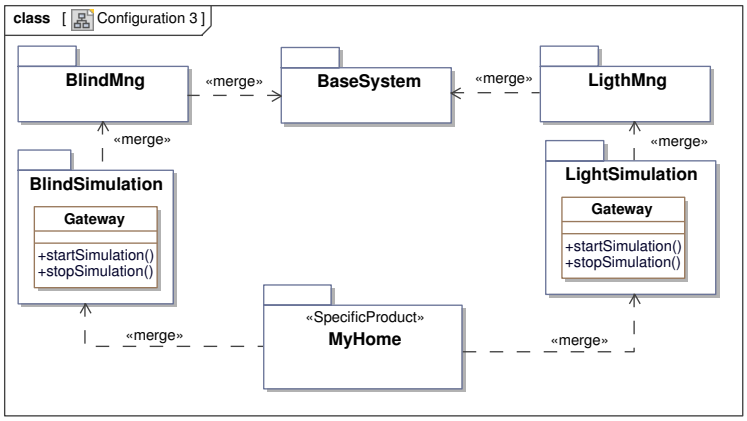

Figura 4.2: Configuración de una casa inteligente con versiones sucias independientes de un mismo método

En este caso, se trata de una configuración de un producto concreto que incluye las características BlindSimulation y LightSimulation, encargadas de simular la presencia de habitantes en el hogar mediante el movimiento de persianas y el encendido y apagado de luces. Obviamente, ambas características dependen de las características de gestión automática de persianas (BlindMng) y gestión automática de luces LightMng, respectivamente. En cada una de estas características, se extiende la clase Gateway para que contenga métodos para iniciar y detener la simulación (startSimulation y stopSimulation, respectivamente).

En este caso, la versión limpia de los métodos startSimulation y stopSimulation, contenida dentro del paquete MyHome, debe delegar en las versiones sucias del método perteneciente tanto a BlindSimulation como LightSimulation, ya que en este caso, al inicial la simulación de presencia, deben activarse tanto la simulación de persianas como de luces. Es decir, por ejemplo, el método startSimulation, de MyHome, contendrá en su interior llamadas a blindSimulation_startSimulation y a lightSimulation_startSimualtion. El orden el cual se generen estas llamadas es irrelevante.

4.3.3. Caso 3: Existen *versiones sucias* dependientes de un método

En este caso, existen varias *versiones sucias* de un método, pero dichas versiones están en el mismo camino, estando una situada a mayor profundidad, más cerca del paquete *hoja* que la otra. Por ejemplo, en el caso de la Figura 4.1, existen dos versiones del método openWindow, de la clase Gateway, en las características SmartEnergyMng y WindowMng. Ambas están en el mismo camino del paquete hoja al paquete raíz (SmartEnergy, WindowMng,

BaseSystem).

En este caso, de acuerdo a la semántica del modelo UML 2.0, la versión del paquete SmartEnergyMng debe sobrescribir la versión del paquete WindowMng. Por tanto, la versión limpia del método debe invocar en este caso sólo a la versión sucia del paquete SmartEnergyMng, ya que se entiende que esta versión *más profunda* es la más actualizada. En caso de haber más de dos versiones dependientes, siempre se escogería la versión más profunda.

4.3.4. Caso 4: Existen *versiones sucias* dependientes e independientes de un método

Este caso se trata de una combinación de los casos 2 y 3. Existen diversas versiones de un método. Estas versiones las podemos agrupar en varios subconjuntos, donde cada subconjunto contiene todas las versiones que son dependientes entre sí. Por ejemplo, para la Figura 4.1, consideremos el caso del constructor de la clase Gateway. Supongamos además, que la característica LightMng también está seleccionada. Dicho constructor, aunque no se muestra de forma explícita en el diagrama, estaría presente en todas las versiones de dicha clase, presente en cada una de las características del sistema.

Para la Figura 4.1, hay tres caminos distintos (recordemos que la característica LightMng también está seleccionada, aunque no aparezca en la figura): (1) MyHome, SmartEnergyMng, WindowMng, BaseSystem; (2) MyHome, SmartEnergyMng, HeaterMng, BaseSystem; y, (3) MyHome, LightMng, BaseSystem. En este caso, habría 5 versiones del constructor de la clase *Gateway*, más concretamente smartEnergyMng_Gateway, windowMng_Gateway, heaterMng_Gateway, lightMng_Gateway y baseSystemMng_Gateway. Tendríamos dos conjuntos de métodos dependientes, { smartEnergyMng_Gateway, windowMng_Gateway, heaterMng_Gateway, baseSystemMng_Gateway }, y { lightMng_Gateway y baseSystemMng_Gateway }.

En este caso, la versión limpia del método debe invocar la versión más profunda de cada conjunto independiente de métodos, en este caso smartEnergyMng_Gateway y lightMng_Gateway. Al igual que en el caso 2, el orden en el cual se invocan estos métodos es irrelevante.

La siguiente sección describe, de forma muy superficial, como se organizan las plantillas encargadas de implementar este no trivial algoritmo de generación de código.

4.4. Generadores de Código C#

Esta sección detalla la secuenciación de las plantillas de generación de código creadas para implementar el algoritmo de la sección anterior. La Figura 4.3 muestra dicha secuenciación.

El punto de partida es idéntico al utilizado para la fase de *Ingeniería del Dominio*; es decir, el generador de código llamado ProjectCreation, el cual

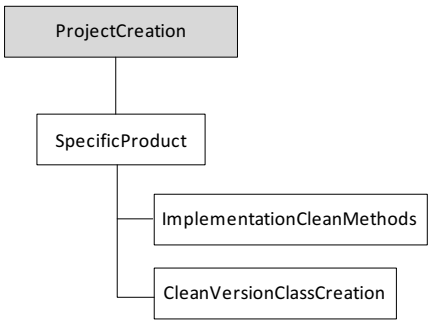

Figura 4.3: Secuencia de ejecución de las plantillas de generación de código (Ingeniería de Aplicaciones)

crea el proyecto Visual Studio 2010 para el producto concreto. Este plantilla invoca a su vez a la plantilla SpecificProduct, que es la que gobierna el proceso de generación de código a nivel de *Ingeniería de la Aplicación*. Para ello, se invocan las siguientes plantillas:

La plantilla ImplementationCleanMethods procesa por cada uno de los caminos incluidos en el producto final, los paquetes pertenecientes a dicho camino y almacena sus clases e interfaces y las *versiones limpias* de sus respectivos métodos siguiendo el algoritmo descrito en la Sección 4.3.

Acto seguido, se invoca la plantilla CleanVersionClassCreation para cada una de las clases e interfaces obtenidas mediante la invocación a la plantilla ImplementationCleanMethods, recibe como argumento las *versiones limpias* de los métodos y genera los ficheros fuente del producto específico acorde a la configuración elegida por el usuario.

Cada plantilla invocada hace uso a su vez de otras subplantillas, que al igual que en la fase de *Ingeniería del Dominio* por razones de claridad y espacio no detallamos.

Una vez implementados los generadores de código para la fase de *Ingeniería de Aplicación*, procedimos a realizar las pruebas pertinentes.

4.5. Pruebas

Una vez implementados los generadores de código para la fase de *Ingeniería de Aplicaciones*, el siguiente paso era diseñar y ejecutar las pruebas necesarias que permitiesen comprobar el correcto funcionamiento de estos generadores. Para diseñar las pruebas, siguiendo el mismo procedimiento que en el caso anterior, utilizando la técnica de clases de equivalencia y valores límites, para luego completar con casos específicos que permitiesen alcanzar el 100 % de la cobertura. A diferencia de la fase de *Ingeniería del Dominio*, en esta fase no se utilizó *EUnit* para ejecutar dichas pruebas, ya que dicha herramienta no se ajustaba a nuestras necesidades. Por tanto, se crearon los

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

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

casos de prueba y se ejecutaron a mano, analizando de forma también manual si la salida producida coincidía con la esperada. La Tabla 4.1 muestra algunos de los casos de prueba ejecutados.

Una vez creados los generadores de código, el siguiente paso es empaquetarlos para posibilitar su distribución y uso. La siguiente sección describe como se realiza dicha fase de despliegue. Tras ejecutar estos casos de prueba y comprobar que los generadores de código funcionaban correctamente, dábamos por concluida la labore de desarrollo de los generadores de código, restando solo su empaquetado y despliegue.

4.6. Despliegue

Una vez creados los generadores de código, el siguiente paso es empaquetarlos y distribuirlos de forma que puedan ser usados de la forma más cómoda posible por diferentes desarrolladores para la creación de productos concretos pertenecientes a nuestra familia de productos.

La forma más fácil de distribuir nuestra infraestructura es crear un plugin, que se integre con la plataforma Eclipse, que permita la generación de un proyecto Visual Studio 2010 que contenga dos proyectos: (1) un proyecto que contiene todos los ficheros fuente para cada una de las características del modelo UML dado; y (2) otro proyecto que contiene los ficheros fuente específicos para las características seleccionadas en la creación de un producto específico del modelo UML proporcionado. Esta sección describe el proceso de creación de dichas extensiones.

El ámbito de desarollo de plug-ins de Eclipse, *Plug-in Development Environment* (PDE), proporciona un entorno cómodo para crear plug-ins e integrarlos con la plataforma Eclipse. De esta forma, realizamos el cambio de contexto necesario para poder invocar nuestro generador de código principal, escrito en EGL, desde el lenguaje de programación java que es el utilizado para la creación de plug-ins en dicho entorno. Además, debemos preprocesar el modelo UML de entrada y realizar ciertas modificaciones sobre el ya que algunas de sus directivas no son compatibles con dicho entorno. Una vez analizado el modelo de entrada procedemos a aplicar los generadores de código de igual forma que venimos haciendo en las etapas anteriores

del desarrollo del Proyecto. Por último, añadimos el entorno gráfico que nos permitirá acceder al plug-in cómodamente desde un menú integrado con Eclipse o desde un botón en la barra de tareas del mismo. A continuación exportamos el proyecto como un plug-in y lo instalamos en nuestro sistema, de esta forma cuando iniciemos Eclipse podremos realizar la transformación de modelo UML a código C# de forma sencilla.

Tanto los generadores de código como la documentación están disponibles a través de una página web, realizada con el único objetivo de dar a conocer el presente proyecto. Dicha web posee 5 secciones, las cuales se describen a continuación: **Introducción:** Contiene un breve resumen acerca del ámbito y los objetivos del proyecto. **Descargas:** Contiene los generadores de código creados. **Documentación:** Contiene la documentación de usuario y vídeos relacionados con el proyecto. **Publicaciones:** Contendrá todas las publicaciones relacionadas con el proyecto. **Contacto:** Datos de contacto con los participantes del proyecto.

4.7. 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 sus correspondientes pruebas, por último se explicado el proceso de empaquetardo y distribución del trabajo realizado durante este Proyecto de Fin de Carrera.