Trabajo Final de Máster

Programación Orientada a Característica Usando Clases Parciales en C#

Máster en Ingeniería del Software e Inteligencia Artificial

Curso 2008-2009

Realizado Por:

Elio Alexis López Salamanca

Dirigido Por:

Dra. Lidia Fuentes y Dr. Pablo Sánchez.

Universidad de Málaga.

Septiembre 2010.

Contenido

Índice de Figuras

Índice de Tablas

1 Introducción.

La programación orientada a característica (Prehofer, 1997), tiene como objetivo encapsular conjuntos coherentes de la funcionalidad de un sistema software en módulos independientes y fácilmente componibles. Dichos módulos reciben el nombre de característica. Una característica es comúnmente considerada como un incremento de la funcionalidad de un sistema (Batory et al, 2005; Zave, 1999).

Las clases parciales de C# (Albahari y Albahari, 2010) permiten dividir la implementación de una clase en un conjunto de ficheros. Cada archivo contiene un fragmento de la funcionalidad de la clase, es decir, cada fragmento contiene un incremento sobre la funcionalidad de dicha clase. Por lo tanto, tal como han sugerido otros autores (Laguna et al, 2007; Warmer y Kleppe, 2006), las clases parciales C# podrían resultar un mecanismo adecuado para la implementación de descomposiciones orientadas a características. La idea sería implementar cada característica en una o más clases parciales, las cuales constituirían un incremento sobre la funcionalidad de la aplicación. Aunque esta idea en principio parece prometedora, hasta ahora no ha sido analizada en profundidad. Por tanto, no existen actualmente evidencias empíricas que confirmen las hipótesis iniciales. Hasta donde alcanza nuestro conocimiento, no existe ningún análisis en profundidad de los aspectos positivos y negativos de las clases parciales de C# como un mecanismo para implementar diseños orientados a características.

Con el objetivo de suplir esta carencia, este trabajo presenta los resultados de aplicar las clases parciales de C# a un caso de estudio industrial, con el objetivo de analizar en profundidad ventajas e inconvenientes de las clases parciales de C# como mecanismo para la programación orientada a características. Las clases parciales no son un mecanismo exclusivo del lenguaje C#, sino que se pueden encontrar en otros lenguajes de programación tales como Ruby. Por tanto, los resultado de este trabajo no serían sólo aplicables al lenguaje C#, sino que podrían extenderse a otros lenguajes que soporten clases parciales. No obstante, antes de aplicar los resultados de este trabajo a otros lenguajes soportando clases parciales, habría que reinterpretar ligeramente los mismos, dado que cada lenguaje puede introducir pequeñas variaciones en su modelo de clases parciales que obliguen a reinterpretar parte de las conclusiones extraídas en este trabajo.

Como caso de estudio para la realización de nuestro análisis, hemos escogido una línea de productos software para la automatización de hogares (Groher y Völter, 2009; Batory, 2005, Sánchez et al, 2007; Nebrera et al, 2008). Este caso estudio se ha empleado durante tres años dentro del proyecto AMPLE[1] para la evaluación de los resultados de la investigación sobre líneas de productos software y tecnologías orientadas a características. Durante este período, se han generado un diseño en UML orientado a características (Nebrera et al, 2008; Fuentes et al, 2009) más una implementación en el lenguaje orientado a características CaesarJ (Aracic et al, 2006), de dicho caso de estudio. En este trabajo, hemos optado por emplear este caso estudio por abarcar un amplio rango de diferentes situaciones y escenarios (Sánchez et al, 2007), característica por la cual ha demostrado ser un excelente punto de referencia para

[1] http://www.ample-project.net

evaluar tecnologías orientadas a características (Groher y Völter, 2009; Fuentes et al, 2009; Sánchez et al, 2007; Nebrera et al, 2008; AMPLE D3.2, 2007; AMPLE D3.4, 2007).

Para evaluar las capacidades de las clases parciales C# como mecanismo de programación orientada a características, en primer lugar hemos identificado que elementos serían deseables que apareciesen contenidos en un lenguaje orientado a características. A continuación, hemos intentado obtener una implementación orientada a características, utilizando clases parciales C#, basada en el diseño, desarrollado en el contexto del proyecto AMPLE (Sánchez et al, 2007; Nebrera et al, 2008; Fuentes et al, 2009), de nuestro caso de estudio. Finalmente, se han evaluado los resultados obtenidos frente al conjunto de características previamente identificadas, haciendo mención de los aspectos positivos y negativos de las clases parciales C# como mecanismo para implementar diseños orientados a características.

Este trabajo está estructurado de la siguiente manera: la sección 2 describe un estado del arte acerca de la programación orientados a característica, así como, el detalle del caso estudio empleado y algunos antecedentes relacionados a las clases parciales C#. En la sección 3 se identifican las propiedades deseables que debe soportar un lenguaje de programación orientado a característica y detalla los resultados obtenidos del análisis a las clases parciales de C# como mecanismo para implementar descomposiciones orientadas a características, así como, algunos trabajos relacionados a clases parciales. La sección 4, describe un sumario y se mencionan futuros trabajos. Finalmente la sección 5, se exponen los conocimientos y habilidades desarrolladas durante el curso del máster.

2 Estado del Arte

2.1 Líneas de Productos Software.

Una línea de productos software, plantea crear una infraestructura necesaria para la rápida construcción de aplicaciones software. Basada principalmente en representar tanto características comunes como características variables entre sistemas software (Pohl et al. 2005; Clements y Northrop, 2002; Kaköla y Dueñas, 2006; Laguna et al. 2007; Hallsteinsen et al. 2006).

El principal objetivo de una línea de productos de software es minimizar tiempo y costos de los desarrollos (Nebrera, 2009), así como, aumentar la calidad de los productos obtenidos a partir de las posibles configuraciones, desde la base común de características de la línea de productos. El desarrollo de software bajo este paradigma generalmente comprende de dos diferentes procesos relacionados entre sí, conocidos como *ingeniería de dominio* e *ingeniería de aplicación*.

El nivel de ingeniería de dominio, inicia desde el documento de requisitos que describe una familia de productos relacionados para un segmento específico de mercado. Por lo que, es necesario diseñar una arquitectura y una implementación de referencia para cada familia de productos. Dicha arquitectura de referencia representará los elementos comunes de todos los productos en la familia y a su vez, debe contener mecanismos que permitan introducir variaciones a los diferentes productos pertenecientes a la familia.

En el nivel de ingeniería de aplicación, se comienza desde un documento de requisitos para un producto específico dentro de esa familia. Este documento establece las variaciones específicas a ser incluidas en el producto en concreto. Conociendo esta información, se introducen dichas variaciones en la arquitectura e implementación de referencia, obteniendo como resultado un producto final. Por lo tanto, el principal beneficio de adoptar una Línea de Productos Software es la reducción de tiempo y esfuerzo de desarrollo para la obtención de productos específicos pertenecientes a una misma familia.

2.2 Caso de Estudio: Una línea de productos para automatización de hogares.

En este trabajo se ha empleado como caso estudio una línea de productos para automatización de hogares, llamados hogares inteligentes (*Smart Home* en inglés). 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. Hemos elegido este dominio puesto que, el uso del enfoque basado en Líneas de Productos Software se hace casi imperativo, debido a la gran variabilidad existente en estos productos. Esta variabilidad se debe tanto a motivos de hardware, dado que los dispositivos a ser controlados e interconectados pueden variar grandemente, como funcionales, dado que existen multitud de funcionalidades que se pueden ofrecer de manera opcional o alternativa al usuario, no siendo necesario que un determinado hogar las posea todas ellas.

Ejemplos comunes de tareas automatizadas dentro de un hogar inteligente son el control de luces, ventanas, puertas, persianas, aparatos de frío/calor, etc. Del mismo modo, se puede incrementar la seguridad de sus habitantes mediante sistemas automatizados de vigilancia y alertas de potenciales situaciones de riesgo, tales como detección de humos o ventanas abiertas cuando se abandona el hogar.

Para implementar esta funcionalidad, un hogar inteligente hace uso de una serie de sensores y actuadores, los cuales se encuentran normalmente conectados a un dispositivo central que los coordina, el cual se conoce como *puerta de enlace* o *Gateway*. Dicho *Gateway* se encarga de leer los datos de los sensores, procesarlos y enviar las órdenes adecuadas a los actuadores. De igual modo, el Gateway puede recibir comandos de los usuarios para modificar elementos de la casa. Dichos comandos son procesados por el *Gateway,* el cual manda las órdenes adecuadas a los actuadores para atender la petición del usuario, cuando fuese posible de acuerdo con las políticas de gestión del hogar especificadas.

En concreto, para el presente trabajo, consideraremos que el software para el hogar inteligente debe implementar las siguientes funcionalidades, todas ellas con carácter opcional.

Funciones Básicas

Se entiende por funciones básicas aquellas que son independientes entre sí, y por tanto pueden incluirse o excluirse de la configuración de un hogar determinado sin prejuicio para las otras.

A. **Control automático de luces**: los habitantes de la casa deben ser capaces de encender, apagar y regular la intensidad de las luces de la casa. El ajuste debe realizarse especificando valores de intensidad.

B. **Control automático de ventanas**: los habitantes de la casa deben ser capaces de controlar automáticamente las ventanas de la casa, especificando el porcentaje de apertura para cada ventana.

C. **Control automático de la temperatura interior:** el usuario puede enviar órdenes al sistema para adecuar la temperatura de su casa a sus gustos y/o necesidades. El sistema interactuará con los diferentes aparatos de frío/calor (aires acondicionados y/o radiadores) para establecer la temperatura indicada por el usuario.

Funciones Complejas.

Estas funciones extienden una o más funciones simples, las cuales manejan de forma coordinada para realizar comportamientos más complejos. Por tanto, dependen de las funciones simples para funcionar correctamente. En caso de que una función compleja sea seleccionada para ser incluida en la configuración de un hogar, es necesario también incluir las funciones básicas que dicha función compleja extiende o usa.

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

Diseño de la línea de productos para hogares inteligentes.

El modelo de características de la Figura 1, representa de forma resumida las funcionalidades que dispone la línea de productos Automatización del Hogar empleada en este trabajo. Este modelo muestra la gestión de luces (LightMng), ventanas (WindowMng) y calentadores (HeaterMng) como variabilidades opcionales. Así como, el manejo inteligente de energía (SmartEnergyMng), el cual coordina funciones con la gestión de ventanas y calefacción para el ahorro de energía. Como por ejemplo, para calentar una habitación el software verifica si la temperatura del exterior es más baja que dentro del cuarto, de ser así, la aplicación cerrará las ventas para evitar pérdida de energía. Claro está, esta funcionalidad requiere que la gestión de ventanas y calefacción sean seleccionadas, como lo especifica la restricción C1 en la Figura.

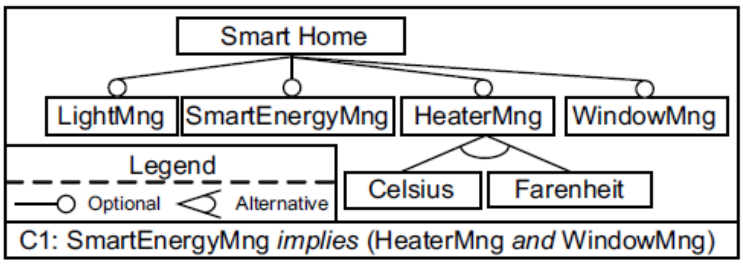

Fig. 1. Modelo de Característica simplificado del hogar inteligente.

La Figura 2, muestra un modelo UML que soporta el diseño de las variabilidades identificadas en el modelo de características. Cada característica de grano grueso como LightMng es encapsulada en un paquete UML, siguiendo lo establecido en el proyecto AMPLE (Sánchez et al, 2007; Nebrera et al, 2008; Fuentes et al, 2009), en cuanto a las reglas de diseño orientado a características. Todos los dispositivos están conectados a una puerta de enlace central (Gateway), el cual, es el encargado de procesar las peticiones de los habitantes de la casa y coordinar todos estos dispositivos. Cada paquete, ej. LightMng, añade nuevas clases que representan los nuevos dispositivos, como por ejemplo LightCtrl. Así mismo, extienden la clase Gateway con nuevos métodos, ej. Switch-Light. La característica SmartEnergyMng sobrescribe los métodos para cambiar la temperatura y abrir o cerrar las ventanas en función de optimizar el consumo de energía.

Los paquetes UML que representan características de grano grueso son combinados por medio de la relación UML *merge*. La cual, unifica los elementos del mismo tipo que concuerdan por el nombre. Los elementos internos de las características, tales como atributos y métodos, son añadidos al resultado de la unificación. Si existen elementos con mismo nombre, el proceso *merge* es aplicado recursivamente. Por ejemplo, las clases Gateway serán combinadas para obtener una versión unificada. Para aquellos métodos pertenecientes a diferentes paquetes y con mismo nombre, serán incluidos sólo la versión más específica en el árbol de jerarquía. En el caso de conflictos de composición, tales como dos atributos con mismo nombre y distinta visibilidad, ej. Uno es público en un paquete y el otro es privado en otro paquete, existen un

conjunto de reglas UML (Object Management Group, 2005) para solucionarlo. Lo cual, por motivos de brevedad hemos omitido.

Las variabilidades de grano fino, son organizadas empleando otras técnicas, tales como parametrización. Por ejemplo, opciones variables en las unidades de medida de temperatura es conseguido estableciendo apropiadamente un atributo en las clases HeaterCtrl y ThermometherCtrl.

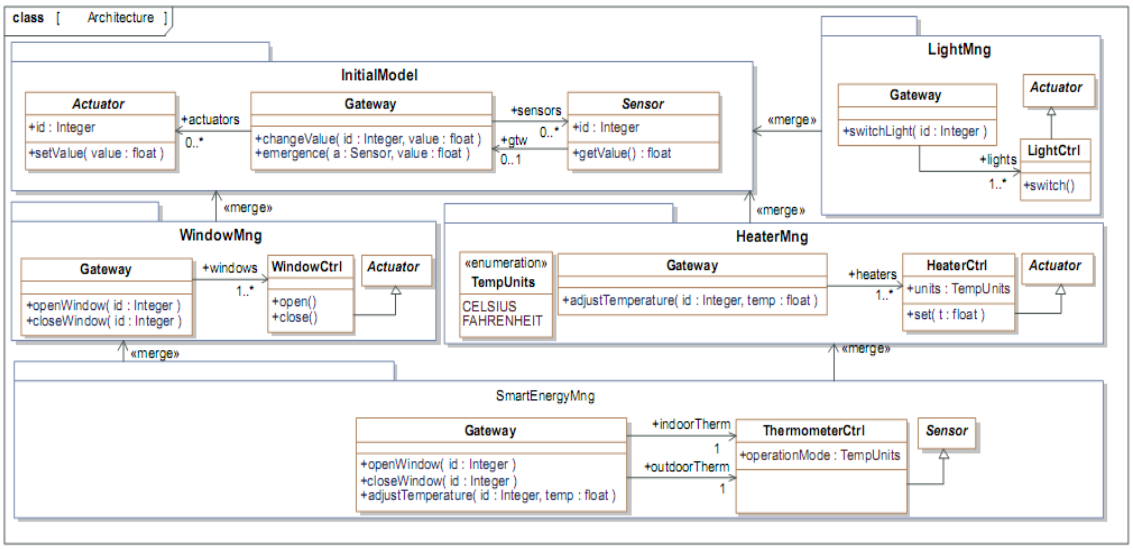

Fig. 2. Modelo de Componentes y clases resumido del hogar inteligente.

2.3 Programación Orientada a Características

La Programación Orientada a Características (FOP, Feature-Oriented Programming) (Prehofer, 1997) tiene como objetivo encapsular conjuntos coherentes de la funcionalidad de una aplicación informática en módulos bien definidos, independientes y componibles. Dichos módulos reciben el nombre de característica. Una característica se define habitualmente como un incremento, con un propósito común, en la funcionalidad de un sistema. Una vez descompuesto un sistema en características fácilmente componibles, debería ser posible obtener diferentes versiones del sistema mediante la inclusión/exclusión por composición de determinadas características.

Principalmente, el concepto comprende en implementar características en segmentos de clases, dividiendo el código de éstas en varios fragmentos de clases. Donde cada segmento contiene un incremento de la funcionalidad en comparación con los otros. Los cuales, pueden ser referenciados o combinados entre ellos obteniendo un producto concreto dotado de un conjunto de distintas funcionalidades.

La programación orientada a características aparece estrechamente ligada al concepto de líneas de productos software dado que la primera es un excelente mecanismo para implementar las segundas. Usando programación orientada a características, muchas de las variabilidades existentes en una línea de productos software pueden implementarse como

características. A la hora de construir un producto específico dentro de una línea de productos software, bastaría con seleccionar las características correspondientes a las variabilidades seleccionadas y componerlas. No obstante, no todas las variabilidades de una línea de producto software se implementan adecuadamente mediante el uso de características. En ciertas ocasiones, como variaciones que suponen incrementos muy pequeños de funcionalidad (ej. variar el color de una interfaz gráfica), resultan más adecuadas otras técnicas, como parametrización o la herencia tradicional de la programación orientada a objetos. Para mayor información remitimos al lector interesado al informe técnico elaborado a tal efecto dentro del proyecto AMPLE (AMPLE D2.2).

La programación orientada a características puede ser vista como una metodología modular de programación que sostiene que la mejor manera de minimizar la redundancia y mejorar la eficiencia, es crear un cierto número de características menores. Este paradigma, se centra en las funcionalidades del sistema, en lugar de los objetos con lo componen (a diferencia de la programación orientada a objeto). Las funciones deben ser vistas como una herramienta específica que ayudan completar una tarea general (Lopez-Herrejon, 2005).

Las siguientes secciones describen brevemente las limitaciones de los lenguajes orientados a objetos respecto a la implementación de características y los lenguajes de programación orientados a características existentes en la actualidad y de mayor relevancia dentro de la comunidad investigadora, haciendo especial hincapié en cómo cada lenguaje resuelve las limitaciones de la orientación a objetos respecto a la implementación de características.

2.4 Limitaciones de la Orientación a Objetos respecto a la implementación de características.

En programación orientada a objetos, para añadir nuevas funcionalidades a las clases ya existentes, comúnmente se realiza por medio de la herencia y polimorfismo. Por ejemplo, para poder añadir métodos requeridos por la característica SmartEnergyMng, es necesario crear una nueva subclase llamada Gateway-HeaterMng que extienda de la clase Gateway-InitialModel.

Como es bien conocido, la relación de herencia entre clases en programación orientada a objetos, es una forma de especialización donde la clase hija es un subtipo de la clase padre. Mediante herencia, podemos añadir incrementos de funcionalidad a una clase. Luego, dependiendo de las características seleccionadas por los usuarios por los usuarios se instanciaran las subclases adecuadas. No obstante, la herencia es jerárquica, lo que hace que la clase hija herede toda la funcionalidad de las clases padres. Por tanto, si una clase padre fuese extendida con tres características alternativas y una opcional, tendríamos problemas a la hora de implementar este esquema usando herencia. Las tres características alternativas se diseñarían como tres subclases de la clase padre. La característica opcional se diseñaría como otra subclase. Si ahora quisiésemos tener una aplicación con una alternativa y la característica opcional, tendríamos que crear una subclase de la alternativa y la opcional que heredase de ambas. En ausencia de herencia múltiple, este esquema no sería factible

Una clara insuficiencia de esta técnica es que, un incremento de funcionalidad representado por un conjunto de nuevas subclases que heredan de un conjunto de clases superiores, no es

posible manejarlo consistentemente como una unidad encapsulada. La insuficiente unidad de encapsulación, deriva en un problema de complejidad en el manejo de la selección de características para la composición. Aún cuando, las clases separadas en paquetes pertenezcan a una misma característica, es necesario seleccionar clases concretas que van a ser usadas en un producto específico. Por ejemplo, para incluir la característica HeaterMng en un producto específico, sería necesario seleccionar concretamente las clases Gateway y HeaterCtrl.

Otro problema es el manejo de las dependencias, debido a la herencia tradicional, las clases y sus subclases deben tener diferentes nombres. Por lo tanto, las referencias a las clases concretas deben ser actualizadas. A mayor número de características en una línea de productos, las relaciones entre clases concretas se complican exponencialmente, lo que resulta una situación indeseable. Además, en el caso que el lenguaje no soporte herencia múltiple, aumenta la complejidad del problema, puesto que extender de varias clases al mismo tiempo no resulta algo trivial. Por ejemplo, en la característica SmartEnergyMng no sería permitido una clase Gateway. Tendríamos que crear dos clases concretas, una que herede de Gateway-WindowMng, otra que herede de Gateway-HeaterMng y combinar ambas en otra clase que las contenga. Esto incrementa la complejidad en las relaciones y dependencias entre clases.

En pocas palabras, una estructura representada por herencia de clases es una limitante en la implementación de líneas de productos software. Puesto que, es considerado como obstáculo para la composición arbitraria de productos a partir de bloques predefinidos.

2.5 Lenguajes de Programación Orientados a Características

Esta sección describe brevemente los lenguajes de programación orientados a características más significativos dentro de la comunidad.

2.5.1 Scala.

Scala (Odersky et al, 2004) es un lenguaje de programación de propósito general multi-paradigma, que fusiona programación funcional y orientada a objetos. Diseñado para expresar patrones comunes de programación, reduciendo el tamaño del código en comparación con lenguajes como Java. Scala es interoperable con conocidas plataformas como el entorno de ejecución Java (JRM), así como con el framework de .Net.

Sus aspectos funcionales hacen que sea muy expresivo al escribir el código. Por ser orientado a objetos, los tipos y el comportamiento son descritos por medio de clases, permite la agregación, parametrización, herencia, etc.

Scala incluye rasgos (*traits* en inglés) en sus tipos de objeto. Un *rasgo* es una forma de abstracción de clases especial, el cual no utiliza ningún parámetro de valor en el constructor. Los rasgos, guardan similitud a las interfaces (ej. interfaces en Java), ya que son usados para definir tipos la especificación de la firma de métodos admitidos. Pero a su vez, los rasgos de Scala pueden ser implementados, por ejemplo, es posible incluir comportamiento básico para algunos métodos declarados en un rasgo.

Por lo tanto, los rasgos se convierten en un conjunto de métodos, completamente independientes de cualquier jerarquía de clases. En las composiciones de rasgos con tipos de sistemas como el de Java, el enfoque de herencia de clases junto a los rasgos coexisten como principal elemento de reutilización y diseño.

Para soportar programación orientada a características, Scala mezcla conceptos de orientación a objetos, tales como *composición lineal mixta* y *programación basada en rasgos*. Cuyo objetivo es potenciar la reutilización del software y servir de alternativa a la herencia múltiple. La implementación de *rasgos* en Scala puede contener clases anidadas y a su vez otros rasgos. Un rasgo en sí, entonces es la representación de una característica o funcionalidad de grano grueso. Por ejemplo, para representar a nivel de código la característica base InitialModel del caso estudio automatización del hogar, es posible implementar la clase Gateway definida, con sus miembros internos (clases y métodos) que cumplen con el comportamiento esperado. En la Figura 3, se puede observar la definición de la característica InitialModel que sirve como base para las demás funciones, como LightMng ó HeaterMng. En la línea 2, es declarada la clase Gateway_InitialModel; la cual contiene la definición de los métodos *emergence* y *chageValue* con sus respectivos parámetros y tipos (líneas 05 y 06). En la línea 08 es declarada una clase interna (anidada) de la clase Gateway.

```
00 package epl;
01 trait InitialModel {
02 class Gateway_InitialModel {
03 def actuators: List[Actuator]
04 …
05 def emergence(s: Sensor, value: Double): void = {...}
06 def changeValue(id: Int, value: Double): Boolean = {...}
07
08 class Actuator {...}
09 }
10 }
```

Fig. 3. Implementación de la característica InitialModel en Scala.

Para extender funcionalidades, con Scala es posible sobrescribir implementaciones básicas de miembros pertenecientes a rasgos, para aprovechar las capacidades disponibles en rasgos o clases más específicas. Para evitar incurrir en conflictos de sobrescritura, Scala requiere incluir el modificador *override* en la declaración de los métodos que sobrescriben a otros. Así mismo, Scala proporciona un mecanismo de *composición mixta*, que permite a los programadores reusar las definiciones de otras clases, sin necesidad de heredarlas. Este mecanismo es implementado con el identificador *With*. Por ejemplo, en el siguiente fragmento de código (ver Figura 4) comprendido de la implementación de la característica SmartEnergyMng, es posible combinar el rasgo WindowMng con HeaterMng. El rasgo SmartEnergyMng queda constituido a partir de la composición mixta, donde WindowMng es un “super-rasgo” y HeaterMng es un mixin. Lo mismo ocurre a nivel de clases, para que el refinamiento se cumpla en ambos casos, cada clase deberá especificar de cuál clase extiende y que miembro combina por medio *with* (línea 02). Al indicar esto, tanto Gateway_WinMng como Gateway_HtrMng pasan a formar parte de los subtipos de Gateway_SmartEng.

```
00 package epl;
01 trait SmartEnergyMng extends WindowMng with HeaterMng {
02 class Gateway_SmartEng extends Gateway_WinMng with Gateway_HtrMng {
03   override def openWindow(id : Integer) {...}
04   override def adjustTemperature(id : Integer, temp : float) {...}
05 }
06 }
```

Fig. 4. Extensión de Características y Combinación de clases en Scala.

```
00 package epl;
01 class Gateway1 extends Gateway_InitialModel
02    with Gateway_WindowMng{...}
03
04 class Gateway2 extends Gateway1 with Gateway_HeaterMng{...}
05
06 class Gateway3 extends Gateway2 with Gateway_SmartEnergyMng{
07    class openWindow extends super.openWindow
08          with super[WindowMng].openWindow;
09
10    class adjustTemperature extends super.adjustTemperature
11          with super[HeaterMng].adjustTemperature;
12 }

-----------------------------------------------------------------


13 object Gateway extends Gateway3 {
14    class openWindow extends super.openWindow { }
15    class adjustTemperature extends super.adjustTemperature { }
16
17 def main(args: Array[String]) : unit = {
18    openWindow = new openWindow();
19    adjustTemperature = new adjustTemperature ();
20    openWindow.run();
21    adjustTemperature.run();
22 }
23 }
```

Fig. 5. Composición de clases combinadas en Scala.

En la composición de la aplicación en sí una nueva clase Object es creada para ejecutar el programa (línea 13 - Figura 5), de forma que sea posible indicar el orden en el cual las extensiones deben ser compuestas. En la Figura 5, esto es representado desde la línea 00 a la 12 como una linealización del orden que seguirá el programa para al final ejecutar los métodos openWindow y adjustTemperature (línea 20 y 21). De esta manera, los métodos más específicos sobrescriben a los generales según las composiciones de clases que sea establecida. Para mayor información acerca de las sintaxis y demás aspectos técnicos puede dirigirse a la especificación del lenguaje[2].

En Scala, el eficiente modelado de la línea de productos software y su representación a nivel de código asociada a las capacidades de combinación y refinamiento de rasgos y clases, permite la composición de funcionalidades de distinta granularidad sin introducir mecanismos de

[2] http://www.scala-lang.org

agrupación de módulos y clases basados en alguna jerarquía. Por tanto, es posible utilizar el mismo rasgo en varias combinaciones.

2.5.2 CaesarJ

CaesarJ (Aracic et al, 2006) es un lenguaje orientado a aspectos que integra conceptos como clases y paquetes en una única entidad, llamada familia de clases. Lo cual, facilita la resolución de distintos problemas presentes en la construcción de software orientado a componentes. Una de las fortalezas de CaesarJ es la reusabilidad, ya que combina elementos de programación orientada a aspectos (como *pointcuts*, *advice*) y conceptos de modularización avanzados, como las clases virtuales, capacidades de composición y combinación. Lo cual, sustenta la capacidad de este lenguaje para soportar programación orientada a características.

Este lenguaje cuenta con la capacidad de extender funcionalidades del software por medio de refinamientos incrementales. Para abordar este aspecto desde lenguajes orientados a objetos, es necesario emplear técnicas clásicas como herencia y polimorfismo de métodos para cada refinamiento, lo que en algunos casos es insuficiente para resolver todos los escenarios del diseño a nivel de código. Puesto que, la herencia es empleada en el mismo nivel de clases. Lo cual, dado un incremento de funcionalidad representado por un conjunto de subclases que extienden a un conjunto de clases superiores, no podría se manejado consistentemente sin una unidad superior de encapsulamiento. Así mismo, esto representa un aumento en la complejidad para tratar eficientemente la selección de características. A mayor número de características, mayor es la complejidad en las relaciones entre clases concretas, dificultando el control de una línea de productos de mediano tamaño.

En este sentido, CaesarJ mejora el tipo de sistema de programación orientado a objetos, a través de dependencias entre familia de clases, las cuales a su vez, constituyen unidades de encapsulamiento adicional que agrupa clases relacionadas. Una familia de clases también es, en sí misma una clase. Así mismo, se introduce el concepto de clases virtuales, que son clases internas (de familias de clases) propensas a ser refinadas a nivel de subclases. En el refinamiento de una clase virtual, implícitamente se hereda de la clase que refina, por lo que también esto es visto como una relación adjunta. También, en un refinamiento pueden ser añadidos nuevos métodos, campos, relaciones de herencia y sobreescritura de métodos. Puesto que en cada familia de clases, las referencias a las clases virtuales siempre apuntarán al refinamiento más específico. Esto significa que, por medio de las clases virtuales se aplica sobre escritura de métodos, permitiendo redefinir el comportamiento de cualquier subclase de una familia de clases.

En términos de programación orientada a características, cada funcionalidad es modelada como una familia de clases. Mientras que los componentes y objetos del dominio específico, son correspondidos por sus clases virtuales. Así mismo, las clases virtuales pueden ser declaradas como clases abstractas, lo que habilita la definición de interfaces en la implementación modular de características. Como se muestra en la Figura 6, la característica InitialModel es definida como una familia de clases (línea 00 -02) con el identificador *cclass*, el cual representa directamente dicha funcionalidad. En la línea 04, se muestra la declaración de la clase Gateway, la cual es definida como perteneciente a la familia InitialModel. Así mismo, en

la línea 11-Figura 6 se define la clase virtual Actuator, la cual podrá ser refinada en los incrementos de funcionalidad subsiguientes.

```
00 package initialModel;
01
02 cclass InitialModel{
03 }

--------------------------------------------------

04 cclass initialModel.InitialModel;
05
06 public cclass Gateway{
07
08 public void emergence(Sensor s, double value){...}
09 public bool changeValue(int id, double value){...}
10
11 public cclass Actuator{...}
12 }
```

Fig. 6. Implementación de la característica InitialModel en CaesarJ.

Los incrementos de funcionalidad entre características, son modelados por una relación de herencia entre las características correspondientes. Así que, como las clases virtuales sobrescriben a nivel de subclases, una familia de clases puede extender y redefinir la funcionalidad de otra familia heredada (Gasiunas y Aracic, 2007). Es posible, que una característica requiera extender la funcionalidad de distintas característica a la vez. Por lo que, es importante contar alguna forma de herencia múltiple. En CaesarJ, la herencia de clases se soporta en un innovador mecanismo de composición por medio de la combinación de clases. El cual, permite heredar de más de una superclase a las familias de clases, siempre y cuando las superclases estén definidas con el mismo tipo. En este caso, la herencia múltiple, es manejada mediante un método especial de linealización del grafo de herencia (Ernst, 1999), de esta manera se resuelve la ambigüedad en el orden de los elementos a ejecutar sobreescritura de métodos heredados. En la línea 04 de la Figura 7, se puede observar el incremento de la característica SmartEnergyMng, la cual hereda de de WindowMng y HeaterMng simultáneamente, por medio del identificador "*&*". Para esta característica, ahora todas las clases virtuales que preserven el mismo nombre y tipo serán consideradas como un refinamiento, con miembros y comportamiento susceptibles a ser sobrescritos, como es el caso de la clase Sensor (línea 07).

```
00 package smartEnergyMng;
01 import HeaterManagement.*;
02 import WindowManagement.*;
03
04 cclass SmartEnergyMng extends WindowMng
05                       & HeaterMng{
06
07 public cclass Sensor{...}
08 }
```

Fig. 7. Extensión de características en CaesarJ.

CaesarJ maneja propagación automática de las dependencias a nivel de clases virtuales. Los nombres de las clases son preservados en cada refinamiento realizado por una familia de clases, lo que permite re-vincular todas las referencias de una clase automáticamente a la clase más específica que redefine. En otras palabras, las declaraciones de las clases virtuales heredadas (y superclases) con el mismo nombre, son referenciadas recursivamente por medio de este mecanismo automático. Y puesto que, las clases son entonces agrupadas por funcionalidad, es posible componer un producto específico con solo instanciar las familias de clases correspondientes a la selección de características.

En conclusión, en CaesarJ los incrementos arquitecturales de grano grueso, pueden ser encapsulados en familias de clases, permitiendo la separación de características reusables a nivel de código y un desarrollo aspectual de componentes a gran escala. Así mismo, la propagación de la composición permite que una funcionalidad sea extendida a partir de varias funcionalidades al mismo tiempo. Esto es soportado principalmente por la capacidad polimórfica de las clases virtuales (Aracic et al, 2005; Gasiunas y Aracic, 2007).

2.5.3 FeatureC++

FeatureC++ (Apel y Rosenmüller, 2008) es un lenguaje de programación moderno que extiende a C++, diseñado para soportar programación basada en características y orientación a aspectos. El cual, ofrece soluciones a problemas de extensibilidad mediante expresiones de refinamientos y manejo de herencia múltiple para tratar relaciones de variabilidad en líneas de productos software.

En esta extensión de C++, las clases son empleadas como la principal unidad de implementación. Una característica es implementada por la colaboración de múltiples clases descompuestas en función de las características del software.

Para implementar características, FeatureC++ emplea la combinación de capas llamadas *mixin layers*. Un *mixin layer*, es un componente estático que encapsula implementación de fragmentos de diferentes clases (conocidos también como *mixins*), el cual garantiza consistencia en las composiciones. Las *mixin layers*, representan una técnica de implementación para diseños basados en componentes con alto grado de modularidad y que ofrece prestaciones para la composición de líneas de productos software. Por tanto, es común que una fracción de clase pertenezca a la implementación de una característica y el resto corresponda a otras características. Por ejemplo, sería posible incluir comportamiento al método adjustTemperature en la clase Gateway de la característica HeaterCtrlMng desde la implementación de la característica SmartEnergyMng.

En FeatureC++, varios mixins pueden constituir una clase, lo cual se conoce como una cadena de refinamiento, las cuales a su vez, pueden ser conectadas entre sí. Los mixins que inician cadenas de refinamiento son llamados constantes y todos los demás son conocidos como "*refinamientos*". Cada constante y refinamiento es implementado como una clase dentro de un mismo fichero de código fuente (*.fcc*). El cual, puede abarcar refinamientos aplicados sobre características constantes (figura 8, línea 01) o sobre otros refinamientos, por medio de la

declaración de la palabra clave *refines* (línea 06). Así mismo, es posible introducir nuevos atributos (línea 07) y métodos (línea 08).

```
01 class Gateway{
02 bool adjustTemperature(int id, float temp){...}
03 ...
04 };
05
06 refines class Gateway {
07    float temp;
08    const float MAX_TEMP = 35.5;
09
10    bool openWindow(int id) {...}
11
12    bool adjustTemperature(int id, float temp){
13          if (strTemp(temp) + temp < MAX_TEMP)
14          super::adjustTemperature(temp); }
15 };
```

Fig. 8. Implementación de Refinamiento para la Clase Gateway en FeatureC++.

Los refinamientos pueden sobrescribir métodos de sus clases superiores, como lo muestra la Figura 8, en la línea 12. Para acceder al método padre se emplea la palabra clave *super* (línea 14), similar a la sintaxis de Java con respecto al uso de super. Generalmente, más que sobrescribir, lo métodos refinadores re-usan el método padre. Así mismo, dado que la herencia múltiple es un concepto bastante útil para expresar refinamientos en lenguajes orientado a objetos, FeatureC++ explota este y otros conceptos de C++ como la programación genérica, en particular plantillas de clases y métodos. Este lenguaje, resuelve el problema de extensibilidad, expresando las extensiones como refinamientos (como muestra la Figura 8) y herencia tradicional para las variaciones. Por ejemplo, la Figura 9 muestra en la línea 08, la implementación de un refinamiento de Actuator. Así mismo, HeaterCtrl y LigthCtrl heredarán (03 y 06 respectivamente) de la clase Actuator mas especializada, independientemente de sus posiciones en la cadena de refinamientos. Esta propiedad facilita la implementación modular de extensiones de clases.

```
01 class Actuator{};
02
03 class HeaterCtrl: Actuator{
04 void HeaterCtrl::Actuator(int id){...}
05 };
06 class LigthCtrl : Actuator{};
07
08 refines class Actuator{...};
```

Fig. 9. Implementación de Variaciones

Otra de las dificultades que soluciona FeatureC++, es el problema de la propagación de los constructores (Apel et al, 2005). En lenguajes convencionales orientados a objetos, los constructores no se heredan de forma automática y tienen que ser redefinidos para cada subclase. Para lo cual, FeatureC++ dispone de propagación de constructores padres a sus respectivas subclases. De esta manera, todos los constructores definidos en una cadena de refinamiento quedarán disponibles en la clase resultante generada. Por ejemplo, como lo ilustra la Figura 9, en la línea 05 el método constructor es generado automáticamente y será el que la aplicación final tome en cuenta en primera instancia (este ejemplo ilustra las

representaciones del lenguaje, no es exactamente el extracto de un programa ejecutable). Además de constructores, también soporta propagación de sobrecarga de métodos para el refinamiento de capas respectivamente.

Adicionalmente lenguaje de programación introduce conceptos de programación orientada a aspectos como los *Aspectual Mixin*, mediante los cuales es posible implementar avanzados, dinámicos y transversales refinamientos que potencian aún más las capacidades de reutilización y modularidad del lenguaje. Básicamente, se trata de aplicar capacidades de orientación a aspectos a los *mixins*. En este enfoque, además de que los *mixins* refinan otros de su misma especie, también integran *pointcuts* y *advices* directamente. En este sentido, *Aspectual Mixin* guardan semejanzas con los llamados *Classpects*, los cuales son entidades que unifican conceptos de lenguajes orientadas a aspectos y a objetos. En este trabajo no se profundizará en las capacidades orientadas a aspectos de FeatureC++, puesto que el enfoque del mismo es hacia los lenguajes de programación orientados a características.

2.5. 4 ObjectTeams

ObjectTeams (Herrmann, 2005) es un lenguaje de programación moderno que va más allá de conceptos tradicionales de programación orientada a objetos, en función de proporcionar mejoras en la modularización del software. Aunque el modelo de ObjectTeams no fue diseñado para construir aplicaciones orientadas a características y líneas de productos, otros autores afirman que éste luce muy apropiado para ello, soportado por la combinación de paradigmas de programación como software orientado a aspectos y roles.

El modelo de ObjectTeams agrupa propiedades aspectuales e introduce un nuevo concepto, el "*team*". El cual, representa un paquete de clases agrupadas, que a su vez son llamadas "*roles*", similares a las clases internas de Java. Los *roles* y *teams*, encapsulan comportamiento (métodos) que puede ser adaptado para obtener la funcionalidad esperada.

Como se detalla en (Hund et al, 2007), cada funcionalidad de grano grueso (o característica en términos de programación orientada a característica) es representada por un *team*. Por ejemplo, el *team* declarado en la línea 01 de la Figura 10, representa directamente a la característica InitialModel. Así, cualquier clase interna de InitialModel, define una clase *rol* sin necesidad modificador adicional.

```
01 public team InitialModel {
02    ...
03 public class Gateway() {...}
04 }
05
06 public team InitialModel {
07    public team HeaterMng {
08          public class Gateway() {...}
09 } }
```

Fig. 10. Definición de la Característica InitialModel con ObjectTeams.

Un *team* también es una clase, por lo que soporta refinamientos de conjuntos de clases por medio de herencia y polimorfismo. Es decir, un *team* que encapsula características, como por ejemplo InitialModel y HeaterMng, es posible vincular a la segunda como una subclase que extiende de InitialModel. A su vez, ObtjectTeams soporta anidación de *teams*, como lo muestra la línea 07 de la Figura 10, la característica HeaterMng es incluida dentro de la característica InitialModel. Por lo que, HeaterMng tendrá propiedades híbridas por ser un *team* y una clase rol a la vez.

Para incrementar funcionalidades, ObjectTeams cuenta con mecanismos de sustitución como lo es la herencia entre clases *roles*. Por medio del cual, un *team* puede incorporar el comportamiento de los roles de un *team* superior relacionando sus clases internas (roles). Dicho mecanismo permite un tipo de herencia implícita asociado a los nombres de los roles. Es decir, si un *team* contiene la definición de un rol con el mismo nombre que otro rol declarado en un *team* superior, la nueva clase rol sobrescribe al rol del *team* padre y hereda implícitamente todas sus propiedades. Esta relación es establecida solamente por medio de la correspondencia de nombres. Un ejemplo de lo anterior descrito está representado en la Figura 11, donde la clase Gateway de la característica HeaterMng (línea 10) sobrescribe y hereda el comportamiento del rol que tiene el mismo nombre en la característica InitialModel (línea 02), puesto que en la declaración del *team* HeaterMng especifica la relación *extends* (línea 07). En las construcciones de herencia implícita, a diferencia de la herencia regular los constructores también son heredados implícitamente y pueden ser sobrescritos como otro cualquier método. Opcionalmente, es posible implementar herencia tradicional utilizando el identificador *@override* en la declaración de un rol, como se señala en la línea 15 el rol Gateway de la característica SmartEnergyMng sobrescribe al de la característica HeaterMng. En ambos tipos de herencia, es permitido realizar llamadas al constructor del rol padre por medio del identificador *tsuper*, como se indica en la línea 18 tsuper.adjutTemperature invoca a la versión de adjutTemperature adquirida del *team* superior. Paralelamente, ObjectTeams cuenta con un mecanismo de extensión por adición, a través del cual es posible la añadidura directa de roles y otros miembros (atributos, métodos, etc.) contenidos en un *team* al extender de éste. Como es el caso, del atributo declarado en la característica HeaterMng (línea 08) y se encuentra disponible en la característica SmartEnergyMng (línea 17) sin necesidad de alguna definición adicional.

```
00  public team class InitialModel{
01
02     class Gateway {
03      changeValue(int id, float value) {...}
04      emergence(Sensor a, float value) {...}
05     }
06  }
-------------------------------------------------------
07 public team class HeaterMng extends InitialModel {
08 const double MAX_TEMP = 35.5;
09
10  class Gateway {
11    adjustTemperature(int id, float temp) {...}
12  }
13 }
```

Fig. 11. Extensión de Características con ObjectTeams.

```
14 public team class SmartEnergyMng extends HeaterMng{
15    @Override class Gateway {
16          adjustTemperature(int id, float temp) {
17                if(temp > MAX_TEMP){
18                return tsuper.adjustTemperature(id, temp);}
19     }
20 } }
-------------------------------------------------------------
21 public team class WindowMng {
22     public class WindowCtrl playedBy Actuator {
23 ... }
```

Fig. 11 (Continuación). Extensión de Características con ObjectTeams.

Por otro lado, las clases roles de ObjectTeams pueden relacionarse con otras clases ajenas al *team* que lo contiene (conocidas como clases *base*) mediante el identificador *playedBy* en la declaración del rol. Dicho identificador define que el rol es asociado a la instancia de una clase base (Herrmann, 2005). En el ejemplo de la Figura 11 (líneas 21-23), la clase rol WindowCtrl de la característica WindowMng es asociado a la clase base Actuator perteneciente a la característica InitialModel:

A partir de la declaración *playedBy*, un rol puede heredar y sobrescribir funcionalidades de sus clases base utilizando el mecanismo *callin* y *callout*. De esta manera, un rol puede comunicarse con los objetos de clases base. En *callin*, las llamadas al objeto base son interceptadas, a través de métodos de enlace que se pueden especificar en la clase rol, con el fin de ejecutar uno de sus métodos. Por ejemplo, definiendo una llamada al método original (open) es interceptada y re-direccionada al método del rol (openWindow). Cumpliendo con el mismo resultado que la sobrescritura y herencia tradicional:

```
void openWindow() <- replace void open();
```

De igual manera, al utilizar *callout*, la clase base implementa open(), el cual es heredado por la clase rol. Así que, debido a la declaración *callout*, al recibir una llamada al método windowOpen(), automáticamente será re-dirigida a la instancia base asociada, invocando el método open():

```
void openWindow () -> void open ();
```

Cabe destacar que, por medio de estos mecanismos (*callin* y *callout*) las funcionalidades de grano fino encuentran su correspondencia directa en la implementación. Así mismo, estas propiedades incrementan la escalabilidad del lenguaje, ya que por ejemplo la clase base a la cual se relaciona un rol, puede ser en sí misma un *team* o incluso una clase rol de otro *team*. Es importante mencionar que, la forma como trabajan estos mecanismos es similar a la herencia clásica salvo importantes diferencias:

- El mecanismo *playedby* del rol es dinámico, ya que sucede entre instancias en tiempo de ejecución. Así mismo, separa en dos partes la herencia: la adquisición y sobrescritura.
- La granularidad es fina, puesto que lo métodos y campos individuales, pueden ser adquiridos y sobrescritos selectivamente. Por lo que, un role puede adquirir métodos individuales de su base utilizando *callout*.

En base a lo anterior, ObjectTeams proporciona un nivel de composición a nivel de características. La selección de características para la composición de una aplicación concreta en una línea de productos software es realizada dinámicamente por medio de la instanciación (vía configuración) y activación de *teams.* La activación de *teams* es ampliamente documentada en la especificación de ObjectTeams[3]. Esta tipo de selección para la composición de funciones de grano grueso, aplica si en la implementación las características son representadas cada una en único *team.*

Para asegurar composiciones coherentes, el lenguaje contempla restricciones que aseguran dinámicamente la selección de un característica requiere implícitamente la inclusión de otras para poder operar correctamente. Por ejemplo, para la correcta instanciación de la característica SmartEnergyMng de nuestro caso estudio Hogar Inteligente, antes deben estar instanciados los *teams* WindowMng y HeaterMng.

En la siguiente sección se detalla un resumen de las propiedades de los lenguajes de programación orientados a características aquí descritos.

2.5.5 Resumen

Criterios	Lenguajes Orientados a Características			
	Scala	**CaesarJ**	**FeatureC++**	**ObjectTeams**
Encapsulación de características	Los rasgos y clases encapsulan comportamiento de distinta granularidad. A su vez, el comportamiento común puede ser factorizado en un rasgo más complejo.	Familias de clases que representan directamente una característica contentiva de clases internas, métodos, atributos, etc. de la SPL.	Definido como capas a nivel de características que encapsulan fragmentos de diferentes clases.	Una característica es representada por un *team.* El cual, es constituido por conjuntos de clases llamadas *roles.*
Extensión de características	Sobre escritura de métodos genéricos por métodos concretos contenidos en los rasgos, sin necesidad de herencia. Extensión por adición con la inclusión de nuevos rasgos.	Refinamiento por medio de clases internas (virtuales) implícitamente por medio de las dependencias entre clases superiores que las contienen.	Expresiones de refinamientos como "*refines*" y *constantes*, así como, manejo de herencia múltiple para tratar relaciones de variaciones en líneas de productos software.	Sobrescritura de métodos por herencia implícita. Mecanismos *callin* y *callout,* que relaciona clases *roles* y *base*. Extensión por adición de *roles* dentro de un mismo *team.*

Tabla 1. Resumen lenguajes orientados a características.

[3] http://www.objectteams.org

Criterios	Lenguajes Orientados a Características			
	Scala	**CaesarJ**	**FeatureC++**	**ObjectTeams**
Manejo de dependencias	Este enfoque simplifica el este aspecto, ya que los rasgos se combinan sin necesidad de referenciar dependencias.	Actualización automática de la referencias en las clases virtuales según las extensiones de funcionalidad seleccionadas.	Propagación de constructores y sobrecarga de métodos hacia su respectivas sub-clases.	La activación de *teams*, representa una forma de gestionar las dependencias con cierta flexibilidad.
Composición de aplicaciones específicas	Operaciones que permiten combinar clases y rasgos, lo cual conforma un artefacto base de grano grueso, habilitando la selección de subconjuntos para obtener un producto concreto.	Instanciación de familias de clases según selección de características.	Composición de características por medio de la combinación de clases envolventes de fragmentos de sub-clases refinadoras, similar al concepto de patrones de diseño en POO.	Instanciación de *teams,* que incorporan roles, los cuales a su vez pueden figurar como variabilidades de grano fino según su nivel en el diseño arquitectural.
Verificación de composiciones	Es posible verificar estáticamente ambigüedades y conflictos de sobre escritura de métodos en la composición de clases.	El lenguaje genera la aplicación final según el orden y jerarquía de clases definida. Por lo que tácitamente valida la consistencia de cada selección de características incluidas en una composición.	Validación de la coherencia de configuraciones en tiempo de ejecución.	Restricciones que aseguran dinámicamente la instanciación coherente de *teams.*

Tabla 1 (Continuación). Resumen lenguajes orientados a características.

2.6 Clases Parciales C#

Las clases parciales C# (Albahari y Albahari, 2010) permiten dividir la implementación de una clase, estructura o interfaz en varios archivos de código fuente. Cada fragmento de clase 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 tanto, tal como han propuesto otros autores (Laguna et al, 2007; Warmer y Kleppe, 2006), las clases parciales C# pueden ser vistas como un mecanismo adecuada para implementar características, dado que cada incremento en funcionalidad para una clase se podría encapsular en una clase parcial separada.

Para poder ser compiladas y agrupadas en una sola clase, todas las clases parciales deben pertenecer al mismo espacio de nombres, poseer la misma visibilidad y deben ser declaradas

con el indicador clave partial. En C#, un espacio de nombre es simplemente empleado para agrupar clases relacionadas y evitar conflictos de nombres. Para especificar los archivos C# que deben ser incluidos en una compilación, se emplea un documento XML que contiene información acerca del proyecto y que especifica que ficheros deben ser compilados generar el proyecto. Por lo tanto, es posible incluir y excluir fácilmente la funcionalidad encapsulada dentro de una clase parcial simplemente añadiendo o eliminando dicha clase parcial de este fichero XML (Figura 12).

```
00 </Project>
01 <ItemGroup>
02 <Compile Include="InitialModel\Gateway.cs" />
03 <Compile Include="LightMng\Gateway.cs" />
04 <!-- <Compile Include="WindowMng\Gateway.cs" />
05 <Compile Include="SmartEnergyMng\Gateway.cs" />
06 -->
07 ...
08 </ItemGroup>
09 </Project>
```

Fig. 12. Implementación del Archivo SmartHome.csproj.

El mecanismo de clases parciales es un medio de añadir o compartir funcionalidad entre un conjunto de clases que no precisan estar relacionadas mediante ningún tipo de relación jerárquica, tal como ocurre con la herencia. Por tanto, las clases parciales permiten componer código sin necesidad de dependencias entre clases.

En resumen, al usar clases parciales, es posible colocar funcionalidad común en una clase separada y mezclarla dentro de múltiples clases, sin obedecer a ninguna jerarquía de herencia. Por lo tanto, algunos autores (Laguna et al, 2007) sostienen que, las clases parciales C# representan una alternativa a la herencia múltiple para manejar variabilidad relacionada programación orientada a características. El objetivo de este trabajo es verificar dicha hipótesis mediante la aplicación de las clases parciales a un caso de estudio industrial.

3. Evaluación de las Clases Parciales de C# como mecanismo para orientación a características.

Esta sección analiza ventajas e inconvenientes de las clases parciales de C# como mecanismo para implementar diseños orientados a características tales como el presentado en la Figura 1. Para ello, primero definimos que propiedades debería tener un lenguaje que soportase orientación a característica y luego analizaremos como las clases parciales de C# soportan esas propiedades mediante su aplicación al caso de estudio de los hogares inteligentes descrito en la Sección 2.1.

3.1 Propiedades deseables en un Lenguaje Orientado a Características.

Esta sección describe qué propiedades son deseables encontrar en un lenguaje orientada a características. Como ha sido comentado con anterioridad, la Programación Orientada a Características (Prehofer, 1997) tiene como objetivo encapsular conjuntos coherentes de la

funcionalidad de una aplicación software en módulos bien definidos, independientes y componibles. Dichos módulos reciben el nombre de característica. Una vez descompuesto un sistema en características fácilmente componibles, debería ser posible obtener diferentes versiones del sistema mediante la inclusión/exclusión por composición de determinadas características.

No obstante, es importante clarificar que no todas las combinaciones de características al componerlas generan una aplicación final correcta. Por tanto, los lenguajes orientados a características deben tratar de asegurar que las composiciones de un conjunto de características siempre deriven en aplicaciones correctas. Por ejemplo, para el caso estudio que nos atañe, un lenguaje orientado característica debería asegurar que si incluimos la característica SmartMng, entonces también deberíamos incluir las características WindowMng y HeaterMng, tal como está especificado en el modelo de características (restricción C1, ver Figura 1).

Con estos objetivos en mente, y basándonos en (Lopez-Herrejon et al, 2005; Chul et al, 2002; Aracic et al, 2006; Chae y Blume, 2009) hemos identificado varias propiedades que sería deseable encontrar en un lenguaje de programación de cara a implementar descomposiciones orientadas a características. A continuación, comentamos cada una de estas propiedades.

Extensibilidad.

Una característica se define normalmente como “un incremento en la funcionalidad de un programa” (Zave, 1999; Lopez-Herrejon et al, 2005). Por ejemplo, en nuestro caso de estudio, LightMng, o WindowMng son incrementos de funcionalidad cuyo objetivo es supervisar y controlar nuevos dispositivos, tales como luces o ventanas. Por tanto, los lenguajes orientados a características deben proporcionar mecanismos para añadir nuevas funcionalidades a las funcionalidades ya existentes. En lenguajes orientados a objetos, tal como se ha comentado en la Sección 2.4, estas extensiones se realizan mediante herencia. La herencia puede resultar adecuada cuando su implementación requiere la extensión de una sola clase. Sin embargo, es frecuente el caso en el cual la implementación de una característica necesita extender varias clases al mismo tiempo. Por ejemplo, la característica de control automático luces (LightMng) requiere añadir una nueva clase para el control de los dispositivos de luz (LigthCtrl) y también necesitamos extender la clase que representa la puerta de enlace (Gateway) para añadirle nuevos los métodos necesarios para poder apagar y encender las luces de la casa.

Las extensiones no son siempre extensiones por incorporación de nuevas funcionalidades, en ciertas ocasiones es necesario sustituir, modificar o sobrescribir el comportamiento previamente definido en clases correspondientes a las características que se extiende. Por ejemplo, la característica SmartEnergyMng, debe sobrescribir la implementación del método adjustTemperature, de la versión de la clase Gateway correspondiente a la puerta de enlace, con objeto de realizar una serie de comprobaciones adicionales antes de ajustar definitivamente la temperatura. Por ejemplo, si se solicita enfriar una habitación cuyas ventanas están cerradas y la temperatura exterior es inferior a la interna, con objeto de ahorrar energía, se apagaran los radiadores si éstos estuviesen encendidos y se abrirán las ventanas.

Este tipo de extensión se realiza en lenguajes orientados a objetos mediante sobreescritura de métodos.

Encapsulación de Características.

Todas las extensiones pertenecientes a una misma característica deben ser añadidas de forma atómica. Es decir, o todas las extensiones son incorporadas a un producto al mismo tiempo, o ninguna de ellas debería ser incorporada en caso contrario. Por ejemplo, la clase Gateway debe ser extendida con el método switchLight si y sólo si la clase Light también se incorpora a la aplicación. Para ello, el lenguaje debe proporcionar mecanismos de agrupación y encapsulación, identificando correctamente el conjunto de elementos que pertenecen a una misma característica. Además, idealmente, estos conjuntos deberían poder ser compilados de forma separada e independiente. Por ejemplo, en la Figura 13, los paquetes UML se usan para agrupar elementos pertenecientes a una misma característica.

Las extensiones requeridas por una característica podrían tener una serie de efectos colaterales indeseados. Por ejemplo, en el paquete InitialModel de la Figura 13, la clase Sensor tiene una referencia a la clase Gateway. Esta referencia se usa para enviar mensajes a la clase Gateway para alterar de situaciones críticas, tales como la detección de fuego. Supongamos que las extensiones requeridas por cada característica se realizan por medio de herencia. En se caso la clase Gateway se refinaría a través de las diferentes características, creándose diferentes versiones de dicha clase, tales como LightMng::Gateway, HeaterMng::Gateway, WindowMng::Gateway y SmartEnergy-Mng::Gateway. De esta forma, dependiendo de las características seleccionadas que el cliente desee incluir en el producto final, tendríamos que instanciar una subclase determinada, o una combinación de ellas, dentro del producto final.

Supongamos ahora que se selecciona únicamente la característica LightMng. En este caso, la versión de la clase Gateway a incluir en el producto final sería LightMng::Gateway. No obstante, la referencia de la clase Sensor corresponderán a la clase Initial::Gateway, perteneciente la paquete InitialModel. Estas referencias deberían ser actualizadas, así como, los métodos constructores, getters, etc. que han uso de la misma para que, en lugar de a Initial::Gateway, referenciasen a la subclase LightMng::Gateway, evitando así problemas relacionados con el sistema de tipos, tales como tediosos castings. Normalmente, esta actualización debe ser realizada manualmente. Sin embargo, algunos lenguajes como CaesarJ (Aracic et al, 2006) u ObjectTeams (Herrmann, 2007) poseen avanzados sistemas de tipos que permiten realizar esta tarea automáticamente.

Por tanto, una gestión automática de las dependencias entre clases por parte del sistema de tipos, así como, de otros efectos colaterales resultantes de la extensión entre características, son también cualidades deseables en lenguajes orientados a características.

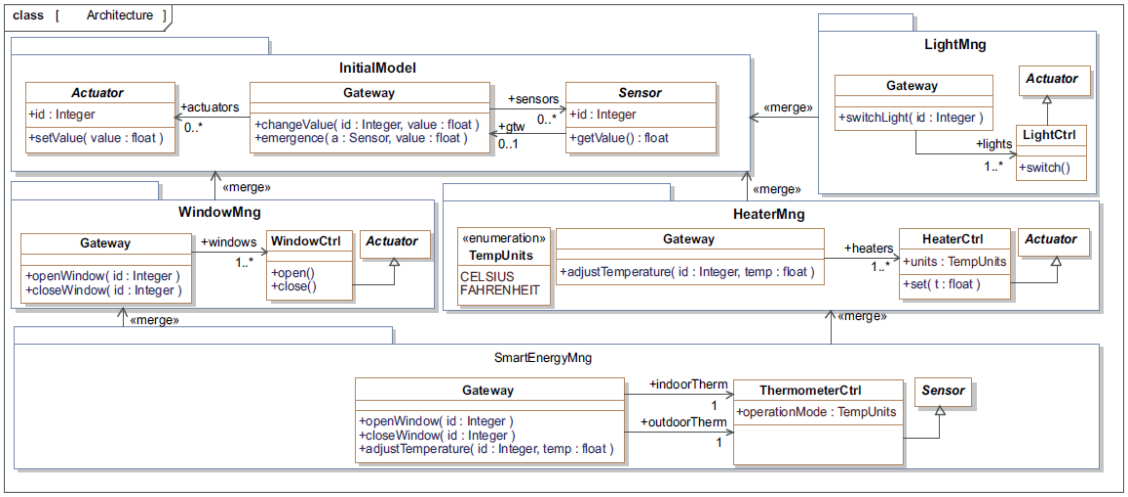

Fig. 13. Diseño arquitectural resumido de la automatización del hogar.

Composición a Nivel de Características.

Los productos software concretos dentro de una descomposición orientada a características se obtienen mediante la selección y composición de un conjunto de características concretas. Por tanto sería deseable que los lenguajes orientado a características proporcionasen construcciones adecuadas para especificar que características deben ser instanciadas o compuestas con el objetivo de generar un producto específico. Esta especificación debería realizarse a nivel de características, especificando el conjunto de características que deben ser incluidas, en lugar de tener que especificar los elementos individuales de cada característica que debe ser incorporados al producto final. Por ejemplo, conforme a la Figura 13, si la característica LightMng fuese seleccionada, lo ideal sería poder especificar simplemente que el paquete LightMng debe ser incluido en el producto final; en lugar de tener que especificar y añadir individualmente a el producto final las clases LightMng::Gateway y LightMng::LightCtrl, y otras más, si las hubiere.

Verificación de la Coherencia de la Composición.

Como hemos comentado con anterioridad, y como por otra parte es lógico, no todas las posibles combinaciones de características dan lugares a configuraciones correctas que permitan obtener productos finales que funcionen de forma correcta. Además, las características no pueden ser compuestas en cualquier orden, siendo sólo ciertas ordenaciones válidas. Por ejemplo, si la característica SmartEnergyMng fuese seleccionada, las características WindowMng y HeaterMng también deberían ser seleccionadas, ya que la primera no puede funcionar sin la presencia de las dos últimas. Por esta misma razón, WindowMng y HeaterMng deben ser compuestas con InitialModel antes que SmartEnergyMng, dado que esta última depende de las anteriores.

Un lenguaje orientado a característica debería tener en cuenta estas restricciones, a fin de evitar la creación de productos incorrectos. También sería deseable que el propio lenguaje se encargase de gestionar de forma automática las dependencias entre características. Por

ejemplo, si la característica SmartEnergyMng fuese seleccionada, el leguaje debería ser capaz de detectar que las características WindowMng y HeaterMng también deben ser incluidas e incluso compuestas antes que SmartEnergyMng. Debería ser el propio lenguaje el que realizase, de forma automática, la inclusión de las características WindowMng y HeaterMng y determinase el orden de composición.

Resumen.

Sintetizando, un lenguaje de programación orientado a características debería proporcionar idealmente:

- Extensión de características tanto por suma de nuevas funcionalidades como por sobrescritura.
- Modularización y encapsulamiento de elementos pertenecientes de las características.
- Gestión automática de dependencia entre clases.
- Composición a nivel de características.
- Verificación de la consistencia de la composición de características.
- Gestión automática de las restricciones entre características.

En la siguiente sección, analizamos si las clases parciales C# como mecanismo para implementar descomposiciones orientadas a características poseen esta serie de propiedades deseables.

3.2 Análisis de las clases parciales de C# como mecanismo para implementar descomposiciones orientadas a características.

Para analizar el potencial de las clases parciales de C# como mecanismo para implementar descomposiciones orientad a características, hemos tratado implementar el caso de estudio de los hogares inteligentes usando clases parciales de C# en la plataforma MS Visual Estudio .Net. El objetivo principal que perseguimos con esta implementación es incrementar la modularidad y facilidad de composición de las características del sistema. Las clases parciales toman el rol principal de módulo de encapsulación de características. No obstante, se han determinado ciertas limitaciones de este mecanismo, entre otras cosas, para representar extensiones por sustitución de características y sub-características en una línea de productos software.

Aunque las clases parciales proporcionan algunas ventajas para la modularización de características, el lenguaje que las soporta, es decir C#, es un lenguaje orientado a objetos. Por tanto, las opciones disponibles para implementar características son los mecanismos tradicionales de herencia y polimorfismo. El principal problema de esta técnica es que, dado un incremento de funcionalidad representado por un conjunto de clases y subclases, no es posible manejarlo explícitamente como una unidad encapsulada que facilite la composición del sistema.

En las siguientes sub-secciones se exponen los resultado aplicar clases parciales C#, a la implementación del diseño arquitectónico orientado a características del caso estudio de automatización del hogar.

La clase Gateway para la característica InitialModel (líneas 00-08, Figura 14) contiene colecciones de sensores y actuadores (líneas 02 y 03), así como los métodos emergence y changeValue (líneas 05 y 06), tal como especifica en el modelo de la Figura 13. La clase Gateway para la característica LightMng (líneas 09-16, Figura 14) extiende la clase Gateway de InitialModel con colecciones para la gestión de luces (línea 12) así como el método switchLight (línea 14).

Archivo InitialModel/Gateway.cs

```
00 namespace SmartHome {
01 public partial class Gateway {
02 protected List<Sensor> sensors;
03 protected List<Actuator> actuators;
04
05 public void emergence(Sensor s, double value)
{...}
06 public bool changeValue(int id, double value)
{...}
07 }
08 }
```

Archivo LightMng/Gateway.cs

```
09 namespace SmartHome
10 {
11 public partial class Gateway {
12 protected List<LightCtrl> lights;
13
14 public bool switchLight(int id) {...}
15 }
16 }
```

Archivo SmartHome.csproj

```
17 </Project>
18 ...
19 <ItemGroup>
20 <Compile Include="InitialModel\Gateway.cs" />
21 <Compile Include="LightMng\Gateway.cs" />
22 <!-- <Compile Include="WindowMng\Gateway.cs" />
23 <Compile Include="SmartEnergyMng\Gateway.cs" />
24 -->
25 ...
26 </ItemGroup>
```

Fig. 14. Implementación de Gateway usando clases parciales.

Las línea 17-26 de la Figura 14 muestra parte del fichero generado para este proyecto (SmartHome.csproj). Este archivo indica que las clases parciales Gateway de las características InitialModel y LightMng serán incluidas en el producto final compilado, pero las clases parciales correspondientes a las características WindowMng o SmartEnergyMng serán excluidas de la clase Gateway resultante. Por lo tanto, el compilador producirá una clase Gateway con la funcionalidad necesaria para la gestión de luces, pero sin gestión automática de ventanas, temperatura ni control inteligente de energía. Motivado a estos resultados, algunos autores

como (Laguna et al, 2007) proponen emplear clases parciales C# para implementar diseños orientados a características.

Las clases parciales son útiles para la separación de código y desarrollo modular de las funcionalidades. Puesto que, las clases parciales empleadas al estilo de los *mixins*, significan una colección de comportamientos fácilmente localizables o separables de las funciones básicas del programa. A diferencia de la herencia, que representa una forma de especialización, al incluir una clase parcial se introducen cualidades adicionales por medio de la definición de sus comportamientos. Por tanto, se ha empleado mas como una técnica que aumenta el grado de modularidad del software, pudiendo construir piezas de programa separadas combinables con otras.

Encapsulación de Características.

Las clases parciales de C# carecen de mecanismos para agrupar y encapsular características. Intentamos aplicar espacios de nombres para este propósito. La idea era colocar todas las clases relacionadas con una misma características dentro de un mismo espacio de nombres. De esta forma, para las características InitialModel y LightMng crearíamos los espacios de nombres llamados SmartHome.InitialModel y SmartHome.LightMng, respectivamente. Sin embargo, en este caso las clases parciales pertenecientes a diferentes características pero representando una misma clase no serian combinadas, al estar situadas en diferentes espacios de nombres. Así que, necesariamente todas las clases parciales deben estar situadas en un mismo espacio de nombre, por lo que no es posible emplear este mecanismo para agrupar el código perteneciente a una misma característica.

Una alternativa a esta situación, podría ser crear una clase para cada característica y anidar las clases pertenecientes a una misma característica dentro de esta clase de grano grueso. No obstante, esta solución tampoco funcionaría, ya que la clase exterior sirve como espacio de nombres para las clases internas. Y aunque cada clase interna sea declarada como clase parcial, al estar colocadas en espacios de nombres diferentes sería imposible su composición.

Un problema derivado de la no existencia de mecanismos para agrupar las clases pertenecientes a una característica es la falta de mecanismos de encapsulación de características. Las clases y métodos pertenecientes a una característica, al no poder ser identificados como pertenecientes a una característica no pueden ser ocultados a otros miembros de características independientes. Por ejemplo, la clase LightCtrl no debería ser visible para la clase HeaterCtrl. Sin embargo, con las facilidades proporcionadas actualmente por las clases parciales de C# es inevitable que la clase HeaterCtrl pueda utilizar la clase LightCtrl.

Una forma de evitar esto, es, cuando se está implementando una clase perteneciente a una cierta característica, excluir las clases parciales correspondientes a las características independientes del fichero de configuración del proyecto, y recompilar el mismo. En este caso, el compilador producirá un error si se trata de utilizar clases pertenecientes a características independientes. Por ejemplo, si se incluye solamente la característica LightMng en el fichero de configuración del proyecto y se excluyen las clases parciales correspondientes a las

características WindowMng, HeaterMng y SmartEnergyMng, al intentar utilizar la clase HeaterCtrl desde la clase LightCtrl, el compilador alertará que está clase no se encuentra definida. No obstante, incluir y excluir ficheros manualmente del archivo de configuración del proyecto y recompilar cada vez que se modifique una clase o alguna clase parcial perteneciente a una cierta característica resulta una tarea tediosa y propensa a errores, por lo tanto no se considera ésta una solución adecuada.

Extensibilidad de Características.

En cuanto a la extensibilidad, las clases parciales permiten añadir nuevos atributos y métodos a las clases parciales existentes. Sin embargo, no permiten extender o sobrescribir métodos existentes.

De esta forma podemos, por ejemplo, tal como especifica el modelo de la Figura 10, añadir un nuevo atributo luces a la clase Gateway de la característica LightMng. Así mismo, deberíamos extender apropiadamente el constructor de la clase Gateway para inicializar dicho atributo. Para ello, intentamos escribir el código representado en la siguiente Figura (Figura 15), la cual contiene el constructor para la clase Gateway perteneciente al paquete InitialModel (líneas 03-06, Figura 15). Este constructor es extendido con nueva funcionalidad en la clase parcial Gateway de la característica LightMng (líneas 10-12, Figura 15).

Archivo InitialModel/Gateway.cs:

```
01 public partial class Gateway {
02 ...

03 public Gateway() {
04   this.actuators = new List<Actuator>();
05   this.sensors = new List<Sensor>();
06 }
07 }
```

Archivo LightMng/Gateway.cs:

```
08 public partial class Gateway {
09 protected List<LightCtrl> lights;
10 public Gateway() {
11   this.lights = new List<LightCtrl>();
12 }
13 }
```

Fig. 15. Constructor Gateway utilizando clases parciales.

Sin embargo, el esquema mostrado en la Figura 13 no es posible ya que, el compilador genera un error avisando de que el constructor de la clase Gateway se encuentra duplicado. Esto significa que, no es posible dividir la implementación de un método en varios archivos, puesto que no está permitido tener métodos de una clase parcial con la misma cabecera en dos ficheros separados. Esto representa una importante limitación respecto a la extensibilidad ya que no podemos extender métodos ya existentes. En este sentido, podría ser interesante si C#, además de clases parciales, soportara métodos parciales.

En el caso de la extensibilidad por sobrescritura, el problema es el mismo que el anteriormente descrito. En este caso usaremos como ejemplo la característica SmartEnergyMng. La clase Gateway, entre otros, debe sobrescribir el método adjustTemperature para verificar si hay que ejecutar operaciones adicionales sobre las ventanas antes de ajustar la temperatura de los radiadores. No obstante, debido a que no está permitido declarar un método adjustTemperature en la clase parcial Gateway perteneciente a la característica SmartEnergyMng con la misma cabecera que el método declarado para la clase Gateway de la característica HeaterMng, no es posible sobrescribir este método.

Además, la sobrescritura de los lenguajes orientados a objetos es más compleja en C# que en los lenguajes convencionales, ya que como en C++, para que un método puede ser sobrescrito, es necesario declararlo como virtual. Así que, es necesario de alguna manera prever si un método va a ser sobrescrito con el fin de declararlo como virtual. Una opción es declarar todos los métodos como virtual, independientemente si estos van a ser sobrescritos o no. Esta práctica es la recomendada por los autores de este trabajo, aunque resulte tediosa.

Dependencias entre Clases.

En referencia la gestión automática de las dependencias entre clases, las clases parciales C# no presentan el problema descrito en la sección 3.1. Este problema establecía que cuando una clase A (por ejemplo, la clase Sensor de la Figura 13), la cual tiene una referencia a otra clase B (ej. Gateway), si la clase A es extendida por medio de herencia a través de diferentes características, cuando se crea un producto final, la referencia a la clase A debería ser cambiada a la clase más profunda en el árbol de herencia correspondiente a una característica seleccionada. De esta forma se evitarían problemas indeseados relacionados con el sistema de tipos, tales como la conversión de tipos (castings).

Este problema no surge al emplear clases parciales C#, puesto que no se usa la herencia como mecanismo de extensión. En nuestro caso, simplemente se combinan los diferentes fragmentos, distribuidos en diversos ficheros parciales, de una misma clase. No se cambia por tanto el tipo de una clase, que sigue siendo el mismo para diferentes características. Además, cuando se crea una instancia de una clase, por ejemplo una instancia de la clase Gateway, esta instancia contendrá la funcionalidad perteneciente a todas las clases parciales seleccionadas y que se hayan combinado para crear la clase Gateway. Así que, respecto a este punto particular, las clases parciales C# proporcionan un funcionamiento similar a lenguajes como CaesarJ u ObjectTeams.

Composición a Nivel de Características.

Debido a que las clases parciales C# no soportan la modularización y encapsulamiento de clases relacionadas con una misma característica, no es posible componer aplicaciones a nivel de características. La composición de productos específicos se realiza por medio de la inclusión y exclusión de clases parciales desde el fichero de compilación del proyecto, como lo muestra la Figura 14. Esto significa que es necesario gestionar de forma individual los elementos pertenecientes a una misma característica y verificar manualmente la coherencia de las configuraciones producidas. Por ejemplo, se debe verificar que la clase LightCtrl se incluye en la

unidad de compilación si y sólo si la clase parcial de Gateway perteneciente a la característica LightMng también se incluye. Por tanto, se deben realizar varias tareas cada vez que un característica es seleccionada o deseleccionada, lo cual resulta tedioso y propenso a errores.

Verificación de la Consistencia de Composiciones.

C# no proporciona ningún mecanismo que garantice la creación de un fichero de compilación que contenga una selección de características válidas, por lo que dicha comprobación tendremos que realizarla manualmente. Sin embargo, si una característica utiliza clases y métodos declarados en otra característica, el compilador generará un error alertando de que se está tratando de usar elementos no definidos. Por ejemplo, la clase parcial Gateway perteneciente a la característica SmartEnergyMng, utiliza las clases WindowCtrl y HeaterCtrl. Si incluimos la clase SmartEnergyMng::Gateway en la unidad de compilación, pero por error no incluimos las clases WindowCtrl o HeaterCtrl, el compilador dará un error anunciando que existen elementos no definidos. No obstante, además de incluir las clases WindowCtrl y HeaterCtrl, deberíamos incluir todas las clases pertenecientes a las características LightMng y HeaterMng.

De esta manera, empleando clases parciales C# es posible detectar dependencias entre clases, pero no dependencias a nivel de características, que sería el caso ideal. Además, el compilador detecta e informa de errores, pero éstos deben ser solventados manualmente. Por ejemplo, si se incluye la característica SmartEnergyMng, las características WindowCtrl y HeaterCtrl deben ser incluidas manualmente. Existen lenguajes, tales como CaesarJ, que soportan la gestión automática de este tipo de dependencias.

Por último, C# tampoco detecta ni soluciona las restricciones de exclusión mutua, es decir, las situaciones donde si se selecciona una característica A, otra característica B no puede ser seleccionada. De cualquier manera, esta propiedad no es común en los lenguajes orientados a características.

3.2.1 Trabajos relacionados.

Hasta donde alcanza nuestro conocimiento, este es el primer trabajo que analiza la capacidad de la clases parciales de C# como mecanismos para implementar programación orientada a características. No obstante, existen varios trabajos que proponen la utilización de las clases parciales de C# para este tipo de implementación así como diversos análisis similares al nuestro, pero aplicados a otros lenguajes de programación.

(Laguna et al, 2007) proponen el uso de las clases parciales de C# como un mecanismo para la implementación de descomposiciones orientadas a características. Sin embargo, esta idea no es evaluada en profundidad. (Warmer y Kleppe, 2006) presentan un enfoque basado en un caso estudio de fábrica de software empresarial, en el cual un sistema es representado a través de un conjunto de modelos parciales. Un modelo parcial es un modelo de dominio específico, el cual contiene una descripción de un fragmento de la funcionalidad del sistema. Para obtener una especificación completa del sistema es necesario componer varios de estos modelos parciales. Estos modelos parciales se pasan como entrada a un generador de código, el cual

genera código fuente C# utilizando clases parciales. En la generación de código se vinculan todos los modelos parciales para formar el modelo completo. Para ello emplean diversas técnicas de desarrollo de software dirigido por modelos. No obstante, Warmer y Kleep emplean clases parciales como una técnica más de implementación, sin considerarlas como un mecanismo para implementar orientación a características.

(Lopez-Herrejon y Batory, 2001) proponen usar como problema estándar para la evaluación de metodologías para líneas de productos software un caso de estudio basado en una familia de aplicaciones para representar grafos. Dicho caso estudio, es sustituido en posteriores trabajos de los mismos autores por una línea de productos software para el cálculo de expresiones matemáticas. En el presente trabajo se ha optado por la automatización del hogar como caso estudio, ya que dicho casos estudios contienen dificultades y características similares a las propuestas por Herrejón y Batory, y teníamos fácil acceso a una amplia cantidad de material relacionado con el mismo (Sánchez et al, 2007; Nebrera et al, 2008; Fuentes et al, 2009).

(Lopez-Herrejon et al, 2005) realizó una evaluación relacionada con el soporte de orientación a características en cinco lenguajes de programación con técnicas avanzadas de modularización, específicamente AspectJ, HyperJ, Jiazzi, Scala y AHEAD. Este trabajo fue extendido en (Lopez-Herrejon, 2007) con pequeños estudios acerca de MultiJava, Classboxes, Colaboraciones Aspectuales, Delegation Layer y Classpects. En (Lopez-Herrejon y Batory, 2006), los mismos autores investigan la relación entre casos de usos y programación orientada a características. Basado en estos trabajos, se ha definido la lista de elementos que debe contemplar un lenguaje orientado a características (ver sección 3.1).

(Kuhlemann et al, 2010) analiza las fortalezas y debilidades de las colaboraciones en cuanto a la implementación de características. Este autor implementó varios casos estudios utilizando el lenguaje Jak, que es una extensión de Java que incorpora el concepto de colaboración. Kuhlemann proporciona un conjunto de recomendaciones acerca de escenarios donde las colaboraciones pueden resultar de utilidad para implementar características.

(Apel y Batory, 2006) estudiaron cuándo y cómo es conveniente utilizar mixins o aspectos como técnicas de implementación orientada a características. Los autores aplican aspectos y mixins a un caso estudio de una línea de productos para una red peer-to-peer. Identifican ventajas e inconvenientes de cada enfoque y proporcionan un conjunto de reglas acerca de cómo y en qué momento cada técnica es más adecuada. Figueiredo (Figueiredo et al, 2008) realiza un estudio empírico sobre como los lenguajes orientados a aspectos, como AspectJ y CaesarJ, afectan la estabilidad de un diseño de software para una línea de productos.

(Mezini y Ostermann, 2004) poseen un trabajo similar. En él, Mezini y Ostermman también analizaron ventajas e inconvenientes de las técnicas de programación orientadas a características y aspectos. Así mismo, proponen CaesarJ como un lenguaje que integra beneficios de ambos enfoques. (Gasiunas y Aracic, 2007) implementaron el caso estudio llamado Dungeon empleando el lenguaje orientado a características CaesarJ. Evaluaron el lenguaje CaesarJ con respecto a criterios de reutilización, extensibilidad, eficiencia del compilador, estabilidad, verificabilidad y mantenimiento. Gasiunas y Aracic asumen que

CaesarJ posee un buen soporte para la modularización de características, por lo que se centran en analizar los efectos colaterales negativos resultantes de dicha modularización. Este trabajo posee un enfoque más humilde y analiza si las clases parciales son un mecanismo adecuado para modularizar características, como previo paso al estudio de cómo otros atributos de calidad se ven afectados, tales como estabilidad y verificación.

(Chae y Blume, 2009) presentaron un trabajo donde analizan qué elementos son útiles en un lenguaje de programación orientado a características. Aplicaron sus conclusiones a un lenguaje para la construcción de compiladores. Emplearon este lenguaje para desarrollar una línea de productos software para una familia de compiladores.

3.2.2 Conclusiones.

De acuerdo con los resultados obtenidos, podemos concluir que las clases parciales de C# no representan un mecanismo adecuado para la programación orientada a características. Los principales problemas son:

- Insuficiente soporte para modularizar características.
- Insuficiente mecanismos para encapsular de características.
- Insuficiente soporte para extender métodos ya existentes.
- Insuficiente soporte para sobrescribir métodos ya existentes.
- Insuficiente soporte para la composición de aplicaciones a nivel de características.

Como punto positivo, las clases parciales C# soportan extensibilidad mediante la incorporación de nuevos atributos y métodos sin requerir herencia, lo cual evita problemas relacionados con el sistema de tipos, como innecesarias conversiones de tipos.

En resumen, las clases parciales de C# por sí mismas no son un mecanismo suficiente para implementar descomposiciones orientadas a características. Por lo tanto, una solución factible podría ser, combinar las clases parciales C# con técnicas de generación de código y desarrollo software dirigida por modelos.

La idea sería, en primer lugar, desde un diseño UML tal como el mostrado en la Figura 2, generar esqueletos de código (clases y cabeceras de métodos). Estos esqueletos de código se completarían manualmente, representando la infraestructura desde la cual se obtendrían productos específicos. A continuación, para obtener productos concretos, deberían crearse configuraciones del modelo de características de acuerdo con las funcionalidades que se deseen incluir en el producto final. Mediante el uso de una herramienta de modelado de características, tal como Hydra[4], aseguraríamos que la configuración creada es correcta y podríamos además gestionar de forma automática dependencias entre características.

A continuación, esta configuración del modelo de características podría servir como entrada a un generador de código, el cual se encargaría de crear el fichero de compilación adecuado,

[4] http://caosd.lcc.uma.es/spl/hydra

conforme a las características seleccionadas, así como cualquier otro artefacto necesario para compilar y construir el producto específico deseado por el cliente.

Por tanto, la responsabilidad de gestionar y componer aplicaciones a nivel de características sería trasladado del lenguaje al modelo de características y al generador de código, los cuales suplirían las carencias de las clases parciales de C#.

4. Sumario y Trabajos Futuros.

Este trabajo ha presentado una evaluación de las clases parciales C# como mecanismo para la programación orientada a características. Se han aplicado las clases parciales a un caso estudio industrial para automatización de hogares. Se identificando beneficios y problemas de las clases parciales C# con relación a la programación orientada a características.

Se ha identificado que las clases parciales son un interesante concepto para implementar extensiones que impliquen la incorporación de nuevos métodos y atributos a clases ya existentes. No obstante, las clases parciales de C# carecen de un mecanismo que permita encapsular características y tratarlas como unidades de composición. Las clases parciales trabajan adecuadamente para implementar características como *LightMng* y *WindowMng*, las cuales son extensiones que implican simplemente añadir nuevas clases y métodos o atributos a las clases ya existentes. El único inconveniente sería que el método constructor de estas clases no puede ser extendido.

Sin embargo, cuando las extensiones requieren algún tipo de sustitución, sobrescritura o extensión de un método ya existente, las clases parciales C# no son un mecanismo adecuado para la programación orientada a características, ya que no soportan sobrescritura de métodos o extensiones de métodos ya definidos. Por ejemplo, no se puede implementar adecuadamente la característica *SmartEnergyMng* usando clases parciales de C#, ya que es necesario sustituir el método *adjustTemperature* para verificar si es necesario realizar operaciones adicionales de acuerdo a los datos recogidos por los sensores.

Un lenguaje orientado a características debería proporcionar construcciones para especificar que características deben ser instanciadas o compuestas con objeto de crear un producto específico. En el caso ideal deberíamos especificar simplemente qué conjunto de características deben ser incorporadas a un producto específico, en lugar de tener que identificar los elementos individuales a incluir por cada característica seleccionada. Como consecuencia de la falta de mecanismos de modularización de características en C#, no es posible ni componer grupos de características ni validar dichas composiciones.

Por lo tanto, se concluye que las clases parciales son un mecanismo útil para realizar aplicaciones orientadas a característica en ciertos escenarios concretos, pero que no son un mecanismo útil para implementar descomposiciones orientadas a características en general.

Por otra parte, como trabajos futuros se plantea la posibilidad de explorar como las técnicas de generación de código y el desarrollo software dirigido por modelos podrían ayudar a solventar las limitaciones identificadas en este trabajo de las clases parciales C#.

Como es bien conocido, un considerable número de empresas de software utilizan la plataforma .NET como plataforma base de desarrollo. Introducir un conjunto de nuevas construcciones para soportar programación orientada a características puede suponer una curva de aprendizaje importante para estas compañías. Con el objetivo de suavizar esta curva de aprendizaje, consideramos que una metodología orientado a características que requiriese para un conjunto de cambios y extensiones mínimos al lenguaje de programación habitual de la empresa, la situación deseable. Por tanto, nuestra idea es combinar las clases parciales con modelos de características y técnicas de generación de código, trasladando la responsabilidad de manejar la selección de características y validación de la composición al modelo de características y al generador de código. Además, tendríamos que idear una solución que permitiese extender métodos ya existentes o incluso sobrescribirlos. De esta manera, los desarrolladores .NET podrían continuar usando su lenguaje de programación habitual, sin requerir aprender nuevos conceptos relaciones con la orientación a características.

5. Conocimientos y Habilidades Desarrolladas.

A lo largo del máster se han fomentado el desarrollo de habilidades tanto en el proceso investigador como en el proceso cognoscitivo en sí, de cada una de las áreas abordas en el curso. Particularmente, puedo identificar algunas de las competencias cultivadas en este período, en cuanto a la formación en aspectos generales de la investigación en informática e inteligencia artificial.

Se han identificado los siguientes elementos necesarios para llevar a cabo un proceso de investigación científico-tecnológica que cumpla con los estándares académicos y trascienda en una aportación importante para la comunidad científica.

- A partir de concebir la idea a investigar, es necesario definir adecuadamente los parámetros que la delimitan, formular los objetivos, la problemática y la razón por la cual merece ser investigada.
- Es importante enmarcar la investigación, en algún estándar o metodología probada que ayudan en la definición de mínimas actividades concretas a llevar a cabo para la consecución de resultados coherentes y de calidad.
- Las destrezas en consulta y búsqueda de fuentes bibliográficas, es clave para el progreso favorable de la investigación. Durante el periodo docente del máster, se abordaron tópicos referentes a técnicas de consulta y referencias bibliográficas aplicadas a investigación científica-tecnológica, por diferentes medios electrónicos, como buscadores académicos especializados, revistas o congresos, entre otros.
- Evaluar con base en criterios de excelencia y objetividad, las aportaciones de otros investigadores. Tener esto presente, resulta favorable en el análisis de antecedentes y trabajos relacionados que puedan ser útiles en nuestra investigación.
- Comunicar apropiadamente resultados y conocimientos que los sustentan en forma expositiva, objetiva y sintética. Siempre respetando la deontología científica. Es decisivo para la aceptación y notoriedad de los resultados tangibles de la investigación.
- Es importante aplicar adecuadas técnicas de presentación oral y recursos audiovisuales, como las impartidas al final del período docente, motivado a la evaluación de varias exposiciones de los alumnos referentes a diferentes tópicos de ingeniería de software.

Se ha asistido a seminarios y conferencias, relacionado con las siguientes asignaturas:

Sistemas Avanzados Basados en Componente:

- Seminario “Introducción Práctica al Desarrollo de Software Dirigido por Modelos con Eclipse”. Impartido por la Dra. Cristina Vicente-Chicote del Dpto. de Tecnologías de la Información y las Comunicaciones de la Universidad Politécnica de Cartagena.

  En este seminario se ofreció a los asistentes una introducción práctica al Desarrollo Software Dirigido por Modelos utilizando algunas de las herramientas ofrecidas por Eclipse para dar soporte a este enfoque. En concreto, se presentaron las principales características de los plug-ins EMF (Eclipse Modeling Framework) y GMF (Graphical

Modeling Framework) destinados, respectivamente, a dar soporte a la definición de nuevos lenguajes de modelado y a la construcción de editores gráficos de modelos.

- Seminario sobre “Ingeniería de Requisitos Orientados a Aspectos”. Impartido por la Dra. Ruzanna Chitchyan de la Universidad de Lancaster.

  En esta conferencia se abordaron tópicos relacionados a la Ingeniería de Requisitos Orientada a Aspectos (AORE, por sus siglas en ingles Aspect-Oriented Requirements Engineering). Proporcionó una visión general de los actuales enfoques de Ingeniería de Requisitos Orientada a Aspectos. Se presentaron técnicas y herramientas actuales para la identificación de requisitos aspectuales (a partir de un texto en inglés), así como, su utilización e importancia para en actividades posteriores del ciclo de vida del desarrollo software.

Fundamentos Teóricos de Inteligencia Artificial.

- Ciclo de conferencias “Nuevas Tendencias en Sistemas Inteligentes y Soft Computing”. Departamento de Ciencias de la Computación e Inteligencia Artificial Universidad de Granada.

  El objetivo del seminario fue ofrecer un foro donde investigadores de primera línea mostraron el estado del arte y últimos logros en Soft Computing e Inteligencia Artificial, tanto a nivel industrial como comercial. Los ponentes invitados expusieron sus últimos desarrollos, la base científica necesaria para hacer posible la transferencia tecnológica; es decir, la transferencia del conocimiento desde la investigación en Soft Computing e Inteligencia Artificial hasta los problemas de la vida real. Así como, explicaron nuevas perspectivas para estimular el interés en las “Técnicas Inteligentes”.

6 Referencias.

(Albahari y Albahari, 2010)
Joseph Albahari y Ben Albahari. "*C# 3.0 in a Nutshell"*. O'Reilly, 4 edition, Enero 2010.

(AMPLE D3.4, 2007)
Vaidas Gasiunas, Pablo Sánchez, Carlos Nebrera, Nadia Gámez, Lidia Fuentes, Jacques Noy´e, Mario Südholdt, Angel Núñez, Christoph Pohl, Andreas Rummler, Iris Groher, Christine Schwanninger y Markus Volter. *"Methodology for Using AOP and MDD in Combination for Variability Management in SPLs"*. Deliverable D3.4, AMPLE Project, Octubre 2007.

(AMPLE D3.2, 2007)
Vaidas Gasiunas, Pablo Sánchez, Carlos Nebrera, Nadia Gámez, Lidia Fuentes, Jacques Noy´e, Mario Südholdt, Angel Núñez, Christoph Pohl, Andreas Rummler, Iris Groher, Christine Schwanninger, and Markus Völter. "*Overview of Extensions/Improvements to Existing Implementation Technologies".* Deliverable D3.2, AMPLE Project, December 2007.

(Apel et al, 2008)
Sven Apel, Thomas Leich, Marko Rosenmüller & Gunter Saake. "*Code Generation to Support Static and Dynamic Composition of Software Product Lines"*. Octubre 2008, Nashville, Tennessee USA.

(Apel y Batory, 2006)
Sven Apel y Don S. Batory. *When to Use Features and Aspects: "A Case Study*". In Proc. of 5th Int.Conference on Generative Programming and Component Engineering (GPCE), páginas59–68, Portland (Oregon, USA), Octubre 2006.

(Apel et al, 2005)
Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. *FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++.* 2005.

(Aracic et al, 2006)
Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. "*An Overview of CaesarJ*". Transactions on Aspect-Oriented Software Development, 3880:135–173, April 2006.

(Aracic et al, 2005)
Ivica Aracic, Vaidas Gasiunas, Mira Mezini, Klaus Ostermann. "*An Overview of CaesarJ*". Darmstadt University of Technology, D-64283 Darmstadt, Germany. Páginas 1-10. 2005.

(Batory, 2005)
Don Batory. "*A Tutorial on Feature Oriented Programming and Product-Lines*", International Conference on Aspect-Oriented Software Development (AOSD). Chicago, USA. Marzo, 2005. http://aosd.net/2005/tutorials/fop.php.

(Batory et al, 2005)
Don S. Batory. "*Feature Models, Grammars, and Propositional Formulas*". In J. Henk Obbink and Klaus Pohl, editors, Proc. of the 9th Int. Conference on Software Product Lines (SPLC), volume 3714 of LNCS, páginas7–20, Rennes (France), Septiembre2005.

(Batory et al, 2004)
D. Batory, J. N. Sarvela, and A. Rauschmayer. "*Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering*", Pgs. 1-10. Department of Computer Science University of Magdeburg, Alemania. 2004.

(Chae y Blume, 2009)
Wonseok Chae y Matthias Blume. "*Language Support for Feature-Oriented Poduct Line Engineering*". In Proceedings of the 1st Int. Workshop on Feature-Oriented Software Development (FOSD), páginas3–10, Denver (Colorado, USA), Octubre 2009.

(Chul et al, 2002)
Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. *"Feature-Oriented Product Line Engineering*". IEEE Software, 19 (4):58–65, July-August 2002.

(Ernst, 1999)
E. Ernst. *"gbeta - A Language with Virtual attributes, Block Structure, and Propagating, Dynamic Inheritance".* PhD thesis, Department of Computer Science, University of Aarhus, Dinamarca, 1999.

(Figueiredo et al, 2008)
Eduardo Figueiredo, Nélio Cacho, Cláudio Sant'Anna, Mario Monteiro, Uir´a Kulesza, Alessandro Garcia, S´ergio Soares, Fabiano Cutigi Ferrari, Safoora Shakil Khan, Fernando Castor Filho, y Francisco Dantas. "*Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability"*. In Proc. of the 30th Int. Conference on Software Engineering (ICSE), páginas261–270, Leipzig (Germany), Mayo 2008.

(Flanagan ET AL, 2008)
Flanagan y Matsumoto. "*The Ruby Programming Language"*. First. O'Reilly. 2008.

(Fuentes et al, 2009)
Lidia Fuentes, Carlos Nebrera y Pablo Sánchez. "*Feature-Oriented Model-Driven Software Product Lines: The TENTE Approach*". In Eric Yu, Johann Eder, and Colette Rolland, editors, Proc. of the Forum of the 21st Int. Conference on Advanced Information Systems (CAiSE), volume 453 of CEURS Workshops, páginas 67–72, Amsterdam (Holanda), Junio 2009.

(Gasiunas y Aracic, 2007)
Vaidas Gasiunas and Ivica Aracic. *Dungeon: "A Case Study of Feature-Oriented Programming with Virtual Classes*". In Proc. of the 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE), 6th International Conference on Generative Programming and Component Engineering (GPCE), Octubre 2007.

(Groher y Völter, 2009)
Iris Groher and Markus Völter. "*Aspect-Oriented Model-Driven Software Product Line Engineering. Transactions on Aspect-Oriented Software Development (Special Issue on Aspects and Model-Driven Engineering)".* 6:111–152, 2009.

(Herrmann, 2007)
Stephan Herrmann. *A Precise Model for Contextual Roles: "The Programming Language ObjectTeams/Java"*. Applied Ontology, 2(2):181–207, 2007.

(Herrmann, 2005)
Stephan Herrmann. "*Programming with Roles in ObjectTeams/Java. Technische Universität Berlin"*. 2005.

(Hund et al, 2007)
Christine Hundt, Katharina Mehner, Carsten Pfeier, Dehla Sokenou. "*Improving Alignment of Crosscutting Features with Code in Product Line Engineering*", In Journal of Object Technology, vol. 6, no. 9, páginas 417-436. Special Issue: Tools Europe 2007.

(Kuhlemann et al, 2010)
Martin Kuhlemann, Norbert Siegmund y Sven Apel. "*Using Collaborations to Encapsulate Features*". An Explorative Study. In Proc. of the 4th Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), páginas139–142, Linz (Austria), Enero 2010.

(Laguna et al, 2007)
Miguel A. Laguna, Bruno González-Baixauli y José M. Marqués. "*Seamless Development of Software Product Lines*". In Proc. of the 6th Int. Conference on Generative Programming and Component Engineering (GPCE), páginas85–94, Salzburg (Austria), Octubre 2007.

(Lopez-Herrejon, 2007)
Roberto E. Lopez-Herrejon. Language *and UML Support for Features: "Two Research Challenges".* In 1st Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), páginas97–100, Limerick (Ireland), Enero 2007.

(Lopez-Herrejon y Batory, 2001)
Roberto E. Lopez-Herrejon y Don S. Batory. *"A Standard Problem for Evaluating Product-Line Methodologies*". In Jan Bosch, editor, Proc. of the 3rd Int. Conference on Generative and Component-Based Software Engineering (GCSE), volume 2186 of LNCS, páginas10–24, Erfurt (Germany), Septiembre 2001.

(Lopez-Herrejon y Batory, 2006)
Roberto E. Lopez-Herrejon y Don S. Batory. *Modeling Features in Aspect-Based Product Lines with Use Case Slices: "An Exploratory Case Study*". In Thomas Kühne, editor, Reports and Revised Selected Papers of Workshops and Symposia at MoDELS, volume 4364 of LNCS, páginas6–16, Genova (Italy), Octubre 2006.

(Lopez-Herrejon et al, 2005)
Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. "*Evaluating Support for Features in Advanced Modularization Technologies"*. In Andrew P. Black, editor, Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP), volume 3586 of LNCS, páginas169–194, Glasgow (United Kingdom), July 2005.

(Lopez-Herrejon, 2005)
Roberto E. Lopez-Herrejon. "*Understanding Feature Modularity in Feature Oriented Programming and its Implications to Aspect Oriented Programming*". Departamento de Ciencias de la Computación de la Universidad de Texas-Austin, 2005.

(Mezini y Ostermann, 2004)
Mira Mezini and Klaus Ostermann. "*Variability Management with Feature-Oriented Programming and Aspects"*. In Proc. of the 12th Int. Symposium on Foundations of Software Engineering (FSE), páginas127–136, Newport Beach (California, USA), Octubre-November 2004.

(Nebrera, 2009)
Carlos Nebrera. "*Transformaciones de Modelo a Código Para Líneas de Productos Software*". Dpto. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Junio 2009.

(Nebrera et al, 2008)
Carlos Nebrera, Pablo Sánchez, Lidia Fuentes, Christa Schwanninger, Ludger Fiege, y Michael Jager. "*Description of Model Transformations from Architecture to Design*". Deliverable D2.3, AMPLE Project, Septiembre 2008.

(Object Management Group, 2005)
Object Management Group (OMG). *Unified Modeling Language: "Superstructure v2.0"*, (formal/05-07-04), August 2005.

(Odersky et al, 2004)
Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger. "*An Overview of the Scala Programming Language"*. Technical Report IC/2004/64.

(Odersky, 2007)
Martin Odersky. "*The Scala Language Specification Version 2.6"*. Programming Methods Laboratory EPFL Suiza. December, 2007.

(Odersky y Zenger, 2005)
Martin Odersky y Matthias Zenger. *"Scalable Component Abstractions*". Páginas 1-7. San Diego, USA. Octubre 2005.

(Prehofer, 1997)
Christian Prehofer. *Feature-Oriented Programming: "A Fresh Look at Objects*". In Mehmet Aksit and Satoshi Matsuoka, editors, Proc. of the 11th European Conference Object-Oriented Programming (ECOOP), volume 1241 of LNCS, páginas419–443, Jyv¨askyl¨a (Finland), June 1997.

(Sánchez et al, 2007)
Pablo Sánchez, Nadia Gámez, Lidia Fuentes, Neil Loughran, y Alessandro Garcia. *"A Metamodel for Designing Software Architectures of Aspect-Oriented Software Product Lines*". Deliverable D2.2, AMPLE Project, Septiembre 2007.

(Warmer y Kleppe, 2006)
Jos Warmer y Anneke Kleppe. "*Building a Flexible Software Factory Using Partial Domain Specific Models*". In Proc. of the 6th Workshop on Domain-Specific Modeling (DSM), 21th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), páginas15–22, Portland (Oregon, USA), Octubre 2006.

(Zave, 1999)
Pamela Zave. FAQ Sheet on Feature Interaction. AT&T, disponible en http://www2.research.att.com/~pamela/faq.html, 1999.