FACULTAD DE CIENCIAS

UNIVESIDAD DE CANTABRIA

Proyecto Fin de Carrera

Desarrollo de una Línea de Productos Software para Automatización de Hogares usando Clases Parciales de C#

(Development of a Software Product Line for Automated Homes using C# Partial Classes)

Para acceder al Título de
INGENIERO EN INFORMÁTICA

Autor: Alejandro Pérez Ruiz
Julio 2011

Capítulo 1

Antecedentes

1.1. Líneas de Productos Software

Una línea de productos software (SPL) es un conjunto de sistemas de software que comparten un conjunto común y gestionado de aspectos que satisfacen las necesidades específicas de un segmento de mercado o misión y que son desarrollados a partir de un conjunto común de activos fundamentales [de software] de una manera prescrita [**?**]

El objetivo de una Línea de Productos Software [**?**] es crear una infraestructura adecuada a partir de la cual se puedan derivar, tan automáticamente como sea posible, productos concretos pertenecientes a una familia de productos software. Una familia de productos software es un conjunto de aplicaciones software similares, que por tanto comparten una serie de características comunes, pero que también presentan variaciones entre ellos.

Un ejemplo clásico de familia de productos software es el software que se entrega con un teléfono móvil. Dicho software contiene una serie de facilidades comunes, tales como agenda, recepción de llamadas, envío de mensajes de texto, etc. No obstante, dependiendo de las capacidades y coste asociado al dispositivo móvil, éste puede presentar diversas funcionalidades opcionales, tales como envío de correos electrónicos, posibilidad de conectarse a Internet mediante red inalámbrica, radio, etc.

La idea de una Línea de Productos Software es proporcionar una forma automatizada y sistemática de construir un producto concreto dentro de una familia de productos software mediante la simple especificación de que características deseamos incluir dentro de dicho producto. Esto representa una alternativa al enfoque tradicional, el cual se basaba simplemente en seleccionar el producto más parecido dentro de la familia al que queremos construir y adaptarlo manualmente.

1.2. Programación Orientada a Características

Los lenguajes orientados a características, tienen como objetivo encapsular conjuntos coherentes de la funcionalidad de un sistema software en módulos independientes y fácilmente componibles para permitirnos alcanzar una mejor extensibilidad y reusabilidad. Estos módulos reciben el nombre de característica. Una característica se puede definir como un incremento de la funcionalidad de un sistema [**?**].
Estos lenguajes son especialmente útiles en el contexto del desarrollo de líneas de producción software, ya que nos permiten separar las características comunes de la mayoría de los productos de la línea de producción de las características que varían de producto a producto.

1.2.1. Limitaciones de la Orientación a Objetos frente a la Programación Orientada a Características

Para analizar las limitaciones que posee la programación orientada a objetos frente a la programación orientada a características se ha hecho uso del problema estándar de las líneas de productos de las expresiones(en inglés, Expression Product-Line). Éste es un problema fundamental en el diseño del software, que consiste en la extensión de nuevas operaciones y representación de los datos para su posterior combinación. Ha sido ampliamente estudiado dentro del contexto del diseño de los lenguajes de programación, donde se enfocaba en lograr la extensibilidad de los tipos de datos y las operaciones de una manera segura.Pero en vez de concentrarnos en este tema, consideraremos los aspectos del diseño y la síntesis del problema de producir una familia de productos. Más concretamente, ¿qué características están presentes en el problema?¿Cómo las podemos modularizar? Y ¿cómo podemos construir diferentes configuraciones?

Descripción del problema de las expresiones

El objetivo es definir los diferentes tipos de datos para representar la siguiente gramática:

Tres operaciones para las expresiones serán definidas en esta gramática:

1. Print: mostrará por consola la expresión en el formato correspondiente(infijo, prefijo o posfijo).
2. Eval: evaluará la expresión y retornará el valor numérico.
3. ShortEval: evaluará la expresión realizando la operación de multiplicación cortocircuitada, es decir, si el primer operando es 0 directamente retornará el valor 0 para la expresión.

```
Exp :: = Integer | AddInfix | MultInfix | AddPostfix | MulltPostfix |
 AddPrefix | MultPrefix
Integer     :: <positive-negative integers>
AddInfix    :: = Exp "+" Exp
MultInfix   :: = Exp "*" Exp
AddPostfix  ::= Exp Exp "+"
MultPostfix ::= Exp Exp "*"
AddPrefix   ::= "+" Exp Exp
MultPrefix  ::= "*" Exp Exp
```

Figura 1.1: Gramática del lenguaje de expresiones

Teniendo el problema presente podemos identificar dos conjuntos de características, por un lado las operaciones {Print, Eval, ShortEval} y por otro los tipos de datos {AddInfix, MultInfix, AddPostfix...}. Por lo tanto, el objetivo es implementar una línea de productos software que nos permita crear las distintas configuraciones posibles para este problema.

Resolviendo el problema con C#

La figura 1 representa el diseño de clases UML que soporta las variabilidades identificadas anteriormente. Este diseño es replicado para cada tipo de operación(la figura 2 muestra el diagrama de clases para la operación de imprimir expresión en formato infijo).
Tras trasladar estos diagramas a C# se observan las carencias que posee un lenguaje orientado a objetos cuando implementamos un problema propio de la programación orientada a características. Por un lado he utilizado el mecanismo de herencia para añadir funcionalidades a una clase existente, pero este mecanismo es jerárquico, lo que hace que la clase hija herede toda la funcionalidad de su clase padre.
Por ejemplo, para crear una nueva configuración que contenga las operaciones evaluar e imprimir en formato infijo se necesitaría crear una nueva subclase que heredase de las dos clases que contienen las operaciones citadas anteriormente, pero esto no es posible, ya que C#, como la mayoría de los lenguajes orientados a objetos, solo permiten hacer uso de la herencia múltiple entre interfaces. Por lo tanto, por cada clase que se quiera heredar debemos crear una nueva interfaz que sea extendida por la clase a heredar y por la subclase que contendrá la nueva configuración.
El uso de esta técnica hace que, un incremento de funcionalidad representado por un conjunto de nuevas subclases que heredan de un conjunto de clases superiores, no sea 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. Aun 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.
Otro problema es el manejo de las dependencias, ya que la herencia tradicional obliga a que las clases y subclases tengan 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 potencialmente, lo que resulta una situación indeseable. Esto incrementa la complejidad en las relaciones y dependencias entre clases.

1.2.2. Ventajas de los lenguajes orientados a características

Los lenguajes orientados a características nos otorgan una mayor flexibilidad, ya que nos permiten que clases individuales puedan ser compuestas por un conjunto de características, por tanto son especialmente recomendados para utilizarlos con las líneas de producción software.

Para estudiar las ventajas de los lenguajes orientados a características se trabajará con CaesarJ que es un lenguaje de programación basado en Java, que nos proporciona una mayor modularidad y el desarrollo a través de componentes reusables.Para ello trabaja con conceptos como clases y paquetes en única entidad, llamada familia de clases, que constituye unidades de encapsulamiento adicional para agrupar 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.

Para hacer uso de lo citado anteriormente y ver su beneficio con las líneas de producción software se ha vuelto a utilizar el ya comentado problema de las expresiones de la subsección anterior. Pero en este caso, el diseño cambia significativamente debido a la características expuestas de CaesarJ.

Por un lado en la figura 3(diagrama de clases con paquetes)se muestra que por cada operación, tenemos una familia de clases, siendo la familia de clases que encapsula a todas las demás, la denominada Expressions. Ésta tiene la estructura de clases representado en la figura 4(Diagrama de clases de Expressions), y cada familia de clases lo que hace es redefinir las clases virtuales de Expressions con las operaciones necesarias en cada caso.
Para realizar configuraciones, únicamente se tiene que crear una nueva clase que extienda a las características que se deseen.A modo de ejemplo,la figura 6 muestra el código necesario para crear una nueva configuración que contenga las operaciones de impresión en formato posfijo y evaluación de una expresión.Con todo esto, vemos como CaesarJ otorga un gran nivel de encapsulamiento y de reusabilidad de los componentes

1.3. Clases Parciales C#

Las clases parciales C# [?] nos permiten dividir la implementación de una clase, estructura o interfaz en varios archivos de código fuente. Cada fragmento representa una parte de la funcionalidad global de la clase. Todos estos fragmentos se combinan en tiempo de compilación para crear una única clase, la cual contiene toda la funcionalidad especificada en las clases parciales. Por lo tanto,las clases parciales C# pueden utilizarse como un mecanismo adecuado 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 para 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 x–>Que será la que contenga lo de las expresiones).

1.4. Caso de Estudio: Hogares Inteligentes Automatizados

El objetivo de estos hogares es el aumento de la comodidad y seguridad de sus habitantes, así como hacer un uso más eficiente de la energía consumida. Se ha elegido este dominio por ser un dominio donde el uso de un 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 enormemente, 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. Los ejemplos más comunes de tareas automatizadas dentro de un hogar inteligente son el control de luces, ventanas, puertas, persianas, calefacción, 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 el funcionamiento de los hogares los dispositivos de control inteligente leen datos de los sensores, procesan estos datos, y activan los actuadores, si fuese necesario. Para algunas tareas de control y automatización los dispositivos de control inteligentes actúan autónomamente.
La puerta de enlace(Gateway, en inglés) es el servidor central del hogar inteligente. Éste realiza el procesado y el almacenamiento de los datos requeridos para las aplicaciones. Los usuarios(tanto residentes como técnicos), pueden acceder a los servicios ofrecidos a través de la puerta de enlace o a través de otras interfaces de usuario disponibles.
El presente proyecto se centrará en el desarrollo de un hogar inteligente como una línea de productos software, con un número variable de plantas y habitaciones. El número de habitaciones por planta es también variable. La arquitectura de la línea de productos implementa varios servicios, clasificados en funciones básicas y complejas, que son descritas a continuación.
Las *funciones básicas* son:

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

2. *Control automático de puertas:* Las puertas pueden abrirse automáticamente. Además, el control de acceso de algunas puertas puede ser necesario.

3. *Control automático de ventanas:* Los residentes tienen que ser capaces de controlar las ventanas automáticamente.Además, si la ventana tiene persianas, éstas deben ser desenrolladas o enrolladas automáticamente.

4. *Control automático de temperatura:* El usuario será capaz de ajustar la temperatura de la casa.La temperatura podrá ser manejada en grados Celsius o Fahrenheit.

Las *funciones complejas* son:

1. *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.

2. *Control inteligente de luces:* Las luces deberán automáticamente apagarse/encenderse dependiendo de diversos factores. Si no hay luz natural fuera de la casa, las luces deberán automáticamente encenderse. Cada vez que un habitante entre a una habitación donde no haya iluminación suficiente, las luces automáticamente se encenderán, a menos que la habitación esté en modo "sleep". Este modo previene el alumbrado automático de una habitación,en el caso de que alguien estuviese durmiendo.

Cada una de estas funciones son opcionales. Las personas interesadas en adquirir el sistema podrán seleccionar el numero de funciones que ellos deseen.

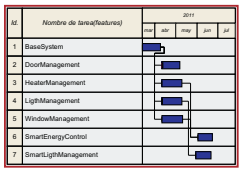

Figura 2.1: Diagrama de dependencias entre tareas

Capítulo 2

Antecedentes

2.1. Planificación

La figura 2.1 muestra