Desarrollo de una Línea de Productos Software utilizando las clases parciales C#: el patrón *Slicer*

Alejandro Pérez Ruiz, Pablo Sánchez Barreiro
Dpto. Ingeniería Informática y Electrónica
Universidad de Cantabria
Santander (Cantabria), España
{perezruiza, p.sanchez}@unican.es

Resumen. Las clases parciales de C# permiten dividir el comportamiento global de una clase en diversos fragmentos. Estos fragmentos pueden luego combinarse de diversas formas, produciendo clases completas con comportamientos similares pero ligeramente diferentes, en función de nuestras necesidades. La semejanza de este mecanismo con la programación orientada a características ha hecho que algunos autores hayan considerado las clases parciales de C# como un mecanismo apropiado para el desarrollo de líneas de productos de software. Sin embargo, un reciente estudio ha demostrado que las clases parciales de C# por sí solas no son suficientes para tal propósito, ya que presentan problemas a la hora de extender comportamientos preexistente. Para solventar dicha deficiencia, este artículo presenta un patrón, denominado *Slicer Pattern*, mediante el cual es posible implementar de forma completa diseños orientados a características en C# mediante clases parciales. La principal ventaja de dicho patrón es que permite utilizar el lenguaje C# como lenguaje orientado a características, sin necesidad de extender dicho lenguaje.

Palabras claves: Línea de Productos Software, Desarrollo Software Orientado a Características, TENTE, Clases Parciales C#, .NET

1 Introducción

El objetivo final de una *Línea de Productos Software* [1] es producir sistemas software similares con el grado de adaptabilidad y personalización que ofrece el software hecho a medida, pero con precios cercanos a los del software producido en masa o para un amplio abanico de usuarios.

Por tanto, una *Línea de Productos Software* debe ser capaz de gestionar de forma eficiente un conjunto de aplicaciones software similares las cuales comparten una serie de características comunes, pero que también presentan variaciones entre ellas. De acuerdo a las necesidades particulares de cada cliente, algunas de estas variaciones, comúnmente denominadas *características*, se incorporarán en su producto software, mientras que otras quedarán excluidas.

De este modo, para que una *Línea de Productos Software* sea rentable, ésta debe gestionar de la forma más eficiente posible la incorporación y exclusión de dichas

características. Con este objetivo en mente, han ido apareciendo en los últimos años diversas soluciones para al modularización, composición y gestión de características. Ejemplos de dichas soluciones son los lenguajes de programación como *CaesarJ* [2], o herramientas como *FeatureHouse* [3].

Dentro de este abanico de soluciones, Laguna et al [4] propusieron la utilización de las *clases parciales de C#* como mecanismo para encapsular características. Las clases parciales de C# permiten dividir el comportamiento global de una clase en diversos fragmentos. Estos fragmentos pueden luego combinarse de diversas formas, produciendo clases completas con comportamientos similares pero ligeramente diferentes, en función de nuestras necesidades. Por tanto, la idea base sería la de encapsular cada característica dentro de una clase parcial.

Sin embargo, un reciente estudio [5] ha demostrado que dicha idea presenta algunos deficiencias, ya que las clases parciales de C# no permiten sobrescribir o refinar comportamientos ya predefinidos. Este artículo presenta un patrón de diseño, denominado *Slicer Pattern*, que solventa dichos problemas, permitiendo utilizar las clases parciales de C# como un mecanismo más para la implementación de líneas de productos software. Para evaluar la aplicabilidad de dicho patrón, se ha implementado una línea de productos para el desarrollo de software de hogares inteligentes [6] utilizando dicho patrón.

La principal ventaja con respecto a otras técnicas, es que las clases parciales de C# son un mecanismo nativo del lenguaje, por lo que es necesario modificar el mismo. Por tanto, las empresas que trabajen con tecnologías .NET pueden utilizarlo sin necesidad de cambiar o modificar sus herramientas actuales de trabajo.

Tras esta introducción, este artículo se estructura como sigue: La Sección 2 describe el caso de estudio elegido en este trabajo para utilizar las clases parciales. La Sección 3 expone las características que hemos identificado que un lenguaje debe poseer para sea apto para la implementación de una línea de productos software. A continuación en la Sección 4 se explican las peculiaridades de las clases parciales C# y muy brevemente se indican sus principales carencias para desarrollar líneas de productos software. Después en la sección 5 se detallará el patrón software que hemos desarrollado para solventar las deficiencias de las clases parciales. En la Sección 6 se explica cómo se deben componer las características bajo el patrón anterior. La Sección 7 presenta una breve discusión sobre el trabajo aquí desarrollado. Durante la Sección 8 se ilustran algunos de los trabajos relativos. Por último en la Sección 9 se presentan las conclusiones y los trabajos futuros.

2 Caso de estudio: una línea de productos software para hogares inteligentes

Esta sección describe el caso de estudio que utilizaremos a lo largo del artículo, que es una línea de producto para el desarrollo sistemas software para el control de hogares inteligentes o automatizados. Dicho caso de estudio proviene de un producto industrial desarrollado por Siemens AG, y ha sido ampliamente utilizado dentro de proyectos europeo denominado AMPLE (Aspect-oriented Model-driven Product Line

Engineering) para evaluar de diversos trabajos relacionados con las líneas de productos software [6].

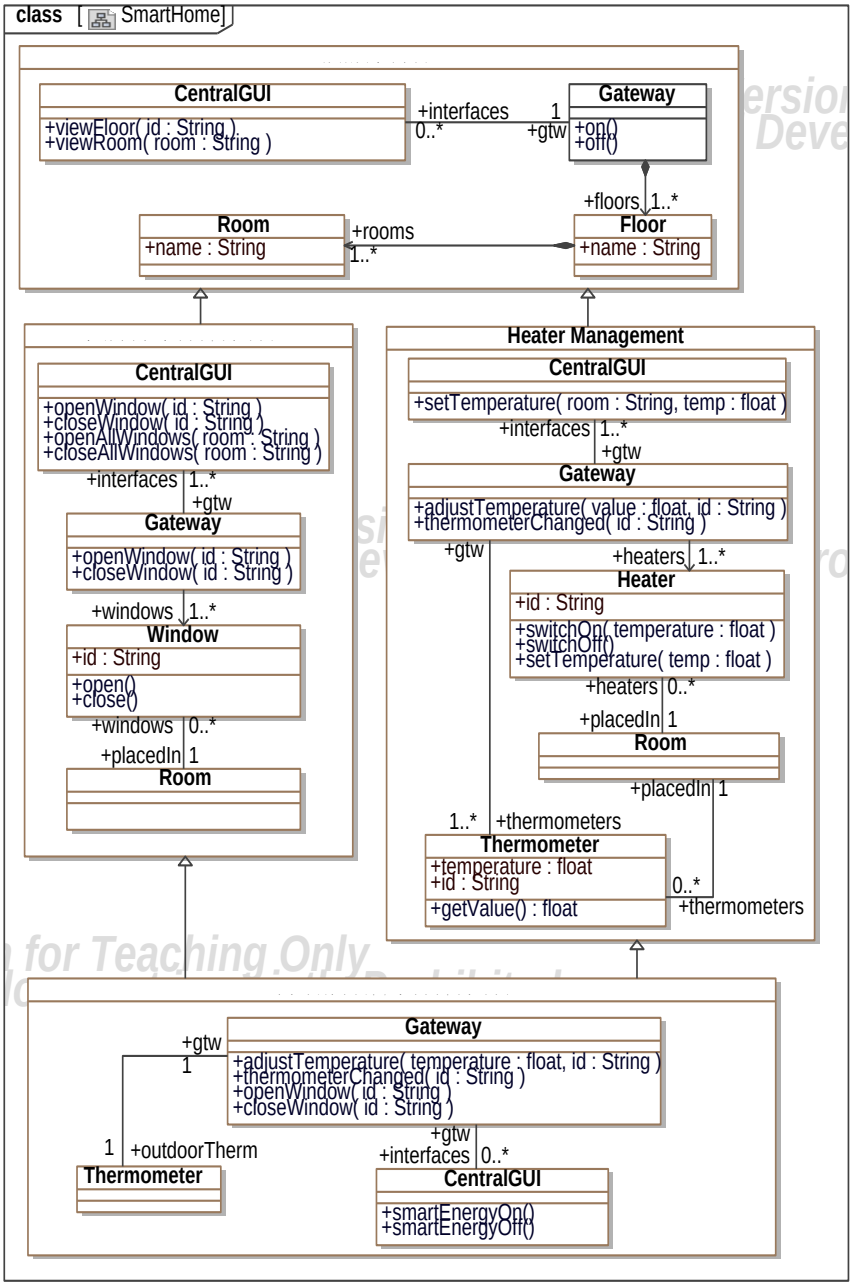

Figura 1. Diseño de la LPS para hogares inteligentes

El objetivo del software de un hogar inteligente es mejorar el confort y la seguridad de los habitantes de dicho hogar, además, de tratar de reducir su consumo de energía. Para ello el software debe controlar y coordinar los diversos dispositivos domóticos instalados en el hogar, tales como luces, calefactores o ventanas.

El software para un hogar inteligente puede adquirirse con diferentes opciones, en función de los dispositivos instalados. Para este artículo, consideramos como opciones funcionalidades para la gestión de luces, ventanas y radiadores. Además, en el caso de que se hayan instalado ventanas y radiadores automatizados, podrá instalarse una funcionalidad de de control inteligente de la energía, la cual se encarga de coordinar ventanas y radiadores en aras de una mayor eficiencia energética. Por ejemplo, en caso de que se desee calentar una habitación y se deban activar los radiadores, dicha funcionalidad se asegurará de cerrar antes las ventanas, para evitar que se disipe el calor.

La Figura 1 muestra un diseño UML donde cada característica opcional de nuestra línea de productos software ha sido diseñada por separado y encapsulada dentro de un rectángulo con el nombre de la característica que representa. En general, el diseño se compone de una serie de clases que representan los diversos dispositivos que están interconectados, los cuales actúan como sensores (e.g. termómetros) o actuadores (e.g., radiadores). Todos los sensores y actuadores se comunican a través de un dispositivo especial que actúa como *puerta de enlace* (Gateway), el cual encarga se coordinar de forma adecuada los diferentes dispositivos. Los habitantes del hogar pueden interactuar con estos dispositivos a través de una interfaz gráfica, la cual envía los comandos adecuadas a la *puerta de enlace.*

La siguiente sección describe que características sería deseable encontrar en un lenguaje para la implementación de un diseño como el de la Figura 1.

3 Propiedades deseables de un lenguaje de programación para la implementación de líneas de productos software

De acuerdo a [4], un lenguaje de programación deberá poseer los siguientes elementos de cara a facilitar la implementación de un diseño de una línea de productos software como el mostrado en la Figura 1.

Soporte para añadir características de manera incremental

El diseño de una línea de productos software suele partir de una base que se va extendiendo con nuevas funcionalidades. En nuestro caso, partimos de un conjunto base de clases, que se va refinando y aumentando en función de las características que deban incorporarse. Por tanto, el lenguaje a utilizar debería ofrecer mecanismos para extender el comportamiento de un conjunto de clases ya existentes. Por ejemplo, en nuestro caso de estudio, la incorporación de la gestión de ventana a un producto implica la incorporación de una nueva clase `Window` al diseño del producto, así como la incorporación de nuevos métodos a clases como `Gateway` (e.g., `openWindow`). En

este caso, las extensiones son puramente aditivas, nuevos comportamientos se suman a los ya existentes.

No obstante, estas extensiones implican a veces no sólo la incorporación de nuevas funcionalidades, sino que es además necesario sobrescribir o refinar comportamientos ya existentes. Por ejemplo, en el caso de incorporación de la funcionalidad de control inteligente de la energía, habría que reescribir el método que enciende los radiadores, para que, en caso de que dicho controlo inteligente esté activado, antes de activar los radiadores, se cierren las ventanas.

Por lo tanto, sería deseable encontrar en un lenguaje de programación para la implementación de una línea de productos software mecanismos que permitiesen tanto añadir nuevas funcionalidades como sobrescribir comportamientos ya existentes.

Mecanismos para encapsular y componer características

Todas las extensiones pertenecientes a una misma característica deberían realizarse de forma atómica. Es decir, o se añaden todas o no se añade ninguna. Por ejemplo, en el caso de la característica de gestión de ventanas, si añadimos los métodos `openWindow` y `closeWindow` a la clase `Gateway`, también deberíamos añadir la clase `Window`. De otra forma, el producto creado sería incorrecto. Por lo tanto, un lenguaje para la implementación de líneas de productos software debería proporcionar mecanismos que controlen cómo se añaden las características a un producto software concreto.

Para poder controlar este proceso de composición, lo primero que necesitamos es saber qué elementos concretos pertenecen a una característica determinada. Es decir, necesitamos algún mecanismo que nos permita encapsular los elementos pertenecientes a una misma característica en algún tipo de módulo. A continuación, la composición de características se realizaría a nivel de módulo, de forma que todas las extensiones contenidas en un mismo módulo se lleven a cabo de forma atómica.

Por tanto, un lenguaje de programación para la implementación de líneas de productos software debería proporcionar facilidades para encapsular y componer características.

Análisis y gestión de la corrección de la composición.

En una línea de productos software, no todas las combinaciones de características dan lugar a productos válidos. Por ejemplo, para poder añadir a un producto la característica de gestión inteligente de la energía, necesitamos que previamente se hayan incluido las funcionalidades para la gestión de radiadores y ventanas.

Por tanto, sería deseable que un lenguaje para la implementación de líneas de productos software pudiese detectar composiciones erróneas, alertando a los desarrolladores de los posibles fallos. Además, en la medida de lo posible, dichos fallos deberían solventarse, de forma automática. Por ejemplo, si se detectase que se va a incluir las características para la gestión inteligente de la energía pero faltase la gestión de ventanas, ésta última debería añadirse automáticamente de forma implícita.

La siguiente sección describe cómo funcionan las clases parciales de C# y explica cuáles son los problemas encontradas en las mismas para la implementación de líneas de productos software.

4 Clases parciales de C# para la implementación de líneas de productos software

Las clases parciales permiten a los desarrolladores dividir la implementación de una clase en diferentes archivos. Cada archivo contiene parte de la funcionalidad global de la clase. Todas estas partes se pueden combinar en tiempo de compilación, generando una única clase con la funcionalidad de todas las clases parciales compuestas.

La Figura 2 muestra un ejemplo de cómo la clase **`Gateway`** puede implementarse usando clases parciales. Consideramos la implementación para los conjuntos de clases **`InitialModel`** y **`WindowMng`** (Ver Figura 1). La implementación de la clase para estas dos características se ha dividido en dos ficheros distintos con el mismo nombre, pero colocados en distintas carpetas (**`InitialModel/Gateway.cs`** y **`WindowMng/Gateway.cs`**) (Ver Figura 2 líneas 01-10 y líneas 11-19).

La clase parcial **`Gateway`** para la característica **`BaseSystem`** (Figura 2, líneas 01-10) contiene listas para almacenar referencias a las plantas del hogar y las interfaces de usuario (líneas 03 y 04). Además, posee métodos para apagar o encender la puerta de enlace (líneas 06-07).

La clase parcial `Gateway` para la característica **`WindowMng`** (Figura 4, líneas 11-19) extiende la clase parcial `Gateway` de `InitialModel` con un conjunto de referencias a objetos ventana (línea 14); así como con algunos métodos para la gestión de ventanas, tales como `openWindow` (línea 16).

En lo sucesivo, nos referiremos a una clase parcial perteneciente a una característica dada usando la siguiente notación: `nombreCaracterística::nombreClase`. Usando esta notación, `WindowMng::Gateway` significaría "La clase parcial `Gateway` perteneciente la característica **`WindowMng`**".

Una vez dividida la implementación de una clase en diferentes clases parciales, podemos decidir qué partes de la misma se incluirán en un producto final mediante la manipulación del fichero de construcción del proyecto. Este fichero es un archivo XML que indica qué archivos formarán parte del proceso de compilación. Por tanto, manipulando este fichero XML podemos incluir o excluir clases parciales de acuerdo a las características que el cliente quiera que tenga su producto final.

A modo de ejemplo la parte final de la figura 2 (líneas 20-30) muestra parte de este fichero de construcción (`SmartHome.csproject`). En este ejemplo concreto se indica que las clases parciales `BaseSystem::Gateway` y `WindowMng::Gateway` deben incuirse en el proceso de compilación, mientras que las clases parciales `LightMng::Gateway` y `HeaterMng::Gateway` serán excluidas. Por lo tanto, el compilador generará una clase `Gateway` con soporte para la gestión de ventanas, pero sin soporte para la gestión de luces y radiadores.

```
Archivo BaseSystem/Gateway.cs
-------------------------------------------------------
01 namespace SmartHome {
02  public partial class Gateway {
03    protected IList<Floor> floors;
04    protected IList<CentralGUI> interfaces;
05
06    public void on() {...}
07    public void addFloor(Floor floor) {...}
08    ....
09  }
10 }
Archivo WindowMng/Gateway.cs
-------------------------------------------------------
11 namespace SmartHome{
12
13  public partial class Gateway {
14    protected IList<Window> windows;
15
16    public void openWindow(String id) {...}
17    ...
18  }
19 }
Archivo SmartHome.csproj
-------------------------------------------------------
20 </Project>
21 ...
22 <ItemGroup>
23 <Compile Include="BaseSystem\Gateway.cs" />
24 <Compile Include="WindowMng\Gateway.cs" />
25 <!-- <Compile Include="LightMng\Gateway.cs" />
26 <Compile Include="HeaterMng\Gateway.cs" />
27 -->
28 ...
29 </ItemGroup>
30 </Project>
```

Figura 2. Implementación del `Gateway` utilizando clases parciales.

Motivados por esta idea, Laguna et al [4] propusieron utilizar las clases parciales de C# para la implementación de líneas de productos software. No obstante, un reciente estudio [5] ha demostrado que la idea de Laguna et al no es aplicable tal cual,

aunque las clases parciales presentan algunas ventajas en comparación a la orientación a objetos.

La principal ventaja encontrada es que las clases parciales evitan que se use la herencia como mecanismo de extensión, lo que reduce el número de clases diferentes dentro de nuestra aplicación. Además, evita el uso de la herencia múltiple en los casos en los que necesitamos extender dos clases pertenecientes a características distintas al mismo tiempo. Este caso se daría, por ejemplo, para la clase `Gateway` de la característica `SmartEnergyMng`, la cual necesitaría extender las implementaciones de Gateway de las características `WindowMng` y `HeaterMng` al mismo tiempo.

Por otro lado las clases parciales de C# no permiten redefinir comportamientos existentes. Por ejemplo, la clase `SmartEnergy::Gateway` necesita poder sobrescribir el método `adjustTemperature` de la clase `HeaterMng::Gateway`. Esto no es posible utilizando clases parciales, ya que no es posible tener dos métodos con el mismo nombre en diferentes clases parciales. Es decir, las clases parciales sólo soportan la extensión por incorporación de nuevos comportamientos, pero no permite la extensión por sobre-escritura o refinamiento.

Además, la gestión de las características debe realizarse a nivel de clase, ya que ésta se realiza mediante la inclusión o exclusión de clases parciales del fichero de construcción del proyecto.

El segundo inconveniente podría resolverse añadiendo cierto soporte de herramientas para manejar dichas inclusiones a nivel de característica. Bastaría con asociar cada clase parcial a una característica, y una herramienta adecuada se podría encargar de manipular el fichero en función de las características seleccionadas. Sin embargo, el primer problema es más grave, ya que sin sobre-escritura no podemos implementar multitud de líneas de productos software.

La siguiente sección describe el patrón que hemos ideado para solventar este problema, y que es la principal contribución de este artículo.

5 Slicer Pattern

El *Slicer Pattern* es un patrón que permite que podamos extender y sobrescribir métodos definidos en las clases parciales de C#. El origen del problema de la sobre-escritura de métodos es que no podemos tener métodos con el mismo nombre y cabecera en diferentes clases parciales.

Por tanto, si evitamos que se repitan los nombres de los métodos, el problema estaría solucionado. De acuerdo a esta idea, para evitar que los nombres de los métodos se repitan, añadimos como prefijo a cada método en una clase parcial con el nombre de la característica a la que pertenece el método. De esta forma aseguramos que métodos repetidos en diferentes características tendrán diferente nombre por tener diferente prefijo.

Por ejemplo, el método `adjustTemperature` en la clase parcial `HeaterMng::Gateway` tendrá como nombre `heaterMng_adjustTemperature`, mientras que este método en la clase parcial `SmartEnergy::Gateway` se deno-

minará `smartEnergyMng_adjustTemperature`. El resto de las clases invocará la versión del método sin prefijar, por motivos que expondremos más abajo.

Con esta sencilla operación conseguimos que las implementaciones de estos métodos puedan convivir en clases parciales diferentes sin colisionar. Por tanto, sólo nos faltaría ser capaz de combinar dichos métodos en función de un conjunto de características seleccionadas concreto. Esto se realiza tal como se describe a continuación.

Cuando queremos componer un producto concreto, por cada clase que deba ser incluida en el producto final, se crea una nueva clase parcial. En dicha clase parcial se creará una versión sin prefijar de cada método de la clase que deba estar presente en el producto final. Por ejemplo, si se ha seleccionada como única característica para un producto concreto `HeaterMng`, crearemos una clase parcial Gateway en la que estén presente los métodos `adjustTemperature` y `thermoterChanged`, sin prefijar.

A continuación, hacemos que la versión sin prefijar de cada método delegue en la versión prefijada que corresponda, de acuerdo al conjunto de características seleccionado. Por ejemplo, en nuestro caso, el método `adjustTemperature` delegaría en `heaterMng_adjustTemperature`, ya que la característica para la gestión inteligente de energía no ha sido seleccionada para nuestro producto. Si ésta estuviese seleccionada, el método `adjustTemperature` debería delegar en `smartEnergyMng_adjustTemperature`.

Recordemos que las clases que utilizan el método `adjustTemperature` nunca invocan las versiones prefijadas el mismo, sino la versión sin prefijar. Esto es lo que asegura que siempre se invoque la versión correcta de un método, de acuerdo al conjunto de características seleccionadas. Explicamos esta idea con ayuda de un ejemplo.

Supongamos en primer lugar que la característica **`HeaterMng`** ha sido seleccionada, pero **`SmartEnergyMng`** no, y analicemos como se comportaría el método `setTemperature` de la clase `CentralGUI`. Esta clase representa la interfaz gráfica de la aplicación. Cuando el usuario ajusta el valor de la temperatura en dicha interfaz, se invoca el método `setTemperature`. Este método a su vez invoca el método `adjustTemperature` de la clase `Gateway`. Tal como hemos comentado anteriormente, siempre se invocan las versiones sin prefijar. La versión sin prefijar en este caso delegaría en la versión prefijada `heaterMng_adjustTemperature`.

Supongamos ahora que además de **`HeaterMng`** se haya seleccionado **`SmartEnergyMng`**. En este caso, la cadena de invocaciones entre métodos sería la misma, pero al llegar a la versión si prefijar de `adjustTemperature`, se invocaría `smartEnergyMng_adjustTemperature`.

Por tanto, lo único que debemos variar en función de las características a incluir en un producto es el contenido de las versiones sin prefijar de un método. Son estas versiones sin prefijar las que soportan la variabilidad de la línea de productos software.

En adelante, llamaremos *versiones sucias* de un método sus versiones prefijadas. A las versiones sin prefijos les denominaremos *versiones limpias*.

Con esta estrategia sí podemos extender y sobrescribir métodos definidos en otras clases parciales, haciendo que podamos implementar extensiones no sólo por extensión, sino también por sobre-escritura.

La técnica de prefijado no es aplicable a los constructores de las clases, ya que éstos tienen unos nombres específicos que impiden el prefijado. La solución en este caso consiste en encapsular la lógica de cada constructor en un método separado que sí pueda ser renombrado. Concretamente, lo que haremos es que cada clase parcial `C` correspondiente a una característica `F` posea un método privado llamado `<F>_init<C>`, que contenga la lógica del constructor para dicha clase y características.

Para componer los constructores se aplica una técnica parecida a la anterior. Por cada clase parcial que deba estar incluida en el producto final, se crea una nueva clase parcial, sino estaba creada antes, para realizar la composición. En dicha clase parcial se añade el constructor de la clase, el cual delegará en el método `init` de la clase parcial que corresponda.

Una de las ventajas de este patrón, es que el proceso de composición es automatizable. La siguiente sección describe dicho proceso de composición de forma algorítmica.

6 Proceso de composición de características de acuerdo al Slicer Pattern

Sea *caracteristicasSeleccionadas* el conjunto de características seleccionadas que debemos componer. El proceso a ejecutar sería el que se describe a continuación:

1. Creamos una nueva carpeta con un identificador apropiado para el producto a desarrollar. Por ejemplo, *HogarJuanPerezGomez*. Denominamos a dicha carpeta o directorio
2. Calculamos el conjunto *clasesSeleccionadas* de todas las clases pertenecientes a las características en *caracteristicasSeleccionadas*.
3. Por cada clase *C* en *ClasesSeleccionadas*, creamos una nueva clase parcial *C* en el directorio de composición, que fue creado en el punto 1.
4. Por cada clase *C* creada en el paso 3, creamos un constructor en dicha clase parcial *C*.
5. Calculamos el conjunto *metodosSeleccionados* de todos los métodos en c*lases-Seleccionadas*. Dos métodos iguales salvo por el prefijo se consideran métodos iguales, por lo que sólo se añaden una vez al conjunto.
6. Por cada método *M* en *metodosSeleccionados* perteneciente a una clase *C*, creamos una versión limpia del método *M* en dicha clase C_i.
7. A continuación cada versión limpia de un método delegará su función en la correspondiente versión sucia, de acuerdo a la característica seleccionada. Esta versión es la versión más profunda en el grafo de extensión, o las versiones más profundas, si hubiese varias.

8. De igual forma, hacemos que el constructor de cada clase parcial delegue en el método *init* de la clase parcial que corresponda, siguiendo las mismas pautas del paso 7.
9. Finalmente, creamos el fichero de construcción XML que indica qué clases parciales se deben compilar. Se incluirán todas las clases parciales contenidas en carpetas correspondientes. Es decir, se excluirán todas las clases parciales pertenecientes a características no seleccionadas y se incluirán todas aquellas contenidas en las características seleccionadas.

Actualmente, este proceso está automatizado mediante un conjunto de generadores de código. A partir de una selección de características, los generadores de códigos recorren la estructura de directorios del proyecto, calculando las clases parciales, métodos, constructores y delegados que son necesarios generar. Por tanto, este código de composición, normalmente conocido como *boilerplate code*, no tiene que escribirlo el programador, sino que es automáticamente generado. Por tanto, el programador no necesita escribir este código, sino simplemente seleccionar un conjunto de características que se adecue a las necesidades del usuario y ejecutar los generadores de código.

La siguiente sección analiza de forma genérica la aplicabilidad de nuestra propuesta.

7 Discusión

Como hemos comentado previamente, existe un amplio abanico de soluciones para la implementación de líneas de producto software, desde lenguajes como *CaesarJ*, a herramientas como *FeatureHouse*. Por lo tanto, es lógico que el lector se pregunte, por qué alguien estaría interesado en utilizar nuestro patrón en lugar de una de las soluciones ya existentes, las cuales, además, suelen ofrecer ventajas con respecto a nuestra solución.

No obstante, estos lenguajes y herramientas cuentan con una barrera de adaptación que muchas empresas no pueden afrontar [7]. Por ejemplo, en nuestro caso, hemos detectado que las empresas de nuestro entorno son bastante reticentes a adoptar nuevos lenguajes de programación o herramientas debido fundamentalmente a dos motivos:

1. El coste de aprendizaje que ello supone
2. El uso de un nuevo lenguaje de programación podría dejar obsoletas muchas herramientas, como suites de pruebas o entornos de integración continua, que sean actualmente utilizados en la empresa. Por ejemplo, no existen plugins que permitan analizar automáticamente la calidad del código escrito en CaesarJ utilizando herramientas como Sonar [8].

Nuestro patrón ofrece una solución de compromiso para dichas compañías, al estar basado en mecanismos nativos del lenguaje C#, no requiere de la adopción de ningún lenguaje o herramienta nueva por parte de las empresas que ya trabajan con la plataforma .NET.

8 Trabajos relacionados

Este es el primer trabajo que describe como una línea de productos software puede ser implementada usando las clases parciales de C#. Para ello, se ha presentado el patrón denominado *Slicer* en este artículo que nos permite extender y sobrescribir los métodos de las clases parciales.

Este trabajo está inspirado en un estudio previo [4]. En él, las clases parciales de C# fueron presentadas como un mecanismo para implementar las líneas de productos software. Sin embargo, en dicho trabajo sólo se propuso la idea pero no fue analizada en profundidad. De hecho, un estudio posterior [5] demostró que la idea propuesta por Laguna et al [4] no era viable. El trabajo presentado en este artículo soluciona los problemas encontrados en la propuesta de Laguna et al, haciendo viable la utilización de clases parciales de C# para la implementación de líneas de productos software.

Algunos libros [9, 10] explican cómo implementar una línea de productos software utilizando la plataforma .NET. Sin embargo, ninguno de ellos usa las clases parciales de C# como un mecanismo para gestionar la variabilidad, únicamente se apoyan en técnicas de orientación a objetos.

El refinamiento de clases [11], como por ejemplo el proporcionado en el lenguaje Jak, es similar a las clases parciales. No obstante, el refinamiento de clases requiere extender un lenguaje convencional de programación, por ejemplo Java, para crear uno nuevo, por ejemplo Jak. Opuestamente las clases parciales son soportadas nativamente por el lenguaje de programación C#.

En otro trabajo previo [12] se propuso un proceso de desarrollo para líneas de productos software utilizando Java como lenguaje objetivo. En este enfoque, las características se separan y encapsulan en módulos bien definidos en la fase de modelado. Después, utilizando técnicas de desarrollo dirigido por modelos, estos módulos se componen en la fase de diseño. A continuación, se genera el código fuente desde los modelos compuestos. Sin embargo, esta propuesta no utiliza clases parciales, sino herencia como mecanismo de extensión. Las clases parciales presentan ciertas ventajas frente a la herencia, como evitar la necesidad de tener que usar herencia múltiple, que suele introducir cierta complejidad accidental en las soluciones creadas.

9 Conclusiones y trabajos futuros

Este trabajo ha presentado el patrón denominado *Slicer*. Este patrón sirve para implementar líneas de productos software usando las clases parciales de C#. Una de las razones que hace muy interesante este patrón reside en que aunque la mayoría de las empresas de software están interesadas en las líneas de productos software, muchas de ellas son reticentes en la adopción de nuevas metodologías de desarrollo que requieran cambios en su lenguaje de programación. Estos cambios implicarían llevar a cabo un esfuerzo que muchas pequeñas compañías no podrían afrontar. Debido a ello, nos hemos visto en la necesidad de encontrar una manera de implementar una línea de productos software usando un lenguaje de programación convencional como C#.

Como la mayoría de compañías software de nuestro entorno trabajan con la plataforma .NET, hemos utilizado un lenguaje provisto en dicha plataforma. Así que inspirados por un trabajo previo [4], hemos decido implementar una línea de productos software usando las clases parciales. Aunque encontramos un grave inconveniente ya que las clases parciales no permiten extender o sobrescribir métodos. Por lo tanto, hemos creado el patrón denominado *Slicer pattern*, que soluciona esta deficiencia.

Como trabajo futuro, nos gustaría integrar este patrón en la metodología de TENTE [13], de modo que la mayor parte del código relacionado con este patrón se puede derivar automáticamente desde modelos de diseño. También esperamos llevar a cabo más estudios empíricos y diseñar un mecanismo para asegurar la encapsulación de características.

Agradecimientos. Este trabajo ha sido financiado en parte por el Programa de Becas Predoctorales de la Universidad de Cantabria, y el Gobierno de España y los fondos FEDER en el proyecto TIN2011-28567-C03-02 (HI-PARTES).

Referencias

1. Pohl, K., Böckle, G., van der Linden, F. J., October 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer. (SI, 1)
2. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K., April 2006. An Overview of CaesarJ. In: Rashid, A., Aksit, M. (Eds.), Transacations on Aspect-Oriented Software Development I. Vol. 3880 of Lecture Notes in Computer Science. pp. 135–173.
3. Apel, S., Kästner, C., Lengauer., C., 2012. Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE Transactions on Software Engineering. To appear.
4. Laguna, M. A., González-Baixauli, B., Marqués, J. M., 2007. Seamless Development of Software Product Lines. In: Consel, C., Lawall, J. L. (Eds.), 6th International Conference on Generative Programming and Component Engineering (GPCE). pp. 85–94.
5. Sánchez, P., López, Elio A. and Fuentes, L. "Implementing Feature-Oriented Decompositions using C# Partial Classes: An Exploratory Study" Journal of Research and Practice in Information Technology 45(1):21-36, February 2013
6. Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way, Rummler, A., Royer, J.-C. (Eds), pp. 27–51 Cambridge University Press, November 2011.
7. Kreuger, C., July-August 2002. Eliminating the Adoption Barrier. IEEE Software 19 (4), 28–31.
8. G. Ann Campbell (Autor), Patroklos P. Papapetrou, SonarQube in Action, Manning Publications; 2013.
9. Lenz, G., Wienands, C., 2006. Practical Software Factories in .NET. Apress.
10. Cwalina, K., Abrams, B., November 2008. Framework Design Guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries. Addison-Wesley.
11. Batory, D. S., May 2004. Feature-Oriented Programming and the AHEAD Tool Suite. In: 26th International Conference on Software Engineering (ICSE). Edinburgh (Scotland, United Kingdom), pp. 702–703.
12. Groher, I., Völter, M., December 2009. Aspect-Oriented Model-Driven Software Product Line Engineering. In: Katz, S., Ossher, H., France, R. B., Jézéquel, J.-M. (Eds.), Transac-

tions on Aspect-Oriented Software Development VI (Special Issue on Aspects and Model-Driven Engineering). Vol. 5560 of Lecture Notes in Computer Science. pp. 111–152.
13. Lidia Fuentes, Carlos Nebrera and Pablo Sánchez. "Feature-Oriented Model-Driven Software Product Lines: The TENTE approach" Proc. of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE), Eric Yu, Johann Eder and Colette Rolland (Eds), CEURS Workshops(453):67-72, Amsterdam (The Netherlands), June 2009.