Capítulo 1

Antecedentes

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

1.1. Caso de Estudio: Software para Hogares Inteligentes

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

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

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

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

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

Funciones básicas

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

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

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

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

Funciones complejas

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

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

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

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

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

1.2. Líneas de Producto Software

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

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

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

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

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

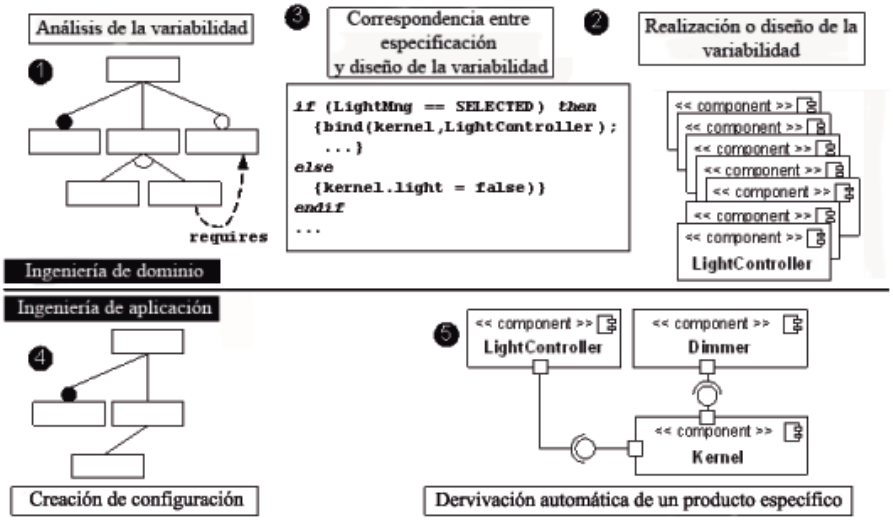

Figura 1.1: Proceso de Desarrollo de una línea de producto software

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

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

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

1.3. TENTE

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

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

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

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

Esta metodología posee diversas ventajas:

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

[1] www.ample-project.net

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

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

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

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

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

Problema 1: Resistencia a cambiar el lenguaje de implementación habitual

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

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

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

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

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

Problema 2: Obligatoriedad de usar la plataforma .NET

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

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

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

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

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

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

1.4. Diseño Orientado a Características con UML

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

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

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

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

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

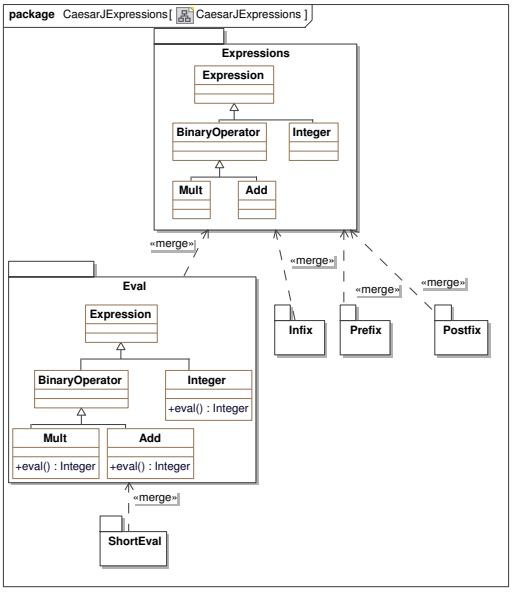

Figura 1.2: Diseño para resolver el problema de las expresiones con CaesarJ

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

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

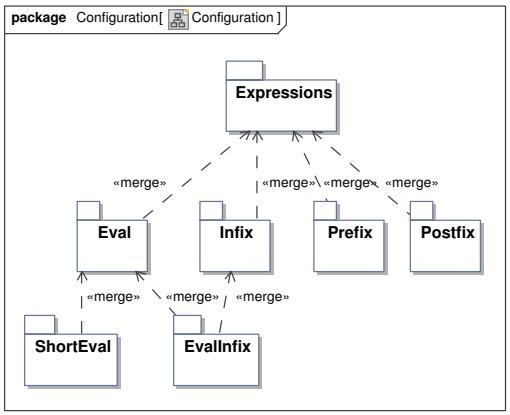

Figura 1.3: Composición de las características Eval e Infix en nuevo producto con CaesarJ

Para realizar una configuración, es decir, para crear un producto concreto

por composición de características, simplemente hay que crear una nueva familia de clases que herede de las familias de clases que correspondan a las características seleccionadas. La figura **??** muestra como se crearía un producto nuevo mediante la composición de las características Eval e Infix.

Con el fin de solventar las limitaciones expuestas en la Sección **??** y apoyándose en el diseño orientado a características (Sección **??**), en la siguiente sección se describirá el patrón para resolver dichas limitaciones, denominado Patrón Slicer.

1.5. Slicer Pattern

Antes de proceder a la explicación del Patrón Slicer, es conveniente presentar el concepto de Clase Parcial C# de manera breve al lector. Las clases parciales 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 [**?**, **?**], dado que cada incremento en funcionalidad perteneciente a una característica se podría encapsular en una clase parcial separada. Sin embargo, las clases parciales tiene un serio inconveniente para ser utilizadas como un mecanismo de apoyo a la orientación de características: no son compatibles con la extensión de métodos ni la reescritura. El Patrón Slicer surge como patrón para solución de esta limitación.

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

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

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

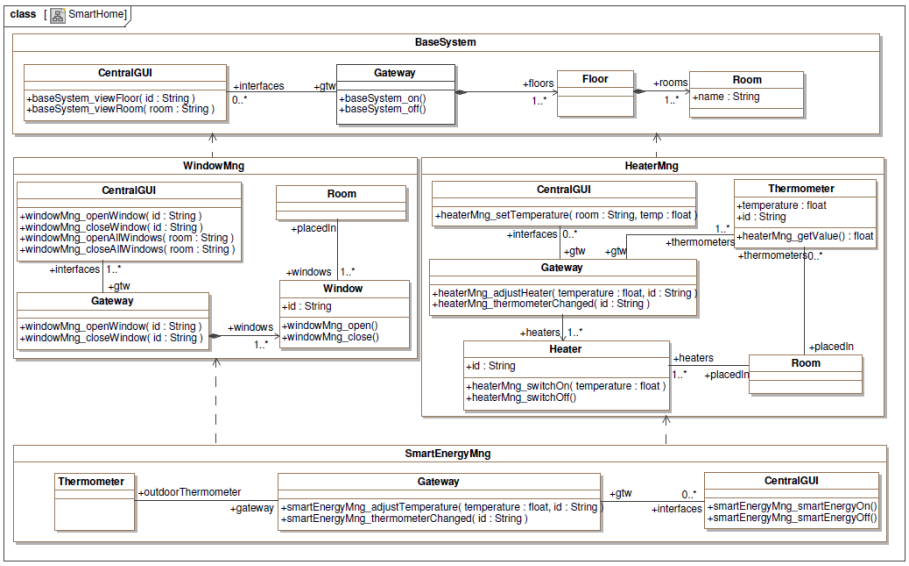

Figura 1.4: Proceso de Desarrollo de una Línea de Productos Software

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

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

El listing **??** muestra cómo se aplica dicha técnica. Se puede apreciar cómo la lógica del constructor para *Gateway* ha sido encapsulada en un método llamado baseSystem_initGateway (listing **??** líneas 03-06). La misma técnica ha sido usada en el listing **??** líneas 10-12 con la característica Win-

dowMng. El siguiente paso es encontrar un mecanismo que permita componer dichos fragmentos de acuerdo a una selección de características dada, de esta forma, como se puede apreciar en el listing **??** líneas 16-20, el constructor para la clase Gateway es creado con las características seleccionadas por el usuario final, en el caso analizado, por ejemplo, solo se ha seleccionado la característica WindowMng.

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

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

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

Listing 1.1: Código para el constructor de la clase Gateway usando clases parciales y patrón slicer

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

1.6. Generación de Código con Epsilon

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

centrado en la generación de código y por tanto en los lenguajes *Epsilon Generation Language* (EGL) y *Epsilon Object Language*(EOL) proporcionados por Epsilon, que se analizan con más detalle a continuación.

Epsilon Generation Language(EGL)

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

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

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

Listing 1.2: Generación del nombre de cada Class contenida en un modelo de entrada

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

Listing 1.3: Resultado de la generación del nombre de cada Class contenida en un modelo de entrada

EGL no solo nos permite generar estos sencillos ejemplos, es una herramienta mucho más potente que permite la generación de funciones creadas específicamente por el usuario para facilitar la obtención de datos de los modelos de entrada. Tal como se aprecia en el listing **??** se puede programar una operación que implemente la cabecera de una clase Java y de esta forma invocar dicha función tantas veces como sea necesario en aquellos programas EGL que lo necesiten. En el listing **??** (línea 4) la operación *visibility* indica

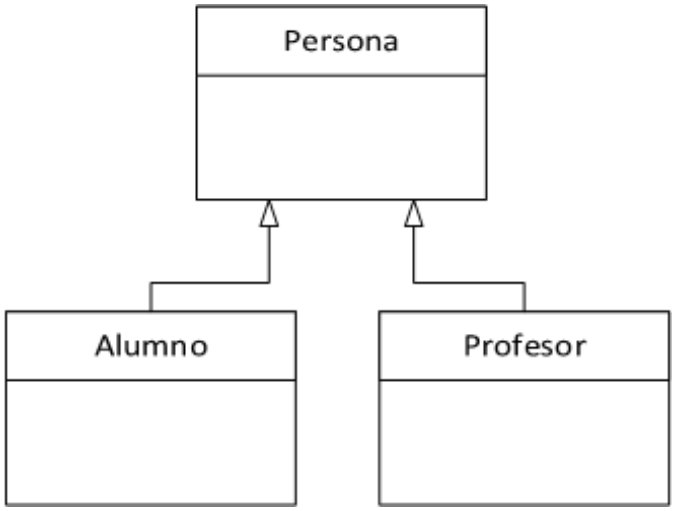

Figura 1.5: Ejemplo de modelo de entrada para generación de código con EGL

la visibilidad de *self* siendo en este caso la visibilidad de la clase tal como muestra la línea 3. Mientras que la operación *name*, al igual que en el listing **??**, se refiere al nombre del elemento actual, en este caso el nombre de la clase.

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

Listing 1.4: Uso de una operación que especifica el texto generado para una declaración de una clase Java

El tipo Template proporciona tres utilidades básicas al usuario:

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

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

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

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

Listing 1.5: Almacenar el nombre de cada Class en disco

Epsilon Object Language(EOL)

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

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

Listing 1.6: Ejemplo de operación EOL

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

1.7. Sumario

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

En el siguiente capítulo profundizaremos acerca del .....

Capítulo 2

Ingeniería del Dominio

En este capítulo se describe la fase de *Ingeniería del Dominio* (en inglés, *Domain Engineering*) de nuestra línea de productos software. Dentro de dicha fase .....

2.1. Transformaciones de Modelo UML a C#

2.2. Generadores de Código C#

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

```
01 [%import "ReturnParameterCreation.egl";
02 import "ParametersCreation.egl";
03 import "../Operations.eol";
04 operation Element classMethods(currentPackage: String, path: String): String {
05  ...
06  opers=private()+void()+currentPackage+"_init"
        +self.firstToUpperCase()+"␣(␣)␣{}\n\t\t";
07  ...
08  for (oper in self.getOperations()){
09      for (par in oper.ownedParameter){
10          ...
11          if (oper.type==null){
12              isReturn=false;
13          }else{
14              if (par.direction.toString().equals("return")){
15                  if (not par.type.name.isDefined()){
16                      isReturn=false;
17                  }else{
18                      isReturn=true;
19                  }//if-par-type
20              }//if-par-direction
21          }//if-oper-type
22      }//for-parameters
23      if (isReturn){
24          operations_return.add(oper);
25      }else{
```

```
26          operations_void.add(oper);
27      }
28  }//for-operations
29  for (oper in operations_void) {
30      if (oper.name==""){
31          methodname="method_"+iter;
32      }else{
33          methodname=oper.name;
34      }
35      opers=opers+oper.visibility()+oper.abstract()+oper.esStatic()
              +virtual()+void()+currentPackage+"_"+methodname
              +"␣("+oper.parameters(currentPackage, path)+")␣{}\n\t\t";
36      // Increase the iterator
37      iter=iter+1;
38  }
39  for (oper in operations_return) {
40      if (oper.name==""){
41          methodname="method_"+iter;
42      }else{
43          methodname=oper.name;
44      }
45      opers=opers+oper.visibility()+oper.abstract()+oper.esStatic()
             +virtual()+oper.returnParameter(currentPackage, path)
             +"␣"+currentPackage+"_"+methodname
             +"␣("+oper.parameters(currentPackage, path)
             +")␣{}\n\t\t";
46      // Increase the iterator
47      iter=iter+1;
48  }
49  return opers;
50 }%]
```

Listing 2.1: Implementación del generador de código MethodsCreation

Para introducir al lector en la implementación de los generadores de código, vamos a analizar en detalle uno de los generadores de código más sencillos: MethodsCreation, el fichero fuente aparece en el listing 1.1. Vamos a proceder al análisis detallado del mismo:

- Líneas 1-3, generadores de código que utiliza y fichero Operations.eol que contiene las funciones básicas comunes a los generadores de código.
- Línea 4, descripción de la función que retornará el texto generado.
- Línea 6, texto correspondiente al constructor de la clase de la forma <nombre del paquete>_init<nombre de la clase>.
- Líneas 8-28, tratamos una a una todas las operaciones descritas en elemento actual (clase o interfaz).
- Líneas 9-22, en cada operación recorremos todos y cada uno de los parámetros.
- Líneas 11-13, si la operación no tiene definido un tipo, es decir, si el usuario ha obviado especificar si la función devuelve una colección, un entero, un elemento de una clase, etc, por defecto se trata como una operación void (operación que no retorna ningún valor).

- Línea 15, si la operación tiene un tipo de retorno definido, comprobamos si dicho parámetro es de retorno.
- Línea 16, si el parámetro es de retorno pero no está definido vuelve a ser tratada como una operación void.
- Línea 18, si el parámetro es de retorno y tiene un tipo definido se trata de una operación que sí retorna un valor.
- Línea 24, si la operación que está siendo analizada retorna un valor, se añade a la lista de operaciones que devuelven un valor.
- Línea 26, si la operación que está siendo analizada no retorna un valor, se añade a la lista de operaciones que no devuelven un valor.
- Línea 29-38, añadir al string resultado la información correspondiente a los métodos de la clase actual que no retornan ningún valor (métodos void).
- Línea 31, si el método no tiene un nombre definido, se otorga un nombre por defecto.
- Línea 35, se realizan llamadas a los generadores de código para obtener los parámetros de la función.
- Línea 39-48, de manera análoga a las operaciones que no retornan ningún valor, se procede a añadir al string resultado los métodos de la clase que sí retornan un valor.
- Línea 49, se retorna el string con todos los métodos de la clase o interfaz actual.

Un vez explicado un ejemplo sencillos, las siguiente sección domain:sec:ejcomplejo explica ejemplos más complejos que quedan a la curiosidad del lector.

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

Antes de comenzar a exponer los ejemplos más complejo, vamos a introducir aquellos conceptos que pasamos por algo en la sección 1.2 dada su complejidad: las relaciones bidireccionales y la herencia múltiple.

Relaciones Bidireccionales

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

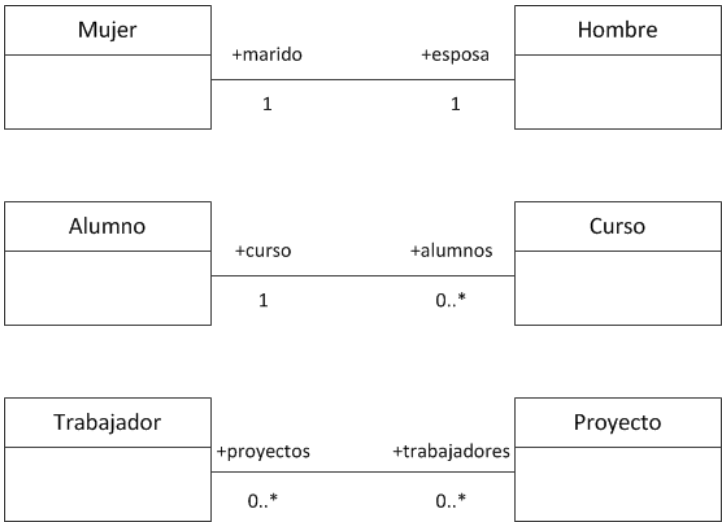

Figura 2.1: Tipos de relaciones bidireccionales

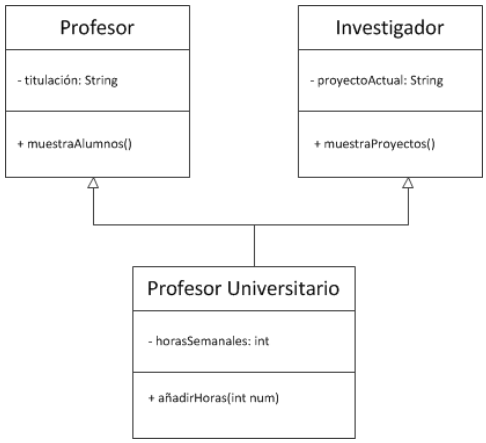

Figura 2.2: Tipos de relaciones bidireccionales

- Bidireccionales one to one, en las que cada clase recibe un elemento de la clase opuesta (figura 1.1, Mujer-Marido).
- Bidireccionales one to many, en la que una de las clases recibe un elemento de la clase destino y la otra recibe una colección de elementos de la clase opuesta (figura 1.1, Alumno-Curso).
- Bidireccionales many to many, en la que ambas clases reciben colecciones de la clase opuesta (figura 1.1, Trabajadores-Proyectos).

Herencia múltiple

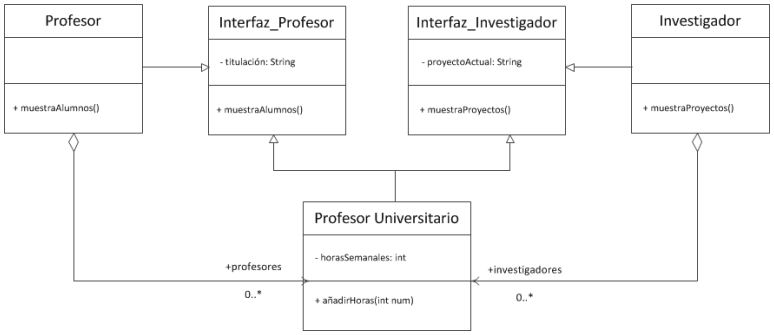

Figura 2.3: Tipos de relaciones bidireccionales

Los modelos UML admiten la herencia múltiple de clases, pero lenguajes como C# no por lo que hay que transformar el modelo iniciar para adaptarlo y que funcione correctamente. Podemos encontrarnos con un conjunto de clases como el descrito en la figura 1.2 donde se puede apreciar que la clase Profesor Universitario presenta herencia múltiple de las clases Profesor e Investigador, por tanto debemos procesar la información de forma que funcione correctamente en C#, las modificaciones que debemos realizar son:

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

2.5. Sumario