Aplicando *scaffolding* en el desarrollo de Líneas de Producto Software[*]

Nieves R. Brisaboa, Alejandro Cortiñas, Miguel R. Luaces, Óscar Pedreira

Laboratorio de Bases de Datos
Universidade da Coruña
Campus de Elviña, A Coruña, España
{brisaboa, alejandro.cortinas, luaces, oscar.pedreira}@udc.es

Resumen Las Líneas de Producto Software (LPS) constituyen una tecnología madura para producir software que ha sido objeto de una gran cantidad de investigación, por lo que existen numerosas técnicas, metodologías y herramientas para crearlas. Sin embargo, es complicado utilizar algunas de estas herramientas en la industria debido a factores como la rápida evolución que han tenido los entornos de desarrollo, lo que provoca que estas herramientas estén obsoletas, la falta de soporte para proyectos que utilizan diferentes lenguajes de desarrollo, o la dificultad en el mantenimiento del código de los productos generados por la LPS. Por otra parte, la popularidad de la técnica de *scaffolding* no ha parado de aumentar entre los desarrolladores de software desde que apareció hace unos años, a pesar de recurrir a alternativas poco valoradas en la academia tales como el uso de preprocesadores.
En este trabajo proponemos la utilización de la técnica de *scaffolding* para implementar una LPS, lo que nos permite superar algunas de las limitaciones clásicas de otras herramientas LPS.

Palabras clave: ingeniería de líneas de producto software, *scaffolding*, arquitectura software de propósito general, gestión de la variabilidad, desarrollo de software orientado a características

1. Introducción

El desarrollo de software basado en Líneas de Producto Software (LPS) es un campo de investigación con casi dos décadas de antigüedad que ha producido multitud de metodologías y herramientas. A pesar de los grandes avances en el formalismo de las LPS y de las ventajas que proporcionan al proceso de desarrollo de software [1,2,3,4], su aplicación en las empresas de desarrollo de software se ha visto reducido a contextos muy específicos como el desarrollo de sistemas embebidos [5,6,7]. En otras áreas más complejas, como el desarrollo

[*] Financiado por el CDTI y el Ministerio de Economía y Competitividad (PGE y Fondos FEDER) [TIN2013-46238-C4-3-R, TIN2013-46801-C4-3-R, Ref. IDI-20141259, Ref. ITC-20151305]; y por la Xunta de Galicia (cofinanciado por FEDER) [GRC2013/053]

de aplicaciones web u otros ámbitos con tecnologías punteras y cambiantes, la aplicación de la ingeniería de Líneas de Producto Software se aplica cada vez con mayor frecuencia, aunque las propuestas en la literatura normalmente se centran en el modelado de la variabilidad del dominio a un nivel de abstración alto más que en la gestión de la variabilidad a nivel de la implementación [8]. Es por ello que existe la necesidad de nuevas herramientas y tecnologías LPS para este tipo de casos [9].

Paralelamente, en la industria de desarrollo de software han aparecido nuevas técnicas y metodologías para agilizar los procesos de desarrollo. Entre ellas destaca el *scaffolding*, popularizado por Ruby on Rails, Yeoman o Spring Roo. El *scaffolding* consiste en la generación de código a partir de plantillas predefinidas y de una especificación proporcionada por el desarrollador. Se suele utilizar para generar código repetitivo que se puede especificar fácilmente y generar a partir de una plantilla. De esa forma, el desarrollador se ahorra parte de la codificación ya que sólo tendrá que revisar y retocar mínimamente el código generado. Aunque el *scaffolding* es un concepto relativamente moderno en el desarrollo de software, los lenguajes de cuarta generación, de moda en los años 80, tenían funcionalidades similares (por ejemplo, generando formularios directamente a partir del modelo de datos de la base de datos). Igualmente, el *scaffolding* puede considerarse como una aplicación informal de los conceptos definidos en el campo de investigación de desarrollo de software dirigido por modelos.

Nuestro objetivo inicial fue crear una LPS para automatizar la generación de aplicaciones web para la gestión de información geográfica [10,11]. Sin embargo, a la hora de elegir la tecnología para implementarla no hemos encontrado ninguna alternativa en el contexto de las LPS que nos permita desarrollar la plataforma deseada debido a las siguientes cuestiones:

- Las herramientas LPS comerciales tienen un coste elevado que no puede ser asumido por una empresa de tamaño medio o pequeño.
- Las herramientas LPS no comerciales suelen depender de tecnología obsoleta (por ejemplo, una versión del entorno de desarrollo Eclipse de 2009 [12]).
- El código de los productos generados suele ser difícil de mantener por su complejidad y falta de legibilidad, especialmente con los enfoques compositivos. Hay muchos trabajos en los que se considera importante y se analiza la claridad del código de la plataforma [13,14], pero no hemos encontrado ninguno en el que se analice la claridad del código de los productos generados.
- Las herramientas LPS suelen estar orientadas a generar productos que usan un único lenguaje de programación (por ejemplo, Java) siendo difícil su aplicación si lo que se quiere generar son productos multilenguaje (por ejemplo, con un servidor Java y un cliente JavaScript).

Por otra parte, debido al diseño de nuestra herramienta, es necesario que parte del código del producto se genere desde cero en tiempo de desarrollo. Por ejemplo, el modelo de datos de nuestros productos será variable (específico de cada producto), por lo que el código fuente para el manejo de los datos tiene que ser generado a partir de la especificación que haga el analista (en concreto esto supone la generación de scripts SQL y de clases Java).

Por todo ello hemos desarrollado un motor de derivación para LPS basado en *scaffolding*. Nuestra propuesta, por una parte, tiene en cuenta conceptos formales de LPS tales como la gestión de un modelo de variabilidad y la validación de las características seleccionadas para un producto. Por otra parte, proporciona las ventajas de *scaffolding*, permitiendo generar productos multilenguaje basados en tecnologías modernas y ampliamente soportadas (HTLM, CSS,JavaScript, etc.) y, mediante el uso de plantillas previamente definidas y bien estructuradas, facilita la generación de código claro, legible y extensible a partir de la especificación del analista.

El resto del artículo se organiza de la siguiente forma: la Sección 2 describe el contexto que motiva la necesidad de nuestra propuesta, la Sección 3 describe el funcionamiento de la herramienta que implementa nuestra propuesta, la Sección 4 presenta un caso de uso concreto de la herramienta, la Sección 5 describe nuestras conclusiones respecto al uso de LPS o *scaffolding*, y la Sección 6 presenta las conclusiones y el trabajo futuro.

2. Contexto, motivación y objetivos

Para entender las decisiones que se han tomado en el diseño de nuestra solución para implementar LPS es necesario explicar el contexto en el que se utilizará. Los aplicaciones habituales que desarrolla la empresa *spin-off* asociada a nuestro grupo de investigación comparten un gran número de funcionalidades comunes (por ejemplo, suelen ser aplicaciones web que tienen gestión de usuarios con autenticación, páginas de contenido estático, listados y formularios de datos, importación y exportación de datos, generación de informes, visualización de información geográfica, etc.). Dado que las necesidades de los clientes son diversas, el modelo de datos es variable y los listados y formularios deben generarse en función del mismo. Además de las funcionalidades comunes, hay bastantes aplicaciones cuya casuística requiere desarrollos bastante específicos y, a priori, no aprovechables en ninguna otra aplicación.

En resumen, los desarrollos de la empresa *spin-off* comparten una serie de características comunes, y además cuentan con ciertas características particulares y exclusivas a cada uno de ellos. Lo que pretendemos es tener una herramienta de generación de código que de soporte al conjunto de características comunes y que genere productos fácilmente extensibles para incluir las funcionalidades específicas.

Teniendo en cuenta lo anterior, hemos confeccionado una serie de requisitos que debe cumplir la herramienta o tecnología usada para implementar nuestra LPS.

- *El código producido debe ser legible y extensible.* Algunas herramientas LPS como AHEAD o FeatureHOUSE, al generar métodos y clases artificiales para aplicar sus mecanismos de composición, producen un código final poco legible (se puede observar en la Figura 1 cómo se genera un método privado artificial al componer dos características que afectan al mismo método).

```
@RestController
@RequestMapping("/api/calculator")
public class CalculatorResource {

    @RequestMapping(method = RequestMethod.POST)
    private ResponseEntity<ResultJSON> calculate__wrappee__Base(@RequestBody RequestJSON
     request) {
        return response entity;
    }

    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<ResultJSON> calculate(@RequestBody RequestJSON request) {
        if operation is division {...}
        return calculate__wrappee__Base(request);
    }
}
```

Figura 1. Ejemplo de código generado por FeatureHOUSE

Nuestra herramienta debe generar un código claro y legible para que sea posteriormente extensible. Al usar *scaffolding* la legibilidad del código va a depender de la calidad de las plantillas que usemos, y no se va a ver afectado por la utilización de la herramienta en sí.

- *Los productos producidos deben poder adaptarse a cualquier diseño.* Es decir, puesto que se trata de desarrollar productos diferentes para clientes diferentes, nuestra herramienta debe producir un código fuente que se pueda extender fácilmente dándole al equipo de desarrollo total libertad en el diseño del producto final, evitando el imponerle estructuras rígidas e inmodificables.
- *La herramienta no debe imponer tecnologías o entornos de trabajo.* Nuestra herramienta producirá código basado en plantillas adaptadas a los entornos de trabajo habituales de la *spin-off*.
- *La herramienta debe ser fácil de usar.* Se trata de que el coste de formación en el uso de la misma sea mínimo.
- *La herramienta debe producir código multilenguaje.* Todos los proyectos de la *spin-off* incluyen una mezcla de varias lenguajes de programación, entre otros: SQL, Java, JSP, JSTL, C#, Visual Basic, ASP, HTML, CSS, y JavaScript. Pocas herramientas LPS existentes en el estado del arte [2,3,4] producen código en diferentes lenguajes de programación a la vez.

Estos requisitos hacen inviable para nuestros propósitos la utilización de la mayor parte de herramientas LPS existentes.

3. Arquitectura, implementación y funcionamiento de la herramienta

3.1. Arquitectura

La Figura 2 muestra la estructura de la herramienta de derivación de la LPS mediante un diagrama de componentes de UML. El componente central es el

motor de derivación que se encarga de gestionar todo el proceso de generación del producto. Para realizar esta tarea delega en tres componentes: el *gestor del modelo de variabilidad*, el *motor de plantillas*, y el *gestor de ficheros*. El componente de la parte superior de la figura es un *cliente de línea de comandos* que proporciona un acceso sencillo al usuario analista para generar nuevos productos.

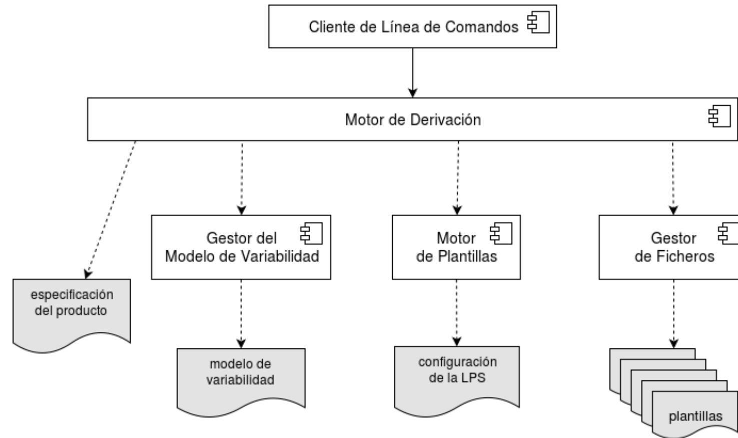

Figura 2. Diagrama de componentes de nuestra herramienta

El ***Gestor del Modelo de Variabilidad*** se encarga de construir el modelo de características a partir de la especificación del usuario y proporciona funcionalidades para importar/exportar un modelo de variabilidad desde y hacia un fichero, para validar que el modelo de variabilidad sea correcto (por ejemplo, que no tenga ninguna característica hoja que sea abstracta), para validar la selección de características que realiza el analista a la hora de generar los productos, y para detectar posibles problemas en el código fuente anotado (por ejemplo, características que aparecen en anotaciones pero no aparecen definidas en el modelo de variabilidad, o viceversa).

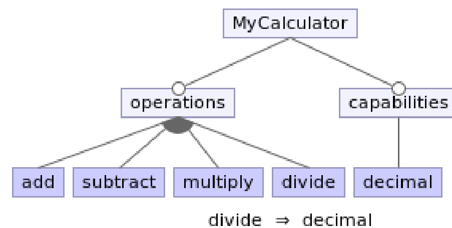

Figura 3. Modelo de características de una calculadora web

El modelo de variabilidad se indica siguiendo la especificación de Feature IDE (en el formato XML de Feature IDE, en JSON o en YAML) y permite restricciones con operadores lógicos además del árbol de variabilidad clásico. En la Figura 3 mostramos un modelo de variabilidad de una LPS simple que genera calculadoras web con distintas funcionalidades.

```
1  @RequestMapping(method = RequestMethod.POST)
2  public ResponseEntity<ResultJSON> calculate(@Valid @RequestBody RequestJSON request) {
3      /*% if (feature.divide) { %*/
4      if (request.getOperation() == Operation.DIVIDE) {
5          if (request.getSecond() == 0) {
6              return new ResponseEntity<ResultJSON>(new ResultJSON(), HttpStatus.BAD_REQUEST);
7          }
8      }
9      /*% } %*/
10     return new ResponseEntity<ResultJSON>(new
        ResultJSON(calculatorService.calculate(request.getFirst(),
11             request.getSecond(), request.getOperation())), HttpStatus.OK);
12 }
```

Figura 4. Ejemplo de método Java anotado

El ***Motor de Plantillas*** utiliza técnicas de *scaffolding* para aplicar la variabilidad seleccionada a las plantillas de la LPS y generar el código del producto final a partir del código fuente anotado y la especificación de la LPS. Las anotaciones en las plantillas se indican mediante comentarios del lenguaje de programación de la plantilla y su contenido es cualquier código JavaScript. En la Figura 4 se muestra un método de una clase Java anotado. En la línea 3 se indica que las líneas 4 a 8 sólo se deben incluir en el código del producto final si el analista ha seleccionado la característica *divide*. Como vemos, la anotación se ha realizado dentro de un comentario Java por lo que no interfiere con el compilador, IDE, o herramienta de validación que esté usando el desarrollador de la LPS.

Además de anotaciones relacionadas con las características, el motor de plantillas permite usar variables en las plantillas. En la Figura 5 se muestra un fichero de definición de proyecto de Maven con variables para la identificación del proyecto que serán sustituidas en el producto por los valores indicados por el analista durante el proceso de generación del producto. Finalmente, el motor de plantillas también puede validar la especificación y las plantillas con funcionalidades como detectar qué características no están referenciadas en ninguna anotación, detectar qué anotaciones no están definidas en el modelo de características, o calcular la complejidad de cada característica analizando cuanto código se ve modificado por cada una de ellas.

Por último, el ***Gestor de Ficheros*** se encarga de encapsular todas las tareas de acceso a las plantillas y permite recorrer recursivamente las carpetas en las que se encuentran las plantillas, leer y escribir ficheros de texto, detectar los

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
     http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <groupId><!--%= data.maven.group %--></groupId>
  <artifactId><!--%= data.maven.artifact %--></artifactId>
  <version><!--%= data.maven.version %--></version>
  <packaging>war</packaging>
  <name><!--%= data.title %--></name>
  <description><!--%= data.description %--></description>
</project>
```

Figura 5. Ejemplo de variables en fichero de proyecto Maven

ficheros binarios para no procesarlos y copiarlos directamente en el directorio de destino, etc.

3.2. Implementación

Desde el punto de vista tecnológico, nuestra solución está implementada en Node.js, un entorno de ejecución multiplataforma basado en el motor JavaScript V8 de Google Chrome. Node.js es ligero, independiente del sistema operativo y de cualquier IDE y su uso está muy extendido y en continuo crecimiento. La librería sólo requiere tener Node.js instalado, que está disponible para todos los sistemas operativos y que además es una plataforma muy activa y en crecimiento, por lo que se espera que su uso siga vigente durante muchos años.

Hasta llegar a la decisión de diseñar e implementar nuestro propio motor de derivación hemos desechado algunas alternativas. La primera aproximación fue utilizar Yeoman[1], una conocida plataforma de *scaffolding* que permite implementar *generadores de arquitecturas* de proyectos de cualquier tipo. Sin embargo, Yeoman tenía dos problemas: es una plataforma demasiado amplia y compleja para nuestras necesidades, y además exige programación *ad hoc* para cada proyecto en el que es usada. El siguiente paso fue utilizar únicamente el motor de plantillas de Yeoman, y otros motores de plantillas modernos. Finalmente acabamos descartando todos ellos por dos motivos: por un lado, ninguna de ellas cubría totalmente nuestros requisitos en cuanto a gestión de las anotaciones, y por otro lado ninguna proporciona funcionalidad para analizar el código anotado respecto al modelo de variabilidad.

Siguiendo este enfoque hemos diseñado e implementado un motor de derivación de LPS basado en *scaffolding* que cumple los requisitos descritos en la Sección 2 y además tiene las siguientes características:

- *Los componentes reutilizables de la LPS son las plantillas del* scaffolding. Los desarrolladores de la LPS usan las anotaciones del *scaffolding* para indicar los bloques que se corresponden con cada característica de la LPS. Esto no

[1] http://yeoman.io/

se aleja demasiado del uso de tecnologías anotativas para implementar LPS, como Antenna [15], Munge [16] o cualquier preprocesador.

- *La especificación del* scaffolding *se realiza con el modelo de variabilidad de la LPS.* La herramienta de *scaffolding* recibe como especificación el modelo de variabilidad y la selección de características del analista. Esta especificación se usa para validar la selección de características realizada y para generar el código fuente del producto a partir de las plantillas.
- *Anotaciones no intrusivas.* Dado que se usa un enfoque anotativo y los componentes se consideran plantillas de código, el lenguaje utilizado no es una limitación. Además, las anotaciones se realizan con comentarios del lenguaje de programación lo que facilita el desarrollo de los componentes reutilizables al no impedir el uso de herramientas de desarrollo estándar.
- *La herramienta de* scaffolding *permite generación de código a partir de la especificación.* Más allá de las funcionalidades estándar de una LPS de generación de código final mediante la inclusión y/o eliminación de bloques de código, nuestra herramienta además está capacitada para generar código fuente desde cero a partir de la especificación. Además del modelo de variabilidad y la selección realizada por el diseñador, la LPS puede recibir especificaciones adicionales (por ejemplo, el modelo de datos o la configuración del aspecto visual del producto) y la herramienta de *scaffolding* puede generar la funcionalidad de acceso a datos o las hojas de estilo CSS a partir de plantillas completadas con la especificación completada. También se permite en las plantillas utilizar bucles y otras estructuras de control avanzadas para generar código, lo que da al diseñador de la LPS una gran libertad y flexibilidad.

3.3. Funcionamiento

El proceso de generación de un producto se invoca utilizando el cliente de terminal y requiere cinco parámetros: el fichero con el modelo de variabilidad, el fichero con la configuración de la LPS, el fichero con la especificación del producto a generar, la carpeta con las plantillas de la LPS, y la carpeta donde se generará el producto nuevo.

Al recibir esos cinco parámetros, el motor de derivación comienza delegando en el gestor del modelo de variabilidad la interpretación del modelo proporcionado por el usuario. Si es correcto, el siguiente paso consiste en interpretar el fichero de configuración de la LPS que indica aspectos como qué cadenas de texto delimitarán el inicio y el fin de las anotaciones en función de la extensión de los ficheros a procesar, o qué ficheros de la carpeta de plantillas se deben ignorar a la hora de generar productos. Esta funcionalidad permite que elementos usados en el desarrollo de los componentes reusables no se transfieran al producto final.

A continuación, el motor de derivación interpreta el fichero con la especificación del producto a generar. Este fichero incluye la selección de características no abstractas que el analista desea para el producto, y la colección de valores que se usarán para sustituir las variables referenciadas en las anotaciones del código fuente. En la Figura 6 vemos un ejemplo de especificación de producto en la que

```
features:
  - decimal
  - add
  - subtract
  - multiply
  - divide
```

```
data:
  title: The Calculator
  description: A simple web calculator
  maven:
    group: es.udc.lbd
    artifact: web-calculator
    version: 0.1
```


Figura 6. Ejemplo de especificación del producto a generar

en el objeto *features* se enumeran las características del modelo de variabilidad de la Figura 3 que se desean para la calculadora concreta que se quiere producir, y en el objeto *data* se indican los valores usados para sustituir las variables de la plantilla de la Figura 5.

En el último paso del proceso, el motor de derivación delega en el motor de plantillas y el gestor de ficheros para generar el código fuente del producto. Para ello, el motor de plantillas se instancia con la configuración específica de la LPS y comienza a recorrer todos los ficheros de la carpeta de plantillas procesando las anotaciones. Este procesamiento se realiza generando para cada plantilla una función JavaScript que, en caso de encontrar anotaciones evalúa el contenido de la misma, y en caso encontrar texto sin anotaciones lo copia directamente al fichero salida. La evaluación de esta función JavaScript genera en la carpeta del producto un fichero de código fuente con las anotaciones aplicadas.

4. Caso de estudio

La *Graph Product Line* (GPL) es una LPS de algoritmos de grafos propuesta por [17] como un problema estándar para la evaluación de tecnologías LPS. GPL consiste en 15 características relacionadas con los grafos en sí mismos, y 2 características relacionadas con su ejecución. Además, las implementaciones de GPL suelen llevarse a cabo de 3 formas diferentes, como también se describe en [17]. Si bien GPL es una LPS simple, es un buen método de evaluación ya que tiene cierta complejidad difícil de abordar aún diseñándolo desde el principio como una LPS, especialmente cuando se mezclan diferentes configuraciones. Las técnicas que no permiten implementar variabilidad con un nivel de granularidad muy fino requieren hacer ciertas concesiones a la hora de implementar GPL tales como replicar bastante código en varias características, o diseñar la LPS con bastantes características y restricciones artificiales.

En cuanto a nuestra implementación, nos hemos centrado principalmente en la simplicidad y legibilidad del código, y por tanto hemos reimplementado desde cero todo el problema en vez de aprovechar código de implementaciones existentes. Además, hemos encontrado bastantes problemas en las implementaciones que hemos podido analizar, como variantes de productos que no llegaban ni siquiera a compilar por errores en el código generado.

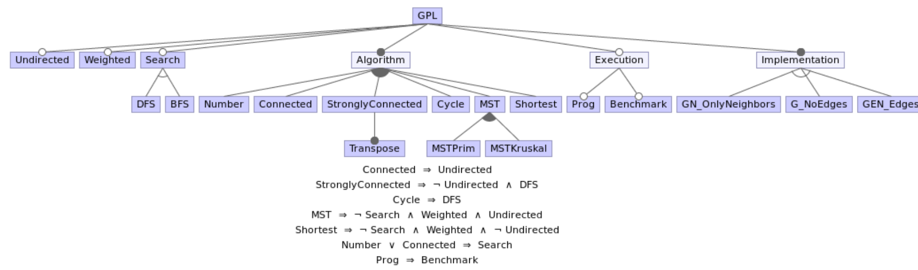

Figura 7. Modelo de características de GPL

Nuestro modelo de variabilidad, que podemos ver en la Figura 7, cuenta con 23 características, siendo 20 de ellas no abstractas, es decir, que tienen influencia en algún fichero de código fuente, y siendo 16 de ellas características hoja. Nuestra herramienta permite, al contrario que otras alternativas, tener características no abstractas que al mismo tiempo no sean hoja, ya que es útil en ciertos casos, por ejemplo para anotar el código común a las características *DFS* y *BFS*, que comparten la característica padre no abstracta *Search*. No existen las características artificiales que vemos en otras alternativas, dado que nuestra herramienta es suficientemente flexible como para proporcionar la granularidad fina necesaria para permitir las tres implementaciones sin requerir restricciones o características artificiales. Podemos ver un ejemplo de esto en el código 8, donde vemos la variación en función de la implementación. Para finalizar, tenemos 7 restricciones que afectan a 12 características.

```
1  graph.getVertices().forEach(v -> {
2
3    /*% if (feature.gNoEdges) { %*/
4    v.getAdjacents().forEach(adj -> edges.add(new Edge(v, adj, v.getWeight(adj.getName()))));
5    /*% } else if (feature.gnOnlyNeighbors) { %*/
6    v.getNeighbors().forEach(nei -> edges.add(new Edge(v, nei.getNeighbor(),
       nei.getWeight())));
7    /*% } else if (feature.genEdges) { %*/
8    v.getNeighbors().forEach(n -> edges.add(n.getEdge()));
9    /*% } %*/
10
11   sets.put(v, new HashSet<IVertex>());
12   // each vertex should be a disjoint set at first
13   sets.get(v).add(v);
14 });
```

Figura 8. Anotación en el código de GPL

Cuando nos comparamos con alternativas compositivas como AHEAD o Feature House quedan patentes los problemas del enfoque compositivo a la hora de implementar la variabilidad *fine grained*. El número de características y res-

tricciones aumenta artificialmente en ambos casos. En concreto, la versión de AHEAD cuenta con 66 características y 32 restricciones (afectando a 41 de las características), mientras que en la versión de Feature House encontramos 38 características y 16 restricciones (afectando a 24 características en este caso). Este mismo problema fue señalado por Kästner et al. [18], cuyo modelo de variabilidad para implementar GPL se acerca mucho más al nuestro, contando con 29 características y 8 restricciones (afectando a 18 características). Parte de las ventajas de nuestra aproximación vienen dadas directamente por la posibilidad de usar las últimas versiones de Java, ya que la ausencia de interfaces, por ejemplo, en el código de AHEAD, es una gran tara.

5. Lecciones aprendidas: Líneas de Producto Software vs *scaffolding*

Hay una clara similitud entre el proceso *scaffolding* y el funcionamiento de una línea de producto software. En ambos casos se parte de ciertos activos de los que, en función de una determinada especificación, se obtiene una salida en forma de código fuente. Los activos son las plantillas anotadas o los componentes reusables, la especificación es una descripción del componente a construir o la selección de características, y la salida se obtiene mediante generación de código o a través del montaje de los distintos componentes. Sin embargo, hay algunas diferencias significativas:

- Generalmente, en una LPS las características se implementan con bloques de código completos que se incluyen o no en los productos finales. En cambio, en *scaffolding* se genera código fuente, ya sea en tiempo de compilación o en tiempo de ejecución.
- Los productos que se generan en una LPS normalmente son productos finales listos para pasar a producción mientras que las herramientas de *scaffolding* están orientadas a ser una ayuda para los desarrolladores.
- En el caso de las LPS es un analista el que genera los productos, dado que simplemente debe elegir las características deseadas entre una lista. En el caso de *scaffolding* suele ser el desarrollador en el que utiliza las herramientas.
- El código fuente generado por las herramientas de *scaffolding* es claro y fácilmente mantenible porque está orientado a desarrolladores, mientras que el código fuente del producto de una LPS acostumbra a ser difícilmente mantenible porque no se espera que sea modificado.
- Las herramientas de *scaffolding* están muy presentes en la comunidad de desarrollo de software y su evolución y uso es muy activo, mientras que las tecnologías de LPS se usan mucho en determinados sectores pero no de manera generalizada.

Después de analizar las diferentes técnicas para implementar LPS, listadas en [2,3,4], hemos detectado una serie de problemas comunes:

- *Lejanía respecto a los lenguajes de desarrollo actuales.* Por una parte, aunque existen herramientas LPS que permiten realizar composición sobre varios lenguajes de programación al mismo tiempo (por ejemplo, FeatureHouse [19] o CIDE [18]), en la práctica sólo se puede utilizar directamente un conjunto muy pequeño puesto que hay que implementar extensiones particulares para cada lenguaje. Por otra parte, algunas de las herramientas sólo funcionan con determinadas versiones del lenguaje de programación en cuestión (por ejemplo, no hemos encontrado ninguna alternativa compositiva que permita usar toda la funcionalidad de las anotaciones de Java disponibles desde la versión 1.5 de 2004).
- *Lejanía respecto a las herramientas de desarrollo actuales.* Muchas tecnologías de implementación de LPS obligan a utilizar herramientas de desarrollo concretas, algunas de ellas ya obsoletas. Es bastante común el uso de la plataforma Eclipse, pero no siempre las herramientas funcionan con la última versión de la misma. Por ejemplo, la herramienta CIDE funciona con Eclipse 3.5 que fue publicado en 2009. En otros casos, el problema se presenta en el momento de hacer funcionar la herramienta. Por ejemplo, hacer funcionar preprocesadores de propósito general (GPP o GNU M4) es realmente complicado.
- *El código del producto generado no es fácilmente mantenible.* El código fuente de los productos generados por las enfoques compositivos acostumbra a no ser mantenible, como ya hemos comentado en la Sección 2. Esto no es un problema en contextos en los que el producto generado por la LPS se despliega sin necesitar cambios, pero en los casos donde no es así modificar el comportamiento de código de un producto generado por una LPS compositiva es una tarea muy complicada e imposible de asumir para un equipo de desarrollo.

Las técnicas de *scaffolding* solucionan estos problemas: están orientadas a las técnicas de desarrollo actuales dando soporte a múltiples lenguajes de programación y funcionando con múltiples entornos de desarrollo, y el código fuente generado es fácilmente legible y mantenible. Sin embargo, también presentan problemas:

- *Falta de formalismo.* Cada tecnología de *scaffolding* define un modelo *ad-hoc* para especificar los componentes que se generan.
- *Elevada complejidad en la definición de las plantillas.* Muchos autores se posicionan en contra del uso de preprocesadores por la problemática de las anotaciones *fine-grained* [20,21] que hacen extremadamente complejo el mantenimiento del código de las plantillas. Pese a ello, los preprocesadores se mantienen como la alternativa más popular en el ámbito industrial para implementar LPS [5,22,23].

Nuestra propuesta se sitúa en un punto intermedio entre los dos campos: usamos *scaffolding* para implementar Líneas de Producto Software de forma que solventemos ciertos problemas en las técnicas de implementación de LPS actuales y haciendo más atractiva su adopción para los equipos de desarrollo de software, pero sin perder los formalismos y conceptos detrás del desarrollo de LPS.

6. Conclusiones y trabajo futuro

En este artículo hemos mostrado nuestra propuesta para crear un motor de derivación LPS basado en *scaffolding*. Nuestra herramienta, por una parte, se apoya en los formalismos de líneas de producto software, y por otra parte proporciona las ventajas de *scaffolding* al generar código fuente fácilmente mantenible y permitir utilizar tecnologías modernas y ampliamente soportadas. En el artículo se describen las decisiones de diseño que se tomaron para construir el motor de derivación, su arquitectura, y el proceso en el que se basa su funcionamiento. Además, se muestra un caso de estudio con una LPS de cierta complejidad en el que se ve que nuestra propuesta es mejor que otras alternativas en términos de complejidad de la LPS y de calidad del código del producto resultante.

Como trabajo futuro tenemos planificado mejorar la funcionalidad de análisis del modelo de variabilidad y del código fuente anotado para obtener mejores indicadores de la calidad del código de la LPS, mejorar la generación de código para permitir la generación de múltiples ficheros a partir de una plantilla y no sólo la generación de un fichero por plantilla con el mismo nombre, y estudiar la integración del motor de derivación en procesos de desarrollo de software permitiendo que las modificaciones en los componentes de la LPS se trasladen fácilmente a los productos generados.

Referencias

1. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, Berlin/Heidelberg (2013) 308 pages, ISBN 978-3-642-37520-0.
3. Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Saake, G.: An overview on analysis tools for software product lines. In: Workshop on Software Product Line Analysis Tools (SPLat), New York, NY, USA, ACM (September 2014) 94–101
4. Databases and Software Engineering Workgroup — University of Magdeburg: Tools for feature-oriented software development. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/fosd-tools/` (Consultado el 02/07/2016).
5. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wasowski, A.: A survey of variability modeling in industrial practice. In: The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013. (2013) 7:1–7:8
6. Weiss, D.M.: The product line hall of fame. In: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. (2008) 395
7. van der Linden, F.: Software product families in europe: The esaps & café projects. IEEE Software **19**(4) (2002) 41–49
8. do Carmo Machado, I., Santos, A.R., Cavalcanti, Y.C., Trzan, E.G., de Souza, M.M., de Almeida, E.S.: Low-level variability support for web-based software product lines. In: The Eighth International Workshop on Variability Modelling of

Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014. (2014) 15:1–15:8
9. Urli, S., Blay-Fornarino, M., Collet, P.: Handling complex configurations in software product lines: a tooled approach. In: 18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014. (2014) 112–121
10. Brisaboa, N.R., Cortiñas, A., Luaces, M.R., Pol'la, M.: A Reusable Software Architecture for Geographic Information Systems based on Software Product Line Engineering. In: Proceedings of the 5th International Conference on Model & Data Engineering (MEDI 2015), Springer (2015) 320–331
11. Brisaboa, N.R., Cortiñas, A., Luaces, M.R., Pol'la, M.: Gisbuilder: a framework for the semi-automatic generation of web-based geographic information systems. In: Proceedings of the 20th Pacific Asia Conference on Information Systems (PACIS 2016). (2016) (Pendiente de publicación).
12. Kästner, C.: Cide: Virtual separation of concerns. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/` (Consultado el 02/07/2016).
13. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008. (2008) 311–320
14. Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G.: Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering **18**(4) (2013) 699–745
15. Pleumann, J., Yadan, O., Wetterberg, E.: Antenna. an ant-to-end solution for wireless java. `http://antenna.sourceforge.net/` (Consultado el 27/04/2016).
16. Munge Development Team: Munge: Simple Java preprocessor. `https://github.com/sonatype/munge-maven-plugin` (Consultado el 27/04/2016).
17. Lopez-Herrejon, R.E., Batory, D.S.: A standard problem for evaluating product-line methodologies. In: Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings. (2001) 10–24
18. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D.S.: Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In: Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings. (2009) 175–194
19. Apel, S., Kästner, C., Lengauer, C.: FEATUREHOUSE: language-independent, automated software composition. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. (2009) 221–231
20. Spencer, H., Collyer, G.: #ifdef considered harmful, or portability experience with C news. In: USENIX Summer 1992 Technical Conference, San Antonio, TX, USA, June 8-12, 1992. (1992)
21. Favre, J.: Understanding-in-the-large. In: 5th International Workshop on Program Comprehension (WPC '97), May 28-30, 1997 - Dearborn, MI, USA. (1997) 29–38
22. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. (2010) 105–114
23. Kästner, C., Apel, S.: Virtual separation of concerns - A second chance for preprocessors. Journal of Object Technology **8**(6) (2009) 59–78