Departamento de Computación.
Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires

"Hacia un nuevo enfoque para el modelado arquitectónico de familias de productos"

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Perez Vultaggio, Laura

Directores:

Lic. Santiago Ceria
Dr. Fernando Asteasuain

Hacia un nuevo enfoque para el modelado arquitectónico de familias de productos

La especificación arquitectónica de una familia de productos tiene como objetivo describir a alto nivel las principales interacciones entre los compontes de los productos que forman la familia. Es fundamental en dicha especificación establecer claramente las funcionalidades que son comunes a toda la familia y aquellas que varían de producto a producto.

Si bien han surgido aproximaciones para el modelado arquitectónico de familias de productos, la comunidad de Ingeniería de Software ha detectado algunas limitaciones. La primera está relacionada con el concepto de variabilidad, el cual no ha sido tratado como un aspecto de primer nivel, causando así debilidades en la arquitectura resultante. Una segunda limitación está relacionada con el nivel de abstracción necesario en una descripción arquitectónica. Muchas aproximaciones terminan utilizando artefactos de software cercanos al código, lo cual puede llevar a tomar decisiones de manera prematura. Finalmente, la falta de una notación homogénea para describir la arquitectura también es señalado como un problema.

Dado este contexto en este trabajo se presenta un primer paso hacia un enfoque declarativo para el modelado arquitectónico en busca de solucionar los problemas detectados. La propuesta está basada en el lenguaje declarativo y gráfico FVS (Feather Weight Visual Scenarios). **FVS posee exhibe propiedades que lo hacen atractivo en un dominio arquitectónico**. Cuenta con una notación sólida, flexible y compacta, con una semántica clara, y es lo suficientemente expresivo. Además, es particularmente útil para explorar comportamiento en etapas tempranas, propiedad necesaria en dominios arquitectónicos.

Palabras claves: Arquitectura de Software, Familia de Productos, Especificación de comportamiento.

Towards a novel approach for modeling software product lines architecture.

Software product lines architecture describes high-level interactions between the main components of the products of the family. In the architecture specification it must be clear which features are common to the family and which features varies between products.

Several known problems negatively affect the specification of software product lines architecture. One of them is related with a key concept known as Internal Variability. The internal variability of components is generally neglected when describing the architecture of a family of products. The lack of a homogenous architectural notation is also pointed out as a significant difficulty for achieving an architectural specification. A third problem arises when the high level abstractions needed at the architectural level are mixed with low-level artifacts as lines of code.

This works aims to take a step towards finding a solution to the mentioned problems. It explores FVS (Feather Weight Visual Scenarios) as a declarative language for specifying software product lines architectures in early stages of software development. FVS holds a solid, simple and flexible notation, with a clear and formal semantics. In addition, it is particularly suitable to explore behavior in early stages, a key feature for any language focused on architectural descriptions.

Keywords: Software architectures, Software Product Lines Architecture, Behavioral specifications.

ÍNDICE GENERAL

1. Introducción 6

1.1 Objetivos 6

1.2 Estructura 7

2. Conceptos preliminares 9

2.1 Camino hacia Arquitecturas de Software 9

2.2 Arquitecturas de Software: Conceptos Básicos 10

2.2.1 Arquitectura y especificación de Requerimientos no Funcionales 11

2.2.2. Vistas Arquitectónicas 13

3. Familia de Productos 15

3.1. Surgimiento de las familias de productos 15

3.2. ¿Qué es una familia o línea de productos de software (LPS)? 16

3.2.1 Modelos y desarrollo de Familias de Productos 18

3.2.2 Beneficios de Familias de Productos 20

3.2.3 Fundamentos de las Familias de Productos 21

3.3 Variabilidad 23

3.3.1 Variabilidad y Familias de productos 24

4. Arquitecturas de Familias de Productos 26

4.1 Modelado de características o Features 26

4.2 Arquitectura con Componentes y Conectores 30

4.3 Plastic Partial Components 31

4.3.1 Plastic Partial Components: Descripción e idea general 32

4.3.2 PPC: Metamodelo 34

4.3.3 PPC: Ejemplo de aplicación Sistema Cajero 35

4.4 Paradigma de Aspectos aplicado al modelado de SPL 37

5. Modelado Arquitectónico de SPL: Algunos problemas y limitaciones 39

5.1 Variabilidad interna 39

5.2 Mantener el nivel de abstracción 40

5.3 Notaciones completas, claras y homogéneas 40

5.4 Conclusiones 41

6. Propuesta de modelado para Arquitecturas de Familias de Producto 42

6.1 FVS: Feather Weight Visual Scenarios 43

6.2. Modelado arquitectónico en FVS 45
6.2.1 El Caso de estudio E-SHOP 45
6.3 Discusión y análisis del caso de estudio 52
7. Trabajo futuro 55
8. Resumen y Conclusiones 56
9. Referencias 57

ÍNDICE DE TABLAS

Tabla 1: Principales conceptos por década 10
Tabla 2: Ejemplo de restricciones entre features 30

ÍNDICE DE FIGURAS

Figura 1: Escenarios de atributos de calidad 12
Figura 2 - Vista Arquitectonica 14
Figura 3- Ejemplo de Componentes comunes en teléfonos móviles [44] 17
Figura 4- Enfoque del doble ciclo de vida aplicado a las líneas de producto 18
Figura 6 - Evolución de la reutilización de software 22
Figura 11 - Meta Modelado de PCC 34
Figura 12 – Modelo de Features 36
Figura 13: Modelado del PPC. 37
Figura 14- Elementos Básicos de FVS 44
Figura 15 - Ejemplo de una regla en FVS 45
Figura 16- Modelo de características para la familia de productos E-SHOP. 46
Figura 17 - Reglas FVS para las restricciones R1 y R2. 47
Figura 18 -Reglas FVS para las restricciones R3 sobre el modo degradado. 48
Figura 19- Reglas FVS para la restricción R4. 48
Figura 20 - Esquema componente Comunicación E-SHOP 49
Figura 21 - Regla FVS exhibiendo la variabilidad presente al enviar mensajes 50
Figura 22 - Nuevo comportamiento para el componente Comunicación. 50
Figura 23 - Un producto implementado seguridad baja. 51
Figura 24 - Nuevo comportamiento para el componente Comunicación. 51
Figura 25 - Regla de Variabilidad Interna para Manejo de Actualización de Saldos 52
Figura 26 - Modelando comportamiento de parte del componente Saldo. 52

1. Introducción

Uno de los conceptos claves dentro de la Ingeniería de Software es la utilización de modelos y abstracciones para poder razonar, explorar y especificar el comportamiento esperado del sistema a construir. En este sentido, el diseño de Arquitecturas de Software [1] constituye uno de los pilares fundamentales para lograr tales objetivos.

Una especificación arquitectónica se enfoca en el diseño y modelado de sistemas con un alto nivel de abstracción, revelando las principales interacciones entre los artefactos de software involucrados, dejando de lado detalles de carácter implementativo de bajo nivel. La importancia del diseño arquitectónico radica en que puede ser visto como el nivel de diseño y modelado que une los requerimientos con el código [2].

Dentro del diseño arquitectónico, el campo denominado Familia de Productos ha adquirido especial interés en el sector industrial [3]. Una Familia de Productos o SPLE (Software Product Line Engineering) [4] se basa en la idea que aplicaciones en un mismo dominio comparten funcionalidad, por lo que es fundamental desarrollar esa parte en común una única vez, y luego reutilizarla en cada uno de los diferentes productos de la familia. Por lo tanto, es crucial en el diseño arquitectónico de SPLE poder modelar tanto la funcionalidad común de cada uno de los productos, como así también la variabilidad entre los productos [3, 7, 8].

Si bien han surgido aproximaciones para el modelado arquitectónico de SPLE [5, 6, 7 ,8], la mayoría de las mismas sufre de distintas limitaciones. Por ejemplo, el concepto de variabilidad entre los productos no ha sido tratado como un aspecto de primer nivel, causando así debilidades en la arquitectura resultante [3,5]. Luego, dada la importancia del área surge la necesidad de profundizar su diseño arquitectónico.

Dado este contexto, esta tesis pretende llevar a cabo un profundo estudio de las principales opciones para el modelado arquitectónico de SPLE, buscando analizar ventajas y desventajas de cada una, para luego presentar un enfoque con el objetivo de superar algunas de las limitaciones detectadas.

1.1 Objetivos

El presente trabajo tiene como principal objetivo analizar ventajas y desventajas de las distintas aproximaciones basadas en el modelado arquitectónico de

una familia de productos, para luego presentar un nuevo enfoque en pos de consolidar una nueva alternativa que constituya un primer paso para lograr resolver problemas detectados en el dominio.

Para cumplir con dicho objetivo se recorre el camino que la rama de “Familias de Productos” o “Líneas de Productos “(PLA) ha transitado hasta el día presente. Esto permitirá mostrar objetivamente aquellos que permitieron lograr avances en el desarrollo de software. El estudio involucra un análisis representativo del universo de las familias de productos o líneas de productos. Junto con la comparación de los diferentes modelos, se exponen de manera intuitiva las ventajas y desventajas de cada uno de ellos, las cuales dependen muchas veces del contexto o dominio en el que se ejecuten.

Un objetivo secundario es que los resultados puedan ser alcanzados por diferentes tipos de audiencias, ya que una arquitectura debe ser analizada no sólo por desarrolladores, sino también por los distintos stakeholders, que no tienen porqué tener conocimientos técnicos. Relacionado con este punto, se buscará mostrar una visión simple para interpretar y entender el tema, mostrando un paralelismo con el desarrollo de productos no estrictamente relacionados al software. La idea es que pueda ser reutilizado por lectores con diferentes perfiles y necesidades.

1.2 Estructura

La presente tesis cuenta con la siguiente estructura.

En el capítulo 2 se presentan conceptos preliminares sobre Arquitecturas de Software, sobre los cuales después se elabora el contenido de la investigación resultante.

El capítulo 3 está enfocado en el concepto de familia o línea de productos de software, analizando cómo se conforman y cómo se analizan o desglosan. Se busca entender cuál fue la necesidad de tener que incluir esta idea a la hora de desarrollar piezas de software, recorriendo el camino evolutivo y conceptual de las SPLE. Este análisis sientas las bases para avanzar hasta conceptos más avanzados que permitirán comprender el porqué de diferentes modelos, estrategias, implementaciones y enfoques del tema.

El capítulo 4 está abocado al modelado arquitectónico de Familias de Productos. Se analizan en el mismo las principales variantes de cómo posicionarse a la hora de iniciar un modelado arquitectónico, mostrando ejemplos que permitan ilustrar sus diferentes motivaciones u objetivos. Lógicamente, esto produce que cada alternativa tenga sus propias ventajas o desventajas, dependiendo muchas veces del contexto del dominio del problema.

En el capítulo 5 se exponen limitaciones detectadas en los enfoques actuales, como ser la falta del modelado de variabilidad interna de componentes, la falta de una notación homogénea y simple, o la necesidad de contar con mayor poder de abstracción para poder razonar en un mundo de arquitecturas de software.

En el capítulo 6 se propone un nuevo enfoque al modelado arquitectónico, como un primer paso hacia un enfoque que logre superar los problemas mencionados. Dicho enfoque está basado en el lenguaje declarativo FVS (FeatherWeight Visual Scenarios), utilizado para la especificación temprana de comportamiento.

Finalmente, a modo de cierre de la presente tesis de licenciatura, los capítulos 7 y 8 dedican algunas líneas a más a una conclusión general del trabajo hecho y planteos o propuestas futuras para continuar en nuevos trabajos o tesis.

2. Conceptos preliminares

El presente capítulo pretender dar un breve panorama introductorio al área de Arquitecturas de Software, como para presentar un marco general que de sustento al resto de la tesis. En otras palabras, es conveniente repasar algunos aspectos de la Ingeniería de Software y las implicancias del diseño y la arquitectura de software. En la Sección 2.1 se describe brevemente el camino dentro de la Ingeniería de Software hasta llegar a la noción de Arquitecturas de Software mientras que en la Sección 2.2 se presentan conceptos inherentes a la arquitectura de software.

2.1 Camino hacia Arquitecturas de Software

Típicamente, la Ingeniería de Software involucra como concepto base la utilización de modelos y abstracciones para poder razonar, explorar y especificar el comportamiento esperado del sistema a construir. Es así que el diseño de Arquitecturas de Software [1] se transforma en la herramienta fundamental para hacerse de los medios necesarios y lograr tales objetivos.

Desarrollar una buena arquitectura de software para un sistema dado, y más aún si se trata de un sistema complejo, es crítico a la hora de cumplir los principales objetivos de un sistema.

Una pieza de software está conformada por diferentes "artefactos" los cuales interactúan de manera particular entre sí. Aquí es que donde entra en juego la especificación arquitectónica, la cual se enfoca en el diseño y modelado de sistemas con un alto nivel de abstracción, pudiendo revelar las principales interacciones entre los artefactos involucrados, pero dejando de lado detalles de carácter implementativo de bajo nivel. Se podría decir que el diseño arquitectónico se convierte en un puente entre dos mundos: los requerimientos y el código [2]. Esto resulta en un valor no menor; por el contrario es una característica de mucha importancia, ya que surge como un tercer nivel de abstracción.

Bien vale la pena entonces mirar años atrás, como para comprender mejor los orígenes o motivaciones de la Ingeniería de Software y así entender mejor también los paralelismos que se pueden encontrar en la Ingeniería de Software actual y sus diferentes ramas.

Durante los años '70s, los investigadores invirtieron mucho de su tiempo estudiando *Diseño de Software.* Esto no fue casualidad, sino que simplemente fue una

necesidad que se trataba de suplir en respuesta al desarrollo de software a gran escala que se masificó en la década de los años '60s. La premisa era clara: la implementación y el diseño pertenecían a mundos separados, y para este último era necesario tener notificaciones particulares, herramientas y técnicas apartes, y por lo tanto ser una entidad distinguida [3].

Luego en los 80's, la Ingeniería de Software comenzó a mover su foco de interés ya no tanto en el diseño específicamente sino más bien en un concepto más amplio del proceso de diseño, con una visión más relacionista entre el diseño en sí, los procesos software y su mantenimiento. Una muestra de esta visión más amplia ha sido la de los lenguajes de software incluyendo notaciones y técnicas del desarrollo de diseño de software. De hecho, esta integración ha producido que los límites entre implementación y diseño, de a poco comiencen a confundirse.

Ya luego en la década de los '90s fue directamente la década de la "*Arquitectura de Software".* Usamos el término "arquitectura" en contraposición de "diseño" que evoca nociones de codificación, de abstracción, de estándares, de entrenamientos formales (de arquitectos de software) y de estilos [4]. La tabla 1 resume estas ideas.

Decada	Concepto
70´s	Diseño de Software
80´s	Integración del Diseño y Procesos
90´s	Arquitectura de Software

Tabla 1: Principales conceptos por década

En la próxima sección se describen algunos conceptos fundamentales para la arquitectura de software.

2.2 Arquitecturas de Software: Conceptos Básicos

Existen dos partes fundamentales en una descripción arquitectónica. Por un lado, debe existir una especificación de los objetivos a cumplir. Es imposible construir algo sin saber qué es lo que se quiere. Por otro, debe haber una manera amigable y entendible de mostrar a todos los interlocutores necesarios la descripción arquitectónica desarrollada. A continuación se describen cada una de estas partes.

2.2.1 Arquitectura y especificación de Requerimientos no Funcionales

El foco principal de las especificaciones de software estuvo sobre los requerimientos funcionales (aquellos que describen qué es lo que debe hacer el sistema [11-13]), generando técnicas como Requerimientos basados en metas (en inglés GORE, por Goal Oriented Requierements Engineering[14]) o Casos de Uso [16], por nombrar sólo dos entre las opciones disponibles.

Sin embargo, otro tipo de requerimientos conocidos como no funcionales han ido obteniendo cada vez un papel más preponderante, incluso hasta ser considerados de la misma importancia que los funcionales [15]. Este tipo de requerimientos se enfoca en restricciones sobre el comportamiento general del sistema, incluyendo conceptos cercanos a la calidad del software [15]. Los ejemplos más conocidos son: performance, disponibilidad, seguridad, usabilidad, modificabilidad y testeabilidad [17]. Es crítico para toda descripción arquitectónica incluir una especificación para todos los requerimientos no funcionales.

Una de las técnicas más conocidas para la especificación de requerimientos no funcionales es mediante la utilización de **Escenarios de Atributos de calidad** [17]. Uno de los roles más importantes de esta técnica es proveer una cuantificación medible y *testeable* sobre un atributo de calidad. Por ejemplo, ¿un sistema que se adapta a distintas interfaces de usuarios pero funciona sobre una única plataforma, es modificable? La respuesta depende sobre qué se desea que el sistema quiera ser modificado. Si se tiene en cuenta la adaptación a distintas interfaces de usuarios, el sistema cumple con el atributo planteado. Sin embargo, si se deseaba poder cambiar fácilmente la plataforma sobre la cual corre el sistema, el objetivo de modificabilidad no se ha cumplido. Todos los atributos de calidad afectando la disponibilidad, performance, seguridad deben quedar correctamente expresados a través de escenarios de calidad. A través de los mismos se notará cuán rápido un sistema debe procesar una transacción, cuánto tiempo puede estar caído, cuánto tardará en recuperarse, etc. También es importante señalar que es común que muchos atributos de calidad entren en conflicto. Dos ejemplos clásicos son los siguientes [17]. Un caso está dado por los atributos de seguridad y disponibilidad. El sistema más seguro tiene una cantidad mínima de puntos de falla, mientras que un sistema con alta disponibilidad cuenta con múltiples puntos de falla (típicamente un conjunto de procesos y procesadores redundantes donde la caída de uno no afectara a los demás). El segundo caso es entre portabilidad y performance. Las principales técnicas para hacer portable un sistema introducen *overhead* en el procesamiento perjudicando la

performance final del sistema. La especificación de atributos mediante escenarios ayuda a exponer y resolver los conflictos que surjan.

La especificación de un atributo de calidad consta de seis partes [17]:

1. Fuente del estímulo: La entidad que genera el estímulo (una persona, un sistema externo, etc.)
2. Estímulo: El estímulo es evento que necesita ser atendido cuando llega al sistema.
3. Ambiente: Describe el contexto en que se encuentra el sistema al momento de recibir el estímulo. Puede ser un ambiente con sobrecarga, sin conexión, etc.
4. Artefacto: Entidad que recibe el estímulo. Puede ser el sistema como unidad, como también una sub-componente, como por ejemplo un subsistema de comunicación.
5. Respuesta: Es la actividad llevada a cabo por el sistema cuando llega el estímulo.
6. Medición: Expresa una cuantificación sobre la respuesta para que la misma pueda ser testeada. Por ejemplo, el tiempo necesario para procesar una transacción en un sistema de compras online.

La figura 1 [17] muestra gráficamente los elementos de una especificación de un atributo de calidad a través de escenarios.

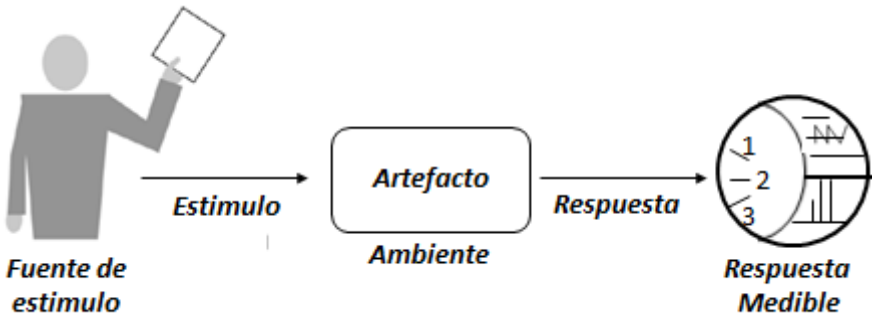

Figura 1: Escenarios de atributos de calidad

A modo de ejemplo, se muestra a continuación una posible especificación de escenario de calidad para el atributo de performance:

1. **Fuente del estímulo: Usuarios**
2. **Estímulo: Inician transacciones**
3. **Ambiente: Normal**

4. **Artefacto: Sistema**

5. **Respuesta: Se procesan todas las transacciones.**

6. **Medición: La latencia promedio para llevar a cabo las transacciones es de 2 segundos.**

2.2.2. Vistas Arquitectónicas

El resultado de la descripción arquitectónica se muestra bajo el concepto de vistas [8]. Cada vista arquitectónica representa una mirada particular sobre el comportamiento del sistema. En [18] se agrupan los distintos tipos de vistas en tres:

- Vista de Módulos: Se puede decir que se corresponde con la estructura estática del sistema. Como mínima, esta vista muestra cómo el código de un sistema está dividido en varias partes, cómo estas partes pueden combinarse y relacionarse, y qué tipo de restricciones posee cada una. Las elecciones de modularización influyen en cómo la modificación de una parte del sistema afecta a otras. Es decir, influyen en la modificabilidad, portabilidad y reuso del sistema.
- Vista de Componentes y Conectores: Esta vista muestra a las entidades del sistema en ejecución y sus posibles interacciones. Por ejemplo, la comunicación entre un cliente y un servidor puede estar dado por un complejo protocolo de comunicación incluyendo permisos y nociones de encriptación.
- Vista de *Alocación*: En esta vista se muestra el mapeo entre los componentes al hardware que los soporta. A través de la misma se puede observar por ejemplo, en qué servidores estarán corriendo los distintos procesos, y cómo se llevara a cabo la comunicación físicamente.

Muchas veces estas vistas pueden combinarse. Por ejemplo, una vista híbrida de módulos y componentes y conectores puede mostrar cómo está dividida la funcionalidad de dos componentes y al mismo tiempo reflejar cómo es el flujo de comunicación entre los procesos que los representan. Otro ejemplo posible podría ser una vista combinada entre Componentes y Conectores y Alocación, mostrando la interacción en ejecución de dos componentes así como la ubicación física de los servidores donde están corriendo dichos componentes.

A continuación se muestra un simple ejemplo de lo que se podría considerar uno de los entregables que se pueden obtener de una Arquitectura de Software. En la Figura 2 se puede observar una vista arquitectónica de un sistema de voz-IP

(**InstaVOIP**). La misma expone cómo los diferentes componentes del sistema están organizados en módulos y/o por capas de abstracción, cuáles son sus responsabilidades. También deja claro qué componentes se comunican entre sí, e incluso en algunos casos provee una noción de cómo o por medio de qué herramienta se estarán comunicando.

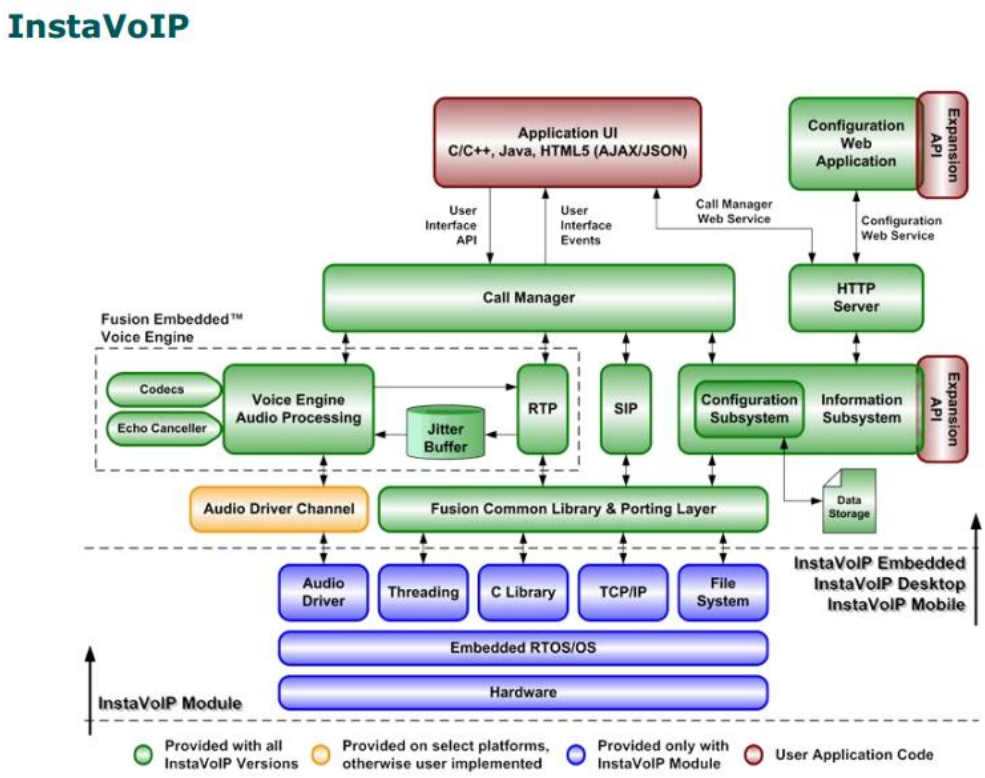

Figura 2 - Vista Arquitectónica

Habiendo dado un marco general de la Ingeniería de Software y de Arquitecturas de Software, en el siguiente capítulo se introduce la temática de familia de productos.

3. Familia de Productos

En este capítulo se verá cómo se conforman, cómo se analizan o desglosan las familias de productos o SPLE (Software Product Line Engineering), entendiendo cuál fue la necesidad de tener que incluir esta idea a la hora de desarrollar piezas de software, recorriendo su camino evolutivo y conceptual.

Con conceptos detrás como variabilidad, componentes, *features*, similitudes y reuso en mente es que nacen las ***Familias o Líneas de Productos o SPLE*** *(Software Product Line Engineering) [7]*, que intentan ya no producir un único producto, sino un sistema o framework que permita gestionar y administrar eficiente y eficazmente las variaciones presentes entre productos.

Dado que SPLE se basa en la idea que aplicaciones en un mismo dominio comparten funcionalidad, es entonces fundamental desarrollar esa parte en común una única vez, y luego reutilizarla en cada uno de los diferentes productos de la familia. Luego, es crucial en el diseño arquitectónico de SPLE poder modelar tanto la funcionalidad común de cada uno de los productos, como así también la variabilidad entre los productos [6, 8, 9].

A continuación se presenta en detalle la noción de familia de productos de la siguiente forma. En la Sección 3.1 se expone el contexto que dio lugar al surgimiento de las familias de productos. La Sección 3.2 profundiza el concepto de familia de productos en sí mientras que la Sección 3.3 presenta nociones de variabilidad.

3.1. Surgimiento de las familias de productos

Para entender el concepto de familia de productos es conveniente exponer el contexto y modo en qué surgieron. Tal análisis es presentado a continuación.

Era muy común durante la década de los 70s y 80s que luego de entregado un producto de software a un cliente, la empresa desarrolladora se tope con problemáticas respecto al manejo de versiones y la evolución del producto. En particular, muchas veces se contaba con código confuso o sin documentación, lo cual lo hace muy complejo de extender con variaciones, llevando a un proceso propenso a errores.

De hecho, esta problemática no afectaba únicamente al desarrollo de software. También le pasaba a las industrias de electrodomésticos, automóviles, telecomunicaciones, etc. Es decir, estaba presente en todo ámbito en donde un producto en serie admite variaciones [5].

En este punto comenzó a surgir entonces la idea de poder ganar en valor de esas diferencias y similitudes entre una familia de productos, para lograr una mejor adaptación a diferentes clientes. Se comenzó a gestionar entonces, el re-uso de código o componentes. Sin embargo, el reuso no estaba planificado desde el comienzo del proyecto como estrategia, sino que se aplicaba de manera *adhoc* cuando se reconocía una oportunidad. Por lo tanto ocurría que el esfuerzo para reutilizar material era más costoso que la no reutilización directamente.

Para resolver esta situación surgió el concepto formal de familias de productos o líneas de productos. La idea no es producir un único producto sino un sistema que permita gestionar y administrar las variaciones de un mismo producto, haciéndolo claro y de forma eficaz. La administración de las variaciones y similitudes implica no solo un aspecto técnico, también involucra procesos y organización:

- El proceso delimita el dónde y cuándo se debe realizar el esfuerzo de reutilizar. Esta no es una decisión simple, y se pude fracasar justamente por no haber tomado aquí la decisión correcta.

- Con respecto a la organización, no es trivial tampoco el componente humano. Aquí el grupo de trabajo deberá mover su foco del "producto" a "familias de productos", lo cual implica una re-organización del grupo, a nivel *ownership* de módulos, responsabilidad ante errores de los módulos, manejo de solicitudes de cambio de los clientes, etc.

3.2. ¿Qué es una familia o línea de productos de software (LPS)?

Para dar una definición o representación sencilla de entender, se podría decir que se trata del ensamblaje de partes de software previamente elaboradas. Está inspirada en procesos usados por los sistemas físicos (computadoras, autos, etc.), y en la reutilización de software[1], asumiendo una ingeniería de productos.

Algunas de las definiciones de autores reconocidos en la materia son las siguientes:

- *"Las líneas de productos de Software (o SPL) se refieren a técnicas de ingeniería para crear un portafolio de sistemas de software similares, a partir de un conjunto de activos de software, usando un medio común de producción" [19]*

[1] *Reutilización de software es el proceso de crear sistemas de software partir de software existente, en el lugar de desarrollarlo desde el comienzo"[21]*

➢ *"Línea de Productos de Software consiste de una familia de sistemas de software que tiene una funcionalidad común y alguna funcionalidad variable:*

- *La Funcionalidad común descansa en el uso recurrente de un conjunto de activos reutilizables (requisitos, diseños, componentes, servicios web, etc.)*
- *Los activos son reutilizados por todos los miembros de la familia." [20].*

Para ilustrar estos conceptos a continuación se presenta un ejemplo muy claro y cotidiano, extraído de [44]. El dominio del ejemplo son los teléfonos móviles o celulares (ver figura 3). El ejemplo permite ilustrar cómo en cuestiones cotidianas está presente el concepto de LPS o Familias de Productos.

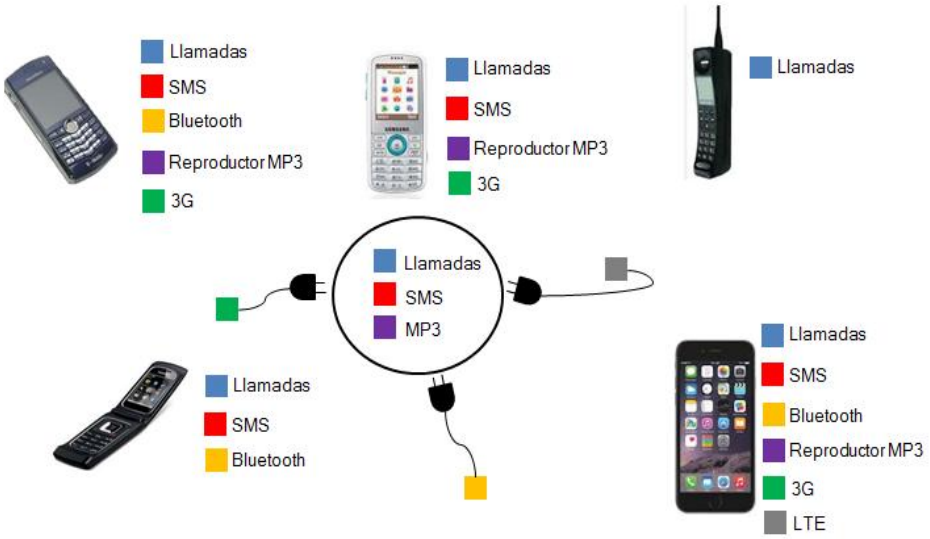

Figura 3- Ejemplo de Componentes comunes en teléfonos móviles [44]

Como se observa en la figura 3, la familia de productos de teléfonos móviles tiene características comunes y variables. Las comunes, como por ejemplo, el servicio de llamadas o SMS, se traducen en artefactos comunes como un diseño arquitectónico común. Toma la forma de un conjunto de componentes reutilizables, capacidades, servicios y tecnologías comunes. Al mismo hay aspectos variables, como por ejemplo funcionalidades como Bluetooth, 3G o LTE, que establecen diferentes comportamientos y funciones.

Es aquí donde aparece este primer punto o palabra clave "Variaciones o Variabilidad". De algún modo este concepto es la base de la concepción e implementación de una LPS o PLA el cual será tratado en profundidad en la Sección 3.3. del presente capítulo.

3.2.1 Modelos y desarrollo de Familias de Productos

En sí mismo al desarrollar una SPL no se intenta construir una aplicación, sino más bien una familia de ellas. Este enfoque supone y obliga a un cambio respecto a un desarrollo orientado a un único producto software, ya que implica moverse a un desarrollo de varios productos que contienen unas características comunes, formando una familia de productos. De esta necesidad nacen varios modelos, la mayoría de ellos basados en el paradigma de SPLE (Software Product Line Engineering), el cual está dividido en dos procesos: la Ingeniería de dominio y la Ingeniería de Aplicación.

Como se explica en [41], la Ingeniería de Dominio se centra en el desarrollo de elementos reutilizables que formarán la familia de productos, identificando las partes comunes y variables de la familia. La Ingeniería de Aplicación se centra en el desarrollo de productos individuales, pertenecientes a la familia de productos y que satisfacen un conjunto de requisitos y restricciones expresados por un usuario específico, reutilizando, adaptando e integrando los elementos reutilizables existentes y producidos en la ingeniería de dominio.

La Figura 4 [23,41], muestra el proceso total de los dos enfoques (Ingeniería de Dominio e Ingeniería de Aplicación) en la que se observa el doble ciclo de vida aplicado a las SPL.

-

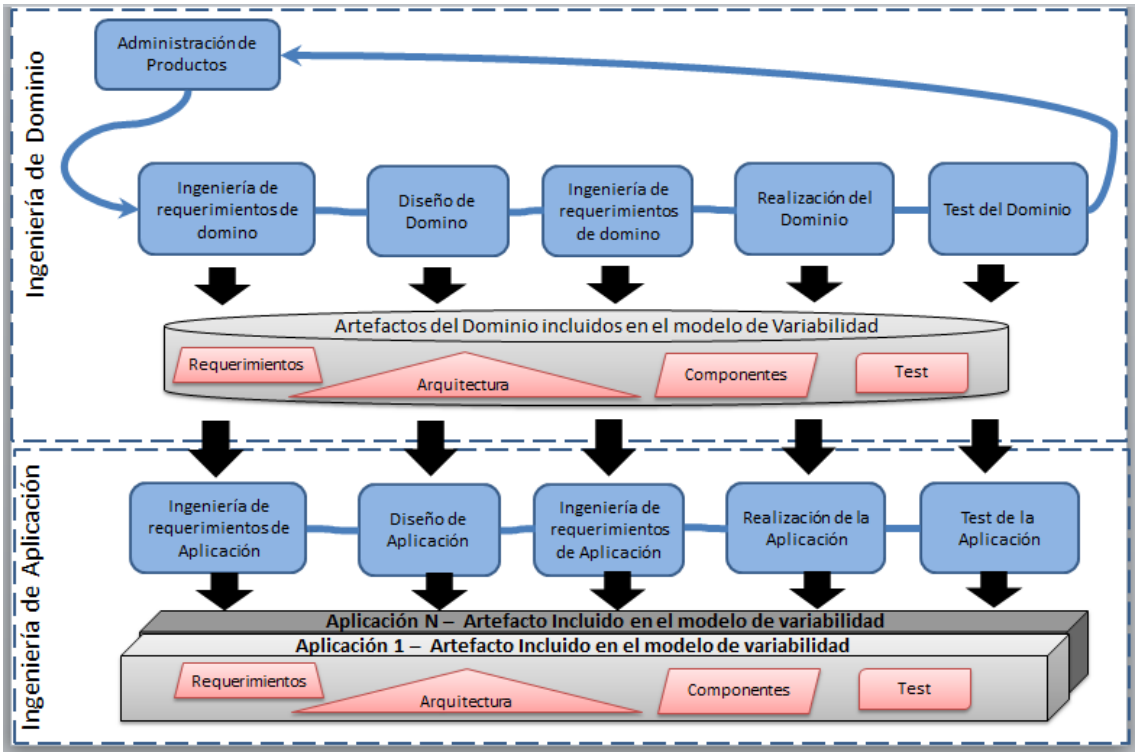

Figura 4- Enfoque del doble ciclo de vida aplicado a las líneas de producto

Otro enfoque del desarrollo del paradigma SPLE, es el propuesto por el Software Engineering Institute (SEI)[2]. Bajo este enfoque se podría representar a una Línea de Productos de Software a través de conceptos como Activos de Software, Decisores, Productos de Software, y Producción. La figura 5 ilustra el esquema propuesto.

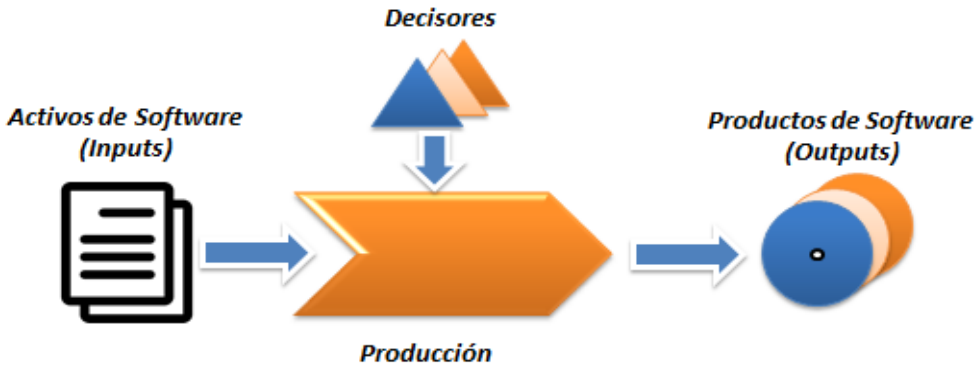

Figura 5 – Modelo Básico de Línea de Productos – Modelo SEI

Del modelo y análisis basado en [5] se identifican claramente cuatro procesos:

- **La Entrada:** Activos de Software.
  - Son básicamente los elementos o componentes que son la base del desarrollo de una pieza de software, y que en este caso representan de manera precisa lo que los productos de la familia deben hacer. Ejemplos de activos de software son: requisitos, diseños, casos de pruebas, etc.
- **El Control:** Modelos de Decisión y Decisiones de Productos
  - Los modelos de decisión están encargados de detectar la variabilidad u opcionalidad de los productos.
  - Luego, se puede describir a un producto como un conjunto de decisiones.
- **El Proceso de Producción:**
  - Se establecen las configuraciones o mecanismos necesarios para componer los productos en base a los activos de entrada.
  - Utilizando las decisiones del producto se determina qué activos de entrada utilizar y cómo configurar los puntos de variación.
- **La Salida:** Productos de Software

[2] http://www.sei.cmu.edu/

- Simplemente son todos los productos que se producen como salida del sistema.

3.2.2 Beneficios de Familias de Productos

De los beneficios más marcados de las familias de productos se pueden mencionar:

- Menores tiempos de entrega de los productos
- Reducción de costos de ingeniería
- Menor tamaño del portafolio de productos
- Reducción de las tasas de defectos
- Mejor Calidad de los productos

Ahora bien, además de estos beneficios fundamentales existen otra clase de beneficios, los cuales pueden clasificarse como Tácticos o Estratégicos [19]. Se puede asumir que lo táctico se refiere a beneficios netamente enfocados a la parte ingenieril del desarrollo o bien a la parte más técnica, y de algún modo son cuestiones que luego se resumen en parte de los beneficios principales.

Beneficios tácticos:

- Reducción en el *Time-to-Market,* típicamente menos demora en el desarrollo y tiempos de entregas.
- Reducción en la cantidad de defectos por cada producto.
- Practicidad a la hora de desarrollar un producto, así como mantenerlo.
- Incremento en el número total de productos que pueden ser entregados en un tiempo dado.

Luego están los beneficios estratégicos, que dan a entender una relación con el negocio o dominio en el que se aplica puntualmente.

Beneficios estratégicos:

- Reducción de tiempos de entrega, lo que impacta en un menor Time-to-Market, así como una reducción en el *Time-to-Revenue* (tiempos de repago de la inversión)
- Mejoras en el valor competitivo del producto final.

- Mayores márgenes de ganancias.
- Mejoras en la calidad del producto, así como una mejor reputación para la empresa.
- Mejor capacidad de adaptarse según crecimiento, rápida y fácil escalabilidad.
- Menos riesgos al entregar el producto al cliente.

3.2.3 Fundamentos de las Familias de Productos

Es importante entender de maneras conceptual algunos de los aspectos más relevantes de las LPS, los cuales sientan las bases para su utilización. Cualquier grupo que quiera utilizar esta técnica a la hora de desarrollar software tendrá que tener en claro estos aspectos más teóricos y conceptuales. Los aspectos conceptuales tienen que ver con los siguientes puntos: la reutilización de software, los activos de software, los componentes de software reutilizables, los dominios y familias y las líneas de productos de software.

A continuación se describen uno por uno estos conceptos.

Reutilización de software

En la Figura 6, tomada y analizada en [5,49] se puede ver gráficamente cómo se ha ido evolucionado en el uso de este concepto. En este sentido siempre se han destacado tres características principales de la reutilización:

1. Estratégica: maneja estratégicamente la variación entre los productos de la línea y elimina la duplicación de esfuerzos de ingeniería.
2. Predictiva: la reutilización de activos se da uno a más productos sobre una línea ya definida y porque se reutilizan arquitecturas de software más que componentes de manera oportunista.
3. Gestionada: la reutilización es simétrica, planificada, institucionalizada y mejorada.

Figura 6 - Evolución de la reutilización de software

Activos de software reutilizable

En pocas palabras, un activo de este tipo es un producto de software que está diseñado para ser utilizado múltiples veces en el desarrollo de diferentes sistemas o aplicaciones.

Algunos ejemplos podrían ser los siguientes:

- Un componente de software
- Una especificación de requisitos
- Un modelo de negocios
- Una especificación de diseño
- Un patrón de diseño
- Una arquitectura de domino
- Un algoritmo
- Un esquema de base de datos
- Una especificación de prueba
- La documentación de un sistema
- Un plan

Componentes de software reutilizable

Una definición posible para este concepto es la siguiente: “Es una pieza de software funcional que es liberada independientemente y que proporciona acceso a sus servicios a través de sus interfaces” [42].

Al hablar de un componente en este caso se hace referencia a un aspecto de bajo nivel donde ya estamos más cerca de la pieza entregable en sí misma. Dicha pieza se puede instanciar de diferentes manera independientemente una de otra, y además prestan servicios utilizando sus interfaces.

Para su correcto manejo estos componentes deben ser poseer las siguientes características: Identificables, autocontenidos, rastreable a través de su ciclo de desarrollo, reemplazable por otro componente, accesible solamente a través de su interfaz, poseer inmutabilidad de sus servicios, proveer documentación de sus servicios y contar con mantenimiento sistemático.

En base a [5] sus características se pueden clasificar de la siguiente manera:

- Según su Modalidad:
  - Caja negra
  - Caja Blanca

- Según su Granularidad
  - Componentes de uso específico
  - Componentes de negocio
  - Frameworks
  - Componentes de aplicación

- Según su fabricante
  - Hechos en “casa”
  - Comerciales Off-the-Shelf

- Según la tecnología usada
  - Imperativos
  - OO
  - Distribuidos

A continuación se presenta el concepto de Variabilidad, el cual es clave dentro del dominio de Familias de Productos.

3.3 Variabilidad

En la última década ha surgido una creciente necesidad de trabajar con sistemas más adaptables y flexibles según diferentes necesidades, dominios y hasta tipos de usuarios. Es decir, se necesitan cada vez más sistemas altamente configurables. Fue en este contexto cuando los sistemas comenzaron a requerir mayor soporte a la "variabilidad" en sus componentes y funciones. Al mismo tiempo, como ya se ha explicado anteriormente en esta tesis, había surgido la necesidad de que los productos de software se agrupen en familias (Familias de Productos- SPL), para poder reutilizar la mayor parte posible de sus componentes y/o artefactos que lo componen. Así, el concepto de Variabilidad cobra relevancia y nacieron varias técnicas para

administrar y modelar variabilidad y más importante aún para encontrar los puntos de variabilidad vs los aspectos comunes. Es en esta tarea en donde se concentran la mayor cantidad de esfuerzos y estudios teóricos-prácticos.

Es importante entonces para poder completar el camino de aprendizaje sobre familia de productos repasar los aspectos más representativos del diseño de variabilidad, así también como sus técnicas de estudio y la relación intrínseca que existe entre variabilidad y el diseño de familias de producto.

Formalmente, se podría definir variabilidad como “La habilidad de un sistema software o artefacto para ser cambiado, personalizado o configurado para usarse en múltiples contextos" [22]. El concepto de variabilidad, encierra otros conceptos que bien merecen ser analizados [23]:

- La variabilidad sujeto: podría ser una variable del mundo real o una propiedad del objeto/elemento, por ejemplo, el color de algún objeto o la forma de pago de una aplicación bancaria.
- Variabilidad Objeto: es una instancia particular de un objeto. En base al ejemplo anterior, sería el color rojo de un elemento o pago con tarjeta de crédito. Es decir, son instancias de la variabilidad sujeto.
- Punto de variabilidad: es la representación de una variabilidad objeto dentro de los artefactos de dominio, los cuales pueden estar enriquecidos por el contexto.
- Una variante: es una representación de una variabilidad objeto dentro de los artefactos de dominio. Es una identidad diferente en requisitos, arquitectura, etc., que identifica una única opción de un punto de variabilidad.

3.3.1 Variabilidad y Familias de productos

La variabilidad, es de hecho, el elemento más característico a la hora de diseñar o trabajar con PLA, ya que su gestión temprana y su correcta gestión son fundamentales para una definición de la familia de productos satisfactoria.

Cuando se gestiona variabilidad en una línea de producto se requiere distinguir tres tipos principales de variabilidad, tal como se detalla en [41]:

1) Opcional

Aquí la variante de un punto de variabilidad es que puede ser parte o no de la LPS de una aplicación. Se establece una relación entre el punto de variabilidad y la variante con una cardinalidad mínima de 0 y una cardinalidad máxima de 1. Recordando el ejemplo en la Figura 3: las variantes que definen los accesorios del móvil como cámara de fotos, *bluetooth*, etc. Las mimas se definen como opcionales, de forma que el cliente puede elegir ninguna de ellos, una o más de una.

2) Obligatoria

En este caso, la variante de un punto de variabilidad es sí o sí parte de la LPS de una aplicación. La relación que se establece entre el punto de variabilidad y la variante es ahora de una cardinalidad mínima de 1 y una cardinalidad máxima de 1. Un ejemplo podría ser: la comunicación encriptada de un sistema ofrece diferentes longitudes de clave: de 128 bits a 1024 bits. El ingeniero de la LPS podría establecer que la encriptación de 128 bits es la mínima protección que se requiere para cualquier aplicación de acceso remoto. Por eso, 128 bits es una variante obligatoria, mientras que 256 bits, 512 bits, y 1024 bits serían variantes opcionales.

3) Alternativa Múltiple

Ahora un punto de variabilidad puede seleccionar o no más de una variante. La relación entre el punto de variabilidad y la variante con la cardinalidad mínima entre 0..1, y máxima de N. Volviendo al ejemplo de la figura 3, las variantes que forman los accesorios del móvil como cámaras de fotos, bluetooth y GPS se pueden definir como una alternativa múltiple. Por ejemplo, establecer que como mínimo ha de elegirse un accesorio y como máximo 2.

Presentado los conceptos de Familias de Productos y Variabilidad, el siguiente capítulo se concentra en el modelado arquitectónico de familias de Productos.

4. Arquitecturas de Familias de Productos

En el presente capítulo se presentan las diferentes estrategias que rondan alrededor del modelado arquitectónico de las Arquitecturas de Familias de Productos. En particular se recorren los diferentes modelos más utilizados para modelar variabilidad, el aspecto más importante en el tema.

Ya se ha mencionado anteriormente el rol relevante que juega en el diseño de PLS el concepto de variabilidad. Dicho concepto se transforma algo clave cuando profundizamos en ese diseño, y así ya comenzar a hablar de Arquitectura de Familia de Productos. En este punto debemos materializar modelos que puedan ser interpretados para luego llevarlos a la práctica e implementar el desarrollo de familias de productos.

A continuación se describen varias técnicas para el modelado arquitectónico de familias de productos y variabilidad. La Sección 4.1 se enfoca en el modelado a través de **Features** [28]. La Sección 4.2 analiza un modelado arquitectónico tradicional basado en componentes y conectores. La Sección 4.3 introduce la noción de *Plastic Partial Components* [6], enfocado en modelar variabilidad interna de los componentes. Finalmente, la Sección 4.4 expone la utilización de la Programación Orientada a Aspectos [31] para el modelado arquitectónico de familias de productos.

4.1 Modelado de características o Features

Una de las formas más usadas para expresar las distintas funcionalidades de un software es la de "*características"* o, de ahora en mas, Features (de su traducción del Inglés). Un feature es entonces, una propiedad que representa una parte común o variable de un producto. Más formalmente:

- *"Un feature es una unidad lógica de comportamiento que es especificada por un conjunto de requisitos funcionales o de calidad" [25].*

Como se detalla en [41], a inicios de la década del ´90, se propone uno de los primeros modelos de Features, el cual, utilizando una estructura de jerarquía de árbol, logra describir toda la variabilidad posible en una línea de producto. La idea es seleccionar una configuración de este modelo y así obtener la definición de un producto específico de la línea de producto.

Luego surge, un modelado denominado FODA (Featured Oriented Domain Analysis) [28], el cual surgió desde el SEI[3]. Como es de esperar, este modelo también persigue distinguir las características comunes y variables de los sistemas de software en un dominio en particular. Aquí se consideran una característica como un *aspecto,* cualidad o característica visible por el usuario, destacada o distintiva de un sistema o sistemas de software.

En pocas palabras en FODA se resalta tres tipos diferentes de Features:

- Obligatorias
- Opcionales
- Grupos de Características XOR

Además, este modelo propone descripciones explícitas para las restricciones *requires y mutex*, las cuales rigen el comportamiento de la interacción entre los distintos features.

Luego, se realizó una extensión del método, y surgió FORM (Feature Oriented Reuse Method). Lo que se hizo aquí fue agregar los grupos de características OR. A partir de esta extensión comenzaron a sumar o integrar componentes, dándole mas poder expresivo a los modelos. Podríamos resaltar el modelo de Czarnecki [27], un meta modelo en el que aparecen cardinalidades a los features y grupos de features.

Básicamente este modelo se trata de un árbol jerárquico, cuyo nodo (raíz del árbol) es un feature. De este nodo raíz cuelgan otros features que también pueden ser simples o grupos de features. Más adelante se presentará un ejemplo de un dominio real para ilustrar estos conceptos. Pero ahora vale la pena entender mejor qué tipo de features están en juego, sumando algo más de contenido a lo que ya ha sido detallado en el capítulo 3:

- Features obligatorios: Estarán seleccionadas si y solo si su padre está seleccionado. Se representan con un círculo lleno en el arco de la característica.
- Características opcionales: Pueden ser seleccionadas solo si su padre está seleccionado. Se representan con un círculo vacío en el arco de la característica.
- Características alternativas XOR: Son un conjunto de características de las cuales una de ellas será seleccionada si su padre es seleccionado. Se representan con un conjunto de arcos agrupados con cardinalidad 1..1.
- Grupos OR de características: Son un conjunto de características de las cuales un grupo de ellas serán seleccionadas si su padre es seleccionado. Se

[3] http://www.sei.cmu.edu/

representan con un conjunto de arcos agrupados con cardinalidad variable (desde 0..1 hasta m..n en general).

- Restricciones mutex y requires: La relación requires significa que al seleccionar una característica la característica unida por la relación es seleccionada también. La relación mutex nos indica que cuando seleccionamos una característica la característica unida por la relación debe ser excluida. Las restricciones serán analizadas más adelante en este capítulo.

La figura 8 [41] muestra la representación gráfica de los elementos recién descritos.

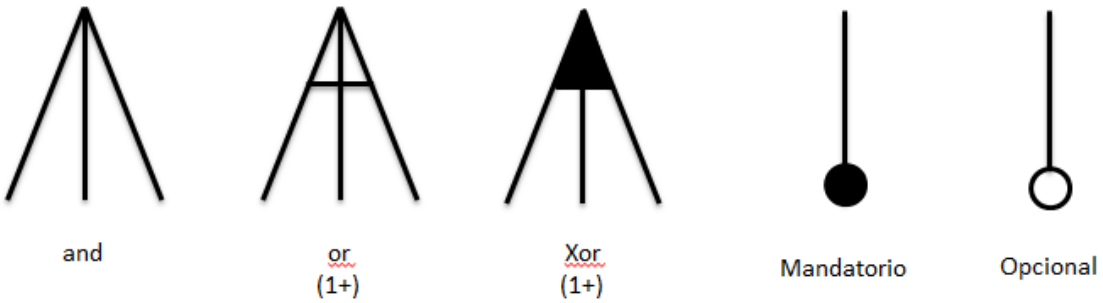

Figura 8- Representación gráfica de los Features y grupos de Features.

Sabiendo interpretar gráficamente los diferentes features es posible observar en la figura 9 un ejemplo de representación de un modelo de características para un sistema de pagos [41].

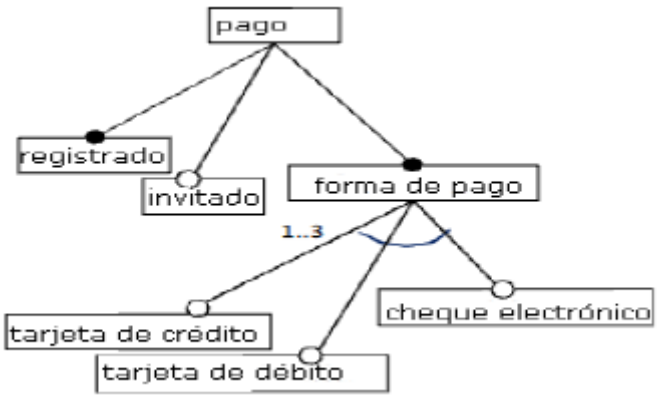

Figura 9-Ejemplo de representación gráfica de un modelo de Features

Viendo el modelo de la figura 9 se puede observar que, en este caso, el sistema de pagos va a contar con varias funcionalidades. Como mínimo, tendrá un usuario registrado, pero también podría tener un usuario invitado. Aquí se ve puede notar como un feature puede tener "variabilidad" y luego al integrarse con otros objetos del sistema tendrán diferentes usos o características. También, en lo que respecta a la forma de pago, se podrán implementar al menos 3 opciones: Tarjeta de crédito, tarjeta de débito y cheque electrónico. Como en el caso anterior, este feature tendrá fácil adaptabilidad a múltiples tipos de pagos.

Luego, queda claro que este manejo se puede extrapolar a sistemas más complejos, en donde la implementación de una u otra modalidad puede insumir recursos y tiempos. Tener esta flexibilidad de implementación otorga una ventaja para disminuir la inversión tanto en recursos como en tiempo.

Retomando el tema de composición de diferentes tipos de Features es necesario detallar cómo se especifica la interacción entre los distintos features. La misma se lleva a cabo a través de reglas de composición, las cuales se detallan a continuación.

Reglas de Composición o Restricciones

Los features están relacionados unos con otros principalmente mediante el uso de reglas de composición, las cuales son un tipo de *restricción* sobre el uso del feature.

Típicamente las reglas de composición tienen dos tipos o formas:

1. Aquellas que requiere la existencia de otro feature (porque son independientes). Se conocen como restricciones de tipo *requires*.
2. Aquellas que son mutuamente exclusivas con otras (no pueden coexistir). Se conocen como restricciones de tipo *mutex*.

Simbólicamente se representan como:

<feature1> ('requires' I 'mutex-with') <feature2>

A continuación se muestra un ejemplo para comprender mejor el enfoque, basado en el ejemplo expuesto en [28]. El ejemplo permite mostrar cuáles son las reglas de composición que se utilizan en los métodos de modelado de features antes descriptos. Supongamos el caso que se quiere modelar una familia de productos enfocada en un administrador gráfico de ventanas. En particular, para el caso de un Administrador de Ventanas se puede introducir la siguiente regla:

*<moveIcon **requires** hasIcons>*

Esto implica que los productos de la familia implementando la característica o feature *moveIcon* requiere la presencia de otra funcionalidad: *hasIcons*. Si bien esto parece obvio (para mover los íconos de una ventana necesito una ventana con íconos), otro tipo de restricciones entre features de la familia pueden no ser tan fáciles de especificar y estas interdependencias podrían perderse dentro de la complejidad general. Continuando el mismo ejemplo, se presentan a continuación otros tipos de reglas, tal como puede observarse en la tabla 2.

Regla	Interpretación
opaqueFeedback **mutex-with** moveErasure	Si la imagen de la ventana completa se mueve, entonces no queda nada en la posición anterior para borrar.
zapEffect **requires** ghostFeedback	Si la imagen de la ventana completa se mueve, no hay ventana para dibujar las líneas.
zapEffect **requires** eraseAfter	Si la imagen vieja fue borrada antes de la operación *move*, podría no haber entonces donde dibujar las líneas.
ghostFeedback **requires** moveErasure	Si *ghostFeedback* es usado, entonces la imagen de la ventana anterior debe ser eliminada en algún punto, antes o después. Así 1 de las 2 alternativas de *moveErasure* debe ser seleccionada.
expcseAfrerMove **requires** overlappedLayout	Una operación *expose* solo puede ser hecha en un sistema *overlapped*.

Tabla 2: Ejemplo de restricciones entre features

Este tipo de reglas surgen en realidad más que nada, de la experiencia y conocimiento del dominio.

4.2 Arquitectura con Componentes y Conectores

Ahora que ya ha sido presentada la noción de feature y qué rol juega en una arquitectura de familia de productos se puede profundizar en el diseño arquitectónico de una familia de producto. Una de las opciones es el modelado a través de componentes y conectores, tal como se los describe en [41]. Básicamente este diseño no es ni más ni menos la clave para la reutilización sistemática, ya que brinda las herramientas para describir la estructura de toda la familia de productos, mostrando sus componentes y las relaciones entre los mismos a través de las interfaces. Justamente, una *interfaz* define una relación contractual entre un componente que requiere la realización de una funcionalidad y otro que la provee. La especificación de la interfaz es independiente del componente que la implementa [41].

Como en el caso de los features, modelar variabilidad a nivel de una arquitectura implica diferenciar componentes comunes y componentes opcionales. Por un lado, están los componentes comunes de la arquitectura, los cuales estarán presentes en cada pieza de software que pertenezca a la misma familia. Por otro lado, los aspectos variables serán capturados por los componentes de software que varían entre miembros de una misma familia. En [41] ilustran esto con el siguiente ejemplo. Dentro de una familia de productos existen algunos que requieren interactuar con el usuario para obtener sus comandos, mientras que otros sólo utilizan comandos internos sin necesidad de interactuar con el usuario. Toda la parte de procesamiento de comandos estará a cargo de componentes comunes, ya que estará presente en todos los productos. La interfaz con el usuario por otro lado será un componente variable, que estará presente sólo en algunos productos. Este tipo de manejo de variabilidad llevado a cabo a través de agregar/remover/modificar elementos arquitectónicos (componentes, conectores, etc.) se conoce como variabilidad externa. Pero a veces este tipo de especificación (variabilidad externa) no es suficiente, ya que puede ocurrir que el comportamiento de un mismo componente posea variabilidad de un productor a otro [6,29].

Debido a esto, nace lo que se conoce como *Variabilidad Interna,* en donde como su definición indica, es necesario especificar variabilidad dentro de cada componente. En el ejemplo de los teléfonos móviles (figura 3), puede observarse un claramente variabilidad interna. Cada teléfono tiene una interfaz gráfica con varias características comunes, siendo estos teléfonos de la misma familia (SPL). Sin embargo, el tipo de interfaz dependerá de la resolución de la pantalla de cada producto (teléfono) que se incorpore. Puede observarse entonces que la interfaz es un componente con variabilidad interna.

4.3 Plastic Partial Components

Se presenta a continuación una alternativa para modelar y especificar la variabilidad interna de componentes arquitectónicos. Esta alternativa fue presentada en el trabajo [6]. La solución está basada en un tipo especial de componentes denominados PPC (Plastic Partial Components).

En esta sección se describe la idea general de la propuesta, se presenta luego un meta modelo para profundizar conceptos y la misma concluye mostrando un ejemplo.

4.3.1 Plastic Partial Components: Descripción e idea general

Básicamente, un PPC es un componente donde una parte de su comportamiento es común a toda la familia de productos, mientras que otra cambia o varía de un producto a otro. Luego, son componentes que quedan parcialmente especificados al modelar la arquitectura de toda la familia de productos, ya que sólo se modela en esta instancia el comportamiento común. Esta es la razón por la cual se denominan *"Partial"* (parcial), ya que son componentes que están parcialmente descritos en el núcleo de la arquitectura, pero listos para ser extendidos en la arquitectura del producto. Y por otro lado la parte de *Plastic (Plástico),* viene dado por la flexibilidad de adaptación de su comportamiento a cada producto de una SPL.

Los PPC resuelven una parte del problema, pero introducen una nueva instancia de definición, ya que el componente está "parcialmente" definido. Luego entonces, se debe definir el comportamiento de un determinado producto dado. Para tal fin es necesario proveer mecanismos para instanciar esa variabilidad que había quedado parcialmente especificada con el comportamiento propio de cada producto. Dichos mecanismos están basados en Principios de Composición de Software Invasivo (Invasive Software Composition Principles) [30] y la Programación Orientada a Aspectos [31].

Para conseguir esta composición se inicia, por un lado, definiendo fragmentos de código que se corresponden con la funcionalidad específica de cada producto. Cada uno de estos fragmentos se denomina *"variants*". Luego, para cada producto, se debe establecer dónde y cuándo se extenderá el comportamiento de un PPC con un determinado variant, siguiendo una metodología similar a la Programación Orientada a Aspectos (POA) [31]. Este proceso se denomina *"weaving"*, siguiendo la terminología de la POA [31]. Un weaving requiere definir dos datos fundamentales, el *dónde* y el *cuándo*. Para la parte que especifica el **dónde,** que se conoce como *"pointcut"*, se establece dónde se inserta el código que corresponde a un variant. Por ejemplo, un pointcut posible podría ser la llamada a un servicio que provee el PPC. La parte del **cuándo** consiste de tres opciones: *antes, después o durante*. Continuando el ejemplo, el código de un variant se podría agregar a un PPC antes, después o durante la llamada a un servicio del PPC.

Finalmente, los lugares donde un PPC puede ser extendido se denominan "variability points" o puntos de variabilidad.

Respecto a los fragmentos de código "variants" los autores proponen una clasificación, según se trate de una funcionalidad que entrecruza toda la arquitectura o no. Luego, se establece la siguiente jerarquía:

- *Features Transversales (crosscutting-features)*
  - *Una característica común de la arquitectura de software en cuestión, el cual se encapsula en una entidad llamada "aspecto o aspect"*
- *Features No-Transversales (non-crosscutting-features)*
  - Es una funcionalidad específica del componente, en este caso se encapsula en una entidad separada llamada *"feature"*

Dado este contexto, en resumen, un PPC se define especificando los siguientes puntos:

1. Su variabilidad interna. Es decir, sus puntos de variabilidad.
2. Los *aspectos o features* que son necesarios para definir el componente de una familia de producto dada.
3. Las conexiones entre los puntos de variabilidad y los aspectos y/o features.

Esta metodología hace posible una gran flexibilidad para componer componentes (aspectos, features y componentes) de un software, del mismo modo a como se arma un rompecabezas.

A continuación se ilustra gráficamente el concepto de PPC descripto en [6] y sus componentes. A los PPC se los distingue de los componentes tradicionales porque tienen dos triángulos blancos para mostrar que están parcialmente especificados. Los puntos de variabilidad se representan con triángulos con la punta rellena, donde se distinguen funcionalidades transversales (AVP) y no-transversales (FVP). Para tener mayor granularidad en la representación gráfica, cada "variant" transversal se representa con un rectángulo horizontal y cada "variant" no transversal con un rectángulo vertical.

La figura 10 muestra un PPC que contiene 2 puntos de variabilidad (uno transversal y otro no transversal) con sus respectivos variants.

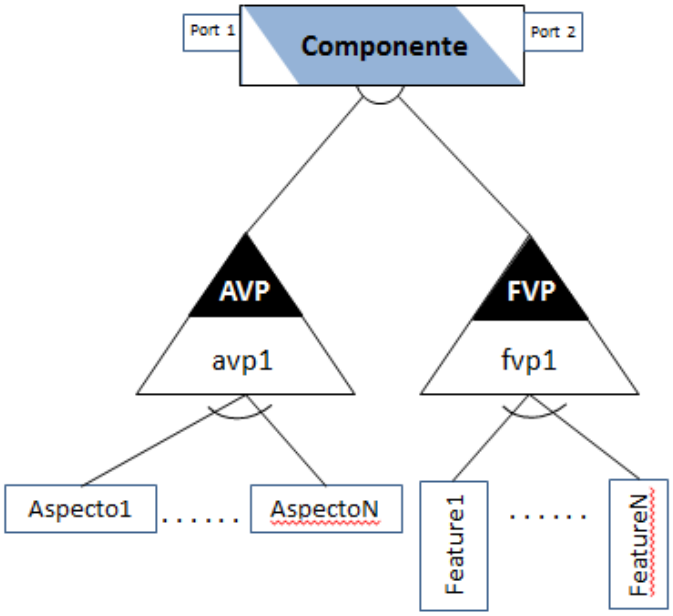

Figura 10 - *Modelo Grafico para Variaciones*

4.3.2 PPC: Metamodelo

Para comprender mejor el concepto de PPC [6] se presenta a continuación el meta modelo de todo el enfoque. El mismo está reflejado en la figura 11.

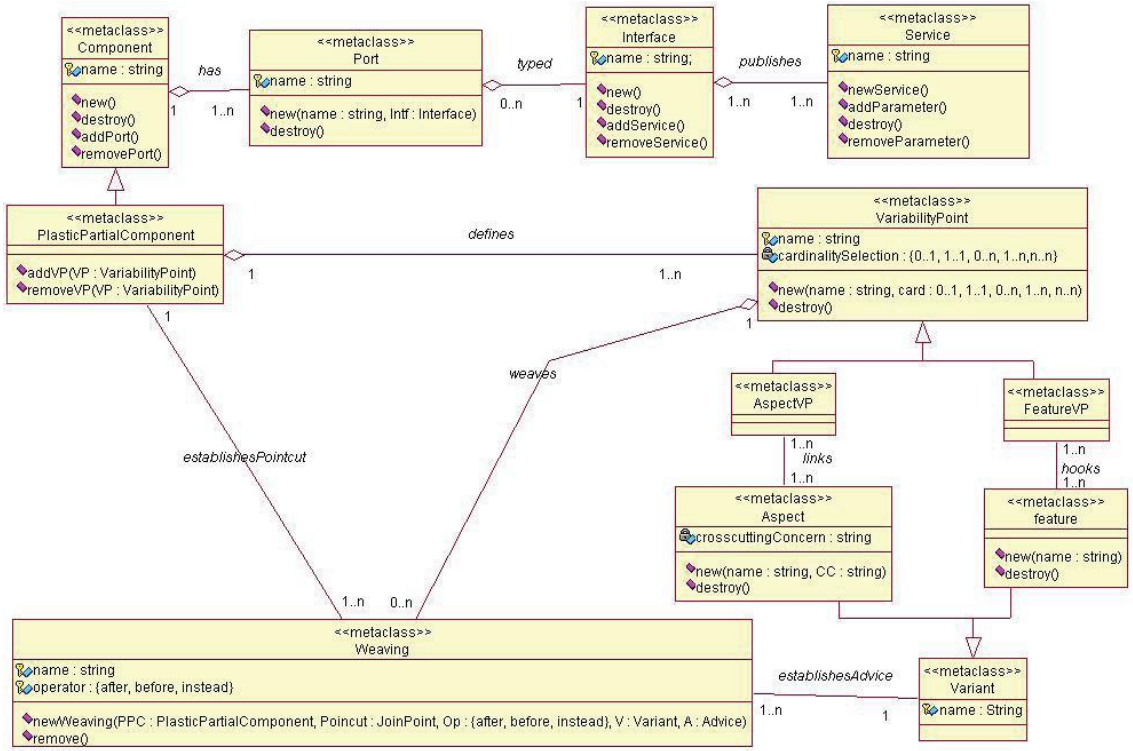

Figura 11 - Meta Modelado de PCC

Lo más importante que se puede deducir del meta modelo es que los PPC son una clase especial de componente, y que la clase *Variability Point* es la clase encargada de modelar la variabilidad a través de 3 conceptos: tipo de variación, tipo de variabilidad (transversal o no-transversal), y la manera de conectar "variants" con PPC.

Respecto al tipo de variación, se puede elegir entre las siguientes opciones:

- 0..1: Opcional y única, cuando un producto es aplicado a la familia/línea de producto, este es opcional para seleccionar una única variación del punto de variación.
- 1..1: obligatorio y único: aquí será obligatorio que una vez elegido el producto se elija una única variación.
- 0..n: opcional y múltiple: ahora será opcional seleccionar una variación de las múltiples opciones de los puntos de variabilidad.
- 1..n: Obligatorio y múltiple: como ya suponemos, en este caso, es mandatorio elegir una variación de las múltiples opciones que habría.
- n..n: por último, ahora se podrán seleccionar múltiples opciones de variación de las a su vez, múltiples opciones disponibles en los puntos de variabilidad.

4.3.3 PPC: Ejemplo de aplicación Sistema Cajero

El siguiente ejemplo está basado en el caso de estudio presentado en [10]. Típicamente los sistemas bancarios consisten en un grupo base de características que brindan funcionalidad a los cajeros automáticos. De todas las funcionalidades que un cajero automático ofrece el ejemplo se enfoca únicamente en el mantenimiento de los estados/balance de cuentas. Esta funcionalidad debe cumplir con el requerimiento no funcional de *disponibilidad.* Algunos productos de esta familia (SPL) requieren disponibilidad restricta 7 x 24, mientras que otros podrían permitir una disponibilidad "más relajada" que se denominará disponibilidad no-estricta.

Un análisis inicial posible de requerimientos es ilustrado en la figura 12, donde se incluyen 3 features: *Cajero*, *Balance* y *Disponibilidad*. Todos ellos son obligatorios, ya que están representados por círculos llenos. El feature *Disponibilidad* tiene un feature del grupo tipo XOR en el cual solo uno de los features agrupados puede ser seleccionado para derivar un producto específico: disponibilidad *estricta y no-estricta.*

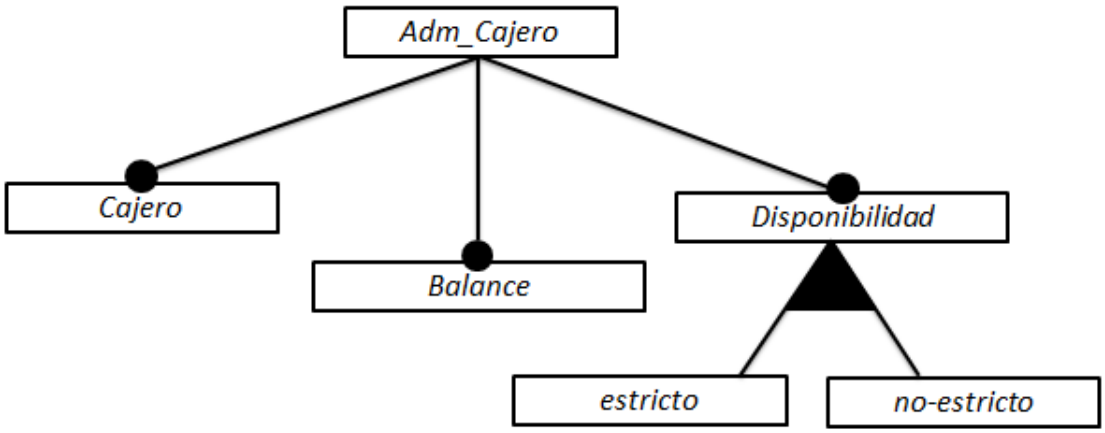

Figura 12 – Modelo de Features

Para este sistema, el modelado arquitectónico incluye un PPC de Saldo ya que una parte de su comportamiento varía de un producto a otro, mientras que el resto es común a toda la familia de productos. Para este PPC, se toman las siguientes decisiones: como el feature Disponibilidad se corresponde con un requerimiento no funcional que entrecruza toda la arquitectura, se modela a través de un tipo de variabilidad aspecto o transversal. Dentro del PPC, se definen entonces dos puntos de variabilidad para las funcionalidades de Actualización y Movimientos.

Algunos productos implementarán una disponibilidad estricta y otros la no estricta. Se definen dos *variants* para la estricta: Sincronización y Balanceo de carga, y dos para la no-estricta: Routing y Monitoreo de Datos. Estos *variants* se corresponden con conocidas tácticas para lograr disponibilidad [17].

Luego, definiendo el weaving apropiado se puede especificar cada producto final como sea necesario, dando así el comportamiento completo para el PPC saldo. Por ejemplo, un producto con disponibilidad estricta utilizando Sincronización y Balance de carga, o bien otro con disponibilidad no-estricta a través de código implementado tácticas de Routing y Monitoreo de Datos.

En la siguiente figura que modela el PPC de Saldo se observa cómo los dos puntos de variabilidad Actualizar y Movimientos podrán ser implementados según tipo de disponibilidad, con uno u otro aspecto.

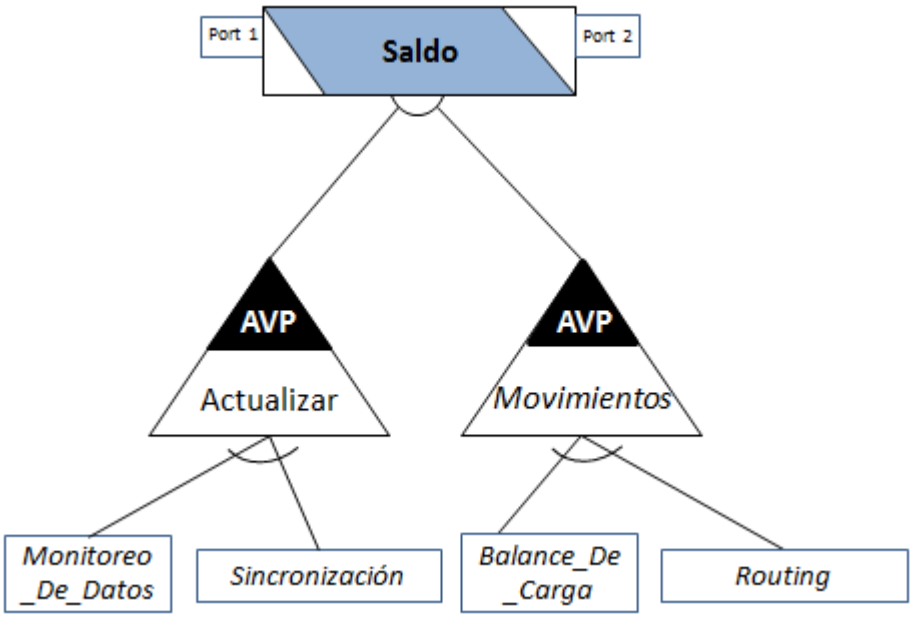

Figura 13: Modelado del PPC

.

4.4 Paradigma de Aspectos aplicado al modelado de SPL

En los últimos años la orientación a aspectos ha surgido como un enfoque interesante para manejar la complejidad de la descripción de artefactos de software. El enfoque orientado a aspectos tiene como núcleo la modularización de conceptos entrecruzados [31], es decir, conceptos que utilizando los mecanismos de estructuración o modularización tradicionales quedarían desparramados por todo el sistema. Tal objetivo es un principio básico de la Ingeniería de Software. Cada aspecto representa y abstrae a cada uno de estos conceptos, y su comportamiento es agregado a un sistema original, denominado sistema base. Los aspectos tienen una estructura de dos partes: los *pointcuts*, los cuales seleccionan donde se introducirá el comportamiento de los aspectos, y los *advices*, los cuales especifican en sí el comportamiento que agregará cada aspecto.

Tradicionalmente, los aspectos pueden agregarse “antes de”, “luego de”, o “en vez de” un instante dado del sistema base. Luego, existe un mecanismo de composición denominado weaving, término al que ya nos hemos referido anteriormente, y aquí lo explicamos, que une el comportamiento de los aspectos a un sistema base para formar el sistema final.

Existen varias alternativas para aplicar la filosofía de aspectos al modelado arquitectónico de familias de productos, tal como puede observarse en [32-35]. Básicamente, la idea es que el comportamiento común equivaldría al sistema base, y cada variabilidad presente en cada producto sea representado por un aspecto.

Presentadas las principales alternativas del modelado arquitectónico de Familia de Productos el capítulo siguiente enumera las principales debilidades y limitaciones identificadas por la comunidad.

5. Modelado Arquitectónico de SPL: Algunos problemas y limitaciones

Si bien existen varias propuestas para el modelado arquitectónico de SPL como se ha expuesto en el capítulo anterior, la comunidad ha detectado algunos problemas y limitaciones. En este capítulo se describen tales problemas para concluir enumerando algunas características deseables que debería poseer un lenguaje de especificación arquitectónica de SPL para atacarlos.

5.1 Variabilidad interna

Como se mencionó anteriormente, la mayoría de las aproximaciones modela los cambios arquitectónicos únicamente a nivel de componentes (removiéndolos o agregándolos, cambiando puertos, conectores, etc.) dejando de lado la posibilidad de especificar que el comportamiento interno de un componente puede variar de un producto a otro. Luego, estas aproximaciones dejan de lado la variabilidad interna de los componentes.

Con respecto a la utilización de features [28] se han encontrado las siguientes limitaciones. Incorporar variaciones como features dentro de la línea de productos activos es una opción posible. Sin embargo no es tarea fácil, dado que una variación puede afectar muchas otras partes de la familia/línea de productos, como también se explica en [45]. Este problema proviene en parte del hecho de que un feature unitario no siempre se corresponde con los componentes implementados en el código. Implementar un feature particular, el cual puede estar disperso en múltiples componentes, requerirá intervenir varias veces en esos sectores de software. Este problemas transversal, hace esta tarea muy difícil para reusar, adaptar o configurar activos de una línea de productos en relación con los features.

Con respecto a las técnicas orientadas a aspectos, también surgen problemas cuando dos aspectos intervienen en el mismo punto, como se indica en [46]. En estos casos las variaciones que introduce un aspecto puede introducir cambios y afectar el comportamiento de otros aspectos. Además, un mapeo uno a uno entre features y aspectos no es una solución factible [45].

5.2 Mantener el nivel de abstracción

Soluciones como el PPC pasan dramáticamente del razonamiento a nivel diseño directo a código. Este esquema puede funcionar si los requerimientos están claros, y la funcionalidad del sistema está claramente establecida también. Sin embargo, en etapas tempranas del desarrollo se necesita contar con métodos que permitan explorar el comportamiento del sistema sin atarse a decisiones prematuras, más cercanas a la implementación, tal como se expresa en [47] y [48]. Es decir, poder analizar distintas posibilidades manteniendo siempre un nivel de abstracción alto, y evitar que aparezcan artefactos más cercanos a la implementación, como líneas de código. Así se permite razonar y analizar con la suficiente abstracción, retrasando lo más posible decisiones de implementación que limiten las posibilidades del sistema de manera prematura.

5.3 Notaciones completas, claras y homogéneas

En un análisis comparativo presentado en [36], se estudian varias herramientas y estrategias para modelar familias de productos, establece que varios problemas en el modelado arquitectónico de familia de productos, nacen desde la notación utilizada para el modelado. En particular, identifican estos problemas con las notaciones exploradas:

- Falta de homogeneidad: Esto se ve claramente al modelar las restricciones de funcionalidad que existen entre varios productos.

Dentro de una familia de productos existen features que pueden tener conflictos entre sí, como ser dos features que no puedan estar presentes simultáneamente, o un feature que requiere la presencia de otro. FODA, según se detalla en [28], incorpora estas restricciones a través de reglas. Estas reglas son expresadas en un formato diferente al de los features, siguiendo un patrón en lenguaje natural para denotar las restricciones. En este sentido el trabajo en [36] señala que las alternativas con distinto tipo de notación pueden llevar a confundir al usuario, perdiendo así usabilidad y entendimiento.

- Distintos niveles de abstracción: Relacionado con el punto anterior, y también con la idea de mantener el nivel de abstracción, el estudio afirma que existen notaciones especialmente dirigidas a stakeholders, para expertos en tecnología, para programadores, para arquitectos, etc. Pero no una alternativa

que contenga una visión integradora. De hecho, algunas tienen como “salida” código o expresiones cercanas a un lenguaje de programación, otros documentos en lenguaje natural, gráficos, etc.

5.4 Conclusiones

Dados los problemas y limitaciones mencionados anteriormente, a continuación se enumeran características deseables que debería poseer un lenguaje de modelado arquitectónico de familia de productos. Las mismas son:

- Cuente con la posibilidad de manejar variabilidad interna.
- Sea posible explorar y analizar el comportamiento, manteniendo el nivel arquitectónico.
- Una notación clara y compacta que permita modelar todos los elementos, tanto comportamiento como restricciones entre features, y que sea flexible para ser entendida por distinto público.

En el siguiente capítulo se describirá una propuesta que apunta a representar un primer paso en conseguir tal lenguaje.

6. Propuesta de modelado para Arquitecturas de Familias de Producto

Tomando como premisas las dificultades enunciadas en el capítulo 5, se propone a continuación un modelo declarativo y flexible que permite manejar y resolver las problemáticas en cuestión. La propuesta está basada en el lenguaje declarativo FVS (Feather Weight Visual Scenarios) [37,43].

En la presente tesis se ha mostrado que el concepto de variabilidad, a la hora de modelar de profundizar en el modelado de familias de productos, es un punto fundamental para lograr que una descripción arquitectónica de una SLP cumpla sus objetivos. Por ejemplo, resulta imprescindible distinguir explícitamente los posibles cambios de comportamiento de un producto a otro, ya que ese es su objetivo esencial. Y uno de los problemas fundamentales en lo que respecta a variabilidad, es el modelado de variabilidad interna.

También se ha mencionado problemas con el nivel de abstracción que manejan los lenguajes o modelos actuales. Aquí el punto es que se realiza un salto directo y muy amplio desde la abstracción del modelado al código, y esto provoca problemas en el entendimiento de la mecánica elegida con interlocutores que no están familiarizados con la implementación y además permite que se pierdan oportunidades de ver y tomar mejores decisiones en etapas tempranas del diseño.

Por último, se comentó la problemática de la notación arquitectónica en cuanto a la variedad de notaciones y especificaciones que pueden confundir y desalentar al usuario en un mundo industrial.

Se requiere entonces un modelado arquitectónico que incluya la posibilidad de modelar la variabilidad interna de los componentes, capaz de mantener el nivel de abstracción necesario para poder explorar el comportamiento arquitectónico en etapas tempranas del desarrollo, y exhibir una notación homogénea en todos sus aspectos. En este trabajo se presenta un paso inicial para cumplir tales objetivos,

presentando al lenguaje FVS (Feather Weight Visual Scenarios) [37,43] como un lenguaje para modelar la arquitectura de una familia de productos.

FVS es un lenguaje visual declarativo para especificar el comportamiento esperado de un sistema en las etapas iniciales del desarrollo de software. Es un lenguaje gráfico y simple basado en escenarios, los cuales representan patrones de eventos exhibiendo gráficamente restricciones sobre trazas.

FVS posee exhibe propiedades que lo hacen atractivo en un dominio arquitectónico. Por un lado, cuenta con una notación sólida y compacta, con una semántica clara, y es lo suficientemente expresivo. Asimismo, su flexibilidad hace posible poder especificar con escenarios tanto propiedades y conflictos entre las características distintivas de cada producto en la familia, variabilidad interna en componentes, como así también señalar claramente aquellas características que son comunes a todos los productos. Por último, FVS es particularmente útil para explorar comportamiento en etapas tempranas, propiedad necesaria en dominios arquitectónicos.

El presente capítulo cuenta con la siguiente estructura. La Sección 6.1 presenta informalmente el lenguaje FVS. La Sección 6.2 analiza FVS como un lenguaje de modelado arquitectónico introduciendo un caso de estudio. Finalmente, la Sección 6.3 presenta las conclusiones del caso de estudio y analiza cómo FVS logra dar un paso para solucionar los problemas detectados en el capítulo anterior.

6.1 FVS: Feather Weight Visual Scenarios

A continuación se presentan, aunque informalmente, las principales características de FVS. Para un panorama más extenso se puede consultar [37,43].

FSV es un lenguaje gráfico basado en escenarios. Estos escenarios son órdenes parciales de eventos denotando esencialmente puntos, los cuales son etiquetados con los posibles eventos que pueden ocurrir en ese punto. A su vez, hay flechas que conectan estos puntos. Las flechas indican precedencia del origen respecto al destino (Figura 14 - a). Además FVS tiene una abreviatura para indicar la próxima ocurrencia de evento luego de otro, o la ocurrencia inmediata anterior de un evento precediendo a otra. Esta abreviatura es una segunda flecha abierta, cerca del punto de destino y si es al revés se encuentra cerca del punto de origen (Figuras 14 - b y c).

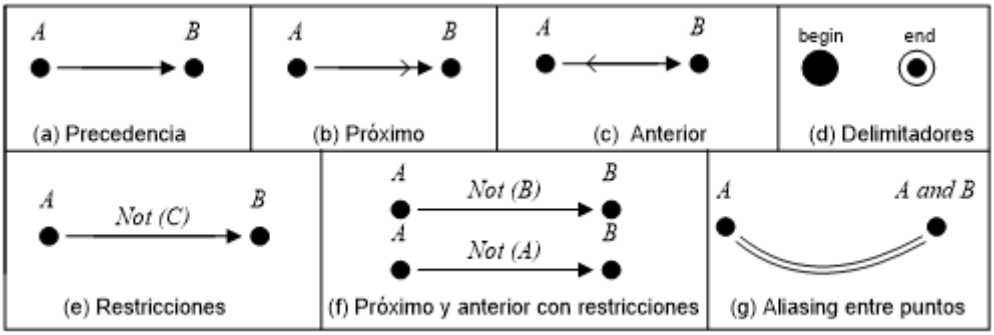

Figura 14- Elementos Básicos de FVS

En la figura 14-d, también se puede ver notación para denotar el inicio y fin de la ejecución. Además las flechas podrán estar etiquetadas para denotar eventos prohibidos entre un evento y otro. La figura 14 – e explica que entre A y B, el evento C no puede ocurrir. Esta notación permite decir cuándo hay restricciones entre eventos. Por último, este lenguaje permite también el concepto de "aliasing" entre puntos, en Figura 14 – g establece que la ocurrencia de A implica también la ocurrencia de B.

Reglas FVS

Una regla se divide en dos partes: un escenario tomando el rol del antecedente, y al menos un escenario tomando el rol del consecuente. Intuitivamente, siempre que en una traza se "encuentre" un antecedente, entonces también se deberá encontrar un consecuente. Una regla puede verse como una implicación, con un escenario antecedente, y uno o más escenarios consecuentes.

El escenario antecedente es una sub-estructura común a todos los consecuentes, lo cual permite establecer relaciones complejas entre los puntos del antecedente y consecuentes. Esto otorga gran flexibilidad a FVS, ya que sus reglas no están limitadas, como la mayoría de las aproximaciones que manejan algún tipo de escenarios de implicación, donde el antecedente funciona únicamente como una estructura que debe preceder a los consecuentes. De esta manera las reglas pueden expresar comportamiento que ocurrió en el pasado, o en el medio de otros eventos. Gráficamente, el antecedente se muestra de color negro, mientras que los consecuentes en color gris. Como las reglas pueden tener más de un consecuente, cada elemento que no pertenece al antecedente se identifica con un número para identificar a qué consecuente pertenece.

Figura 15 - Ejemplo de una regla en FVS

En la Figura 15 se ve un ejemplo. Siempre que un evento *PedidoDeAcceso* es seguido por un evento *PedidoOtorgado* (sin la ocurrencia de un evento *LogOff* entre ellos), una de dos posibles secuencias de eventos debe también observarse en la traza. Las posibles secuencias son:

- La primera (consecuente 1) requiere que luego de que se hace el pedido, se ingrese de manera correcta una contraseña.
- La segunda (consecuente 2) contempla la posibilidad de que la contraseña haya sido ingresada antes de la realización del pedido.

Es decir, o se ingresó la contraseña luego de hacerse el pedido, o el usuario ya la había ingresado al momento de realizar el pedido. Es importante notar el poder expresivo de FVS, donde el antecedente no está forzado a preceder al consecuente, como sí ocurre en otras notaciones donde se describe el comportamiento siguiendo implicaciones o reglas [38-39].

6.2. Modelado arquitectónico en FVS

Habiendo introducido las reglas básicas de este lenguaje, se introduce FVS dentro del universo de dominios arquitectónicos. Para tal fin se presenta un caso de estudio. En el caso de estudio se podrá observar que la capacidad de FVS para denotar comportamiento a nivel de arquitecturas de software para una familia de productos sirve tanto para modelar la interacción entre las distintas características distintivas de cada producto en la familia, la variabilidad interna de componentes así como un producto en particular de la familia. Finalmente, y para facilitar su comparación con el enfoque de PPC, se modela en FVS el ejemplo del cajero automático de la Sección 4.3.3.

6.2.1 El Caso de estudio E-SHOP

El caso de estudio elegido se trata de un sistema de e-commerce (E-SHOP), el cual puede representarse con la notación de *feature*s que ha sido previamente tratada en la presente tesis. La representación puede observarse en la figura 16.

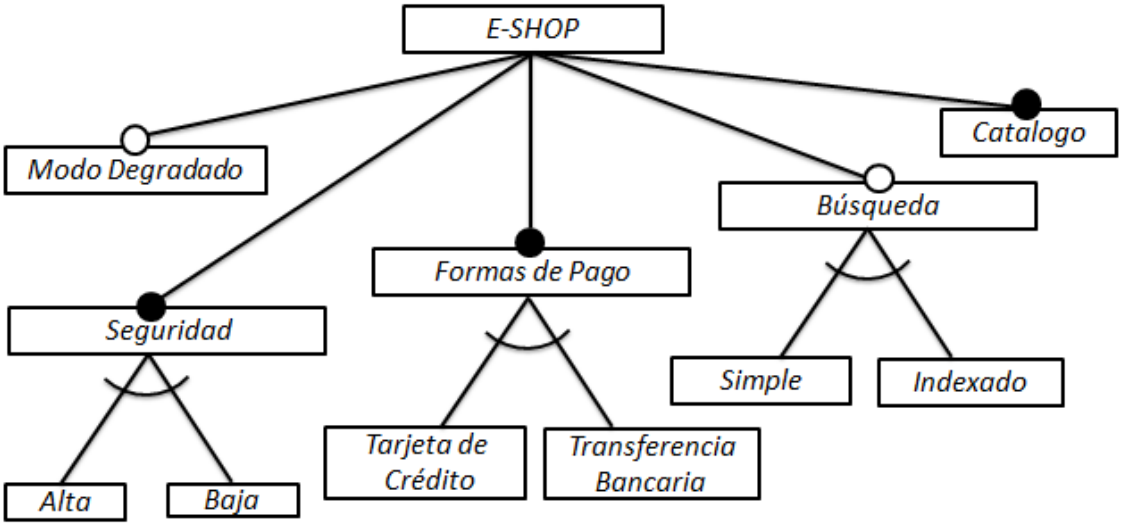

Figura 16- Modelo de características para la familia de productos E-SHOP.

La familia de productos E-SHOP cuenta con las siguientes funcionalidades o características de carácter obligatorio (deben estar presente en todos los productos): formas de pago, seguridad, y catálogo. Dichas funcionalidades obligatorias están denotadas con un círculo relleno negro. Cuenta además con dos funcionalidades optativas, las cuales están representadas gráficamente con un círculo vacío: la *búsqueda de productos*, y la posibilidad de funcionar en un *modo degradado* en situaciones donde la capacidad de los recursos del sistema esté sobrepasada. La Figura 16 muestra además que los productos pueden funcionar con una de dos modalidades de pago (*transferencias bancarias* o a través de *tarjeta de crédito*), que las búsquedas pueden ser *indexadas* o *simples*, y que la seguridad puede ser *alta* o *baja*.

La familia de productos E-SHOP debe cumplir además las siguientes restricciones:

- **R1:** *Los productos con pagos con tarjeta de crédito deben tener seguridad de nivel alto*.
- **R2:** *Las búsquedas indexadas no pueden estar simultáneamente con nivel de seguridad alto debido a limitaciones de performance*.
- **R3:** *Durante el funcionamiento de un producto en modo degradado no pueden ocurrir búsquedas indexadas o de catálogo*.
- **R4:** *Cada transferencia bancaria realizada con éxito debe guardarse en un log por políticas de seguridad.*

A continuación se muestra cómo estos requerimientos que imponen restricciones entre las características que puede exhibir cada producto se modelan en FVS.

Especificación de características y restricciones

Para el modelado de características en FVS se introducen eventos representando cada feature o característica, y las reglas modelan las relaciones e interacciones válidas entre los mismos. Las reglas de la Figura 17 modelan las restricciones R1 y R2.

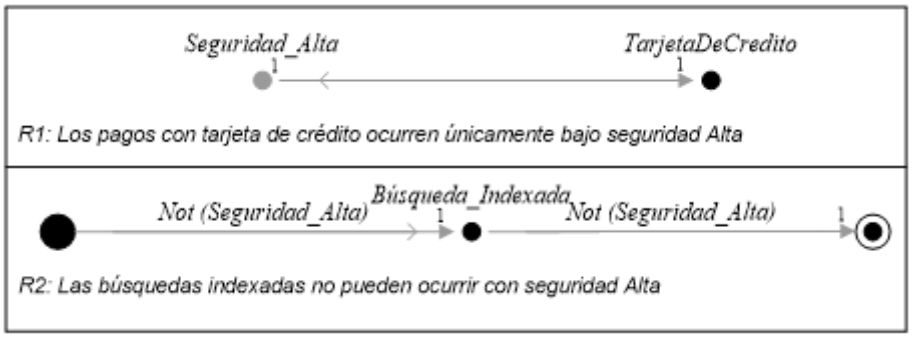

Figura 17 - Reglas FVS para las restricciones R1 y R2.

La regla para la restricción R1 establece que, siempre que esté presente un evento *TarjetaDeCrédito* entonces tiene que ser el caso que haya ocurrido un evento de *Seguridad_Alta* en el pasado. La regla para la restricción R2 exige que sí *Búsqueda_Indexada* está presente entonces no ocurrió nunca previamente *Seguridad_Alta* ni tampoco ocurre en el futuro.

Las reglas de la figura 18 modelan la restricción R3 sobre el funcionamiento del modo degradado. La regla de la parte superior establece que una vez que ocurre el modo degradado, entonces no hay ni búsquedas indexadas ni funcionalidad del catálogo hasta que termina la ejecución (consecuente 1) o bien hasta que el sistema vuelve a funcionar en modo normal (consecuente 2). De manera similar, la regla en la parte inferior exige que siempre que haya ocurrido una búsqueda indexada o de catálogo tiene que ser el caso que el sistema esté desempeñándose en modo normal.

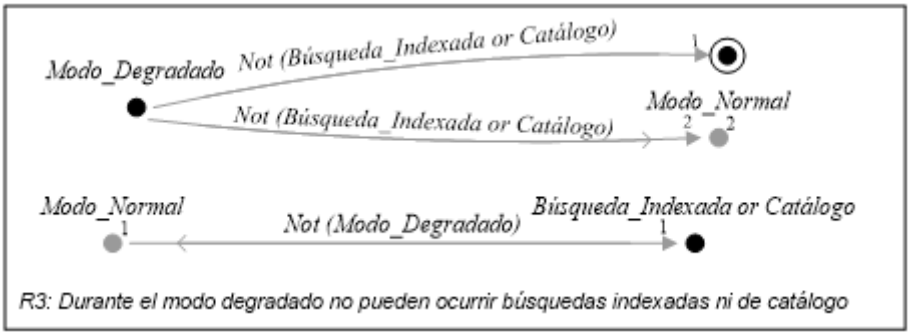

Figura 18 -Reglas FVS para las restricciones R3 sobre el modo degradado.

Finalmente la regla de la Figura 19 modela la restricción *R4*. La misma especifica que siempre que se haya iniciado y terminado con éxito una transferencia, entonces en algún momento entre que se inició y terminó, la información correspondiente fue debidamente guardada en un log siguiendo las políticas de auditabilidad requeridas. Notar que la regla por un lado exige únicamente que las transferencias exitosas sean almacenadas en el log, ya que el antecedente incluye el comienzo y el final de la transferencia. Por otro lado, el almacenamiento en el log puede ocurrir en cualquier momento entre el comienzo y el final, y no se fuerza un instante en particular. Esto permite ilustrar la flexibilidad de la notación al momento de explorar el comportamiento deseado.

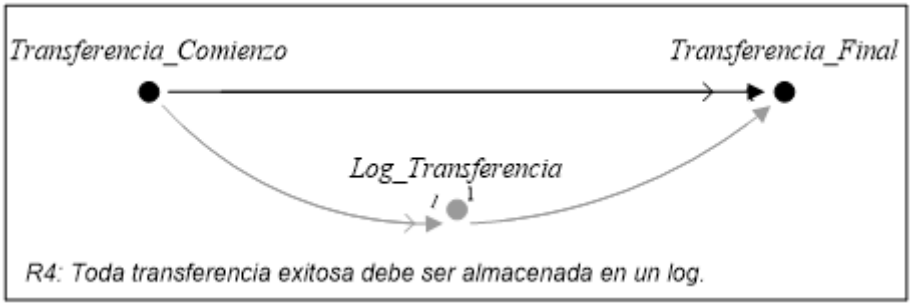

Figura 19- Reglas FVS para la restricción R4.

Variabilidad interna de componentes en FVS

Para analizar la variabilidad interna de un componente, se analiza uno de los posibles componentes del caso de estudio: el componente de *Comunicación*, el cual cumple su función exponiendo una simple interfaz de dos servicios *Enviar* y *Recibir (Ver Figura 20).*

Figura 20 - Esquema componente Comunicación E-SHOP

Como se explicó en el análisis de la Figura 16 algunos productos de la familia implementarán este componente utilizando tácticas de arquitectura [40] que aseguren un alto nivel de seguridad en las comunicaciones, mientras que otros lo implementarán seleccionado la opción de seguridad baja. Esto implica que el componente Comunicación posee variabilidad interna. Es decir, que los servicios que provee pueden ser diferentes en productos específicos de la familia, con niveles altos o bajos de seguridad.

El nivel alto de seguridad puede ser atacado implementado a través de *encriptación* y validación empleando *checksum* cada vez que un mensaje es enviado o recibido. Dichos mecanismos se corresponden con las tácticas de seguridad *Mantener Confidencialidad de los Datos* y *Mantener Integridad de los datos*. De manera similar, el nivel de seguridad bajo puede ser obtenido mediante técnicas de logging al enviar o recibir mensajes, aplicando la táctica de *Auditabilidad*.

Al describir el comportamiento arquitectónico de la familia de productos E-SHOP debe quedar especificado explícitamente que este componente posee variabilidad interna. En algunos productos, antes de enviar un mensaje, los mismos son encriptados y se aplican funciones de checksum. En cambio en otros son simplemente almacenados en un log. La variabilidad interna en FVS puede ser expresada con escenarios con múltiples consecuentes. En esta instancia, los eventos representarán posibles acciones del sistema. Por ejemplo, la Figura 21 muestra las reglas que modelan la variabilidad interna del componente Comunicación en su servicio de Enviar (reglas análogas pueden agregarse para el servicio de Recibir). En este caso, el comportamiento dado en el consecuente 1 especifica cada un mensaje antes de ser enviado es encriptado y se realizan funciones de checksum, mientras que el comportamiento denotado en el consecuente 2 especifica las acciones correspondientes al logging.

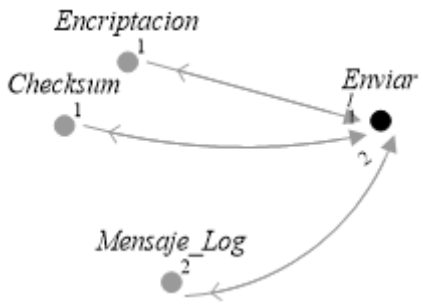

Figura 21 - Regla FVS exhibiendo la variabilidad presente al enviar mensajes

Notar que la regla no exige un orden entre la encriptación y el checksum para la implementación de seguridad *Alta*. Mientras que ocurran ambos, no importa cuál ocurre primero. Nuevamente esta flexibilidad es importante a la hora de explorar el comportamiento, no forzando que un evento tenga que ocurrir antes que otro si no es necesario expresarlo en ese momento. También puede suceder que el arquitecto de software al ver que no hay un orden específico decida cambiar el comportamiento para forzar que la encriptación ocurra primero. Luego, se reescribe la regla de la Figura 21 como lo ilustra la Figura 22. Esto muestra la posibilidad de realizar razonamiento arquitectónico que otorga FVS.

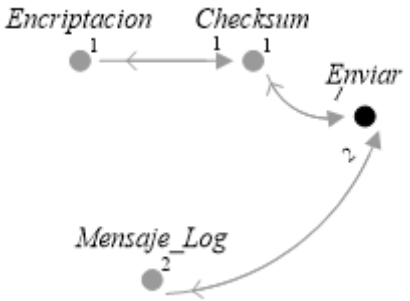

Figura 22 - Nuevo comportamiento para el componente Comunicación.

Especificación de un producto

Uno de los aspectos más importantes en la especificación de un producto es que debe detallar cómo se resolverá la variabilidad interna de cada componente involucrado. En nuestro caso de estudio, el arquitecto de software deberá seleccionar cual de las dos opciones posibles de comportamiento de la Figura 22 estará presente en el producto concreto bajo análisis. En la Figura 23 se muestran las reglas donde se ha seleccionado una seguridad baja (los mensajes son almacenados en un log)

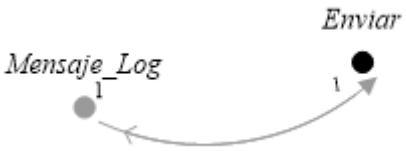

Figura 23 - Un producto implementado seguridad baja.

Es importante notar en este punto un aspecto de interés. Para el caso de estudio del componente *Comunicación* cuando en un determinado producto se elige el nivel de seguridad deseado, la selección debe ser la misma tanto al *enviar* como al *recibir*. Es decir, en todos los puntos donde el componente exhiba variabilidad, la selección debe ser consistente. Esto es fácilmente modelado en FVS agregando una nueva regla. Por ejemplo, continuando con la especificación de un producto con seguridad baja se introduce el comportamiento reflejado en la Figura 24. Si un mensaje fue almacenado en un log, entonces no pudo haber ocurrido antes un evento de encriptación o de checksum, así como tampoco puede ocurrir en el futuro.

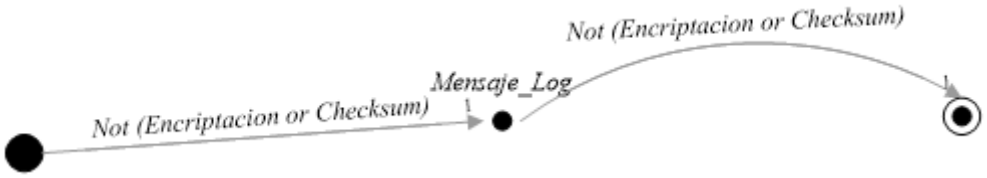

Figura 24 - Nuevo comportamiento para el componente Comunicación.

Revisitando el ejemplo del cajero

Para poder facilitar la comparación de enfoques se presenta a continuación en FVS el modelado del ejemplo del cajero, presentado previamente en la sección 4.4.3. En dicho ejemplo el componente Saldo exhibía variabilidad interna (ver Figura 13). A continuación se modelan en FVS algunas reglas de variabilidad este componente. La regla de la figura 25 establece qué, cuando sea invocado el servicio de Actualizar, o bien se hará un monitoreo de datos, o los datos se sincronizarán.

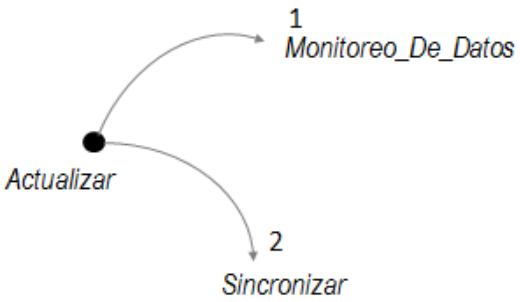

Figura 25 - Regla de Variabilidad Interna para Manejo de Actualización de Saldos

Adicionalmente, también podemos sumar una regla FVS que modela comportamiento, usando el mismo ejemplo del componente Saldo. En este caso vemos que si se ejecuta un Monitoreo de Datos, al cual según lo planteado es implementado para el caso de Disponibilidad *no-estricta,* entonces no podrá existir la implementación de su contraparte Balance de Carga, en el módulo de Movimientos. De esta manera se garantiza que la disponibilidad no-estricta ha sido elegida de manera consistente en el producto.

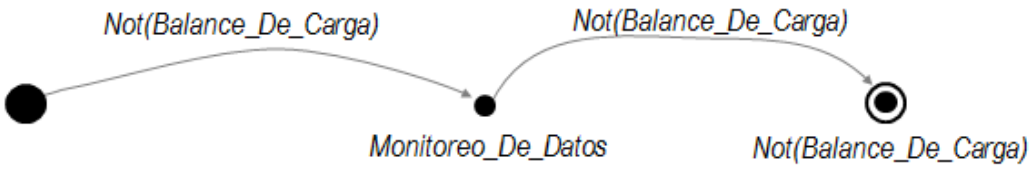

Figura 26 - Modelando comportamiento de parte del componente Saldo.

6.3 Discusión y análisis del caso de estudio

Si bien el caso analizado es simple cuenta con el suficiente comportamiento para realizar conclusiones de su modelado en FVS.

En primer término bajo los escenarios FVS es posible modelar tanto las distintas características dentro de la familia de productos así como las restricciones entre las mismas, exhibiendo homogeneidad en la notación. En las Figura 17, 18 y 19 se muestran reglas para modelar los requerimientos arquitectónicos R1, R2 R3 y R4. Bajo el mismo concepto de reglas es posible especificar todo el comportamiento. En otras notaciones como [26-28] las restricciones y las características se modelan con distintas notaciones. Mientras que las características se modelan con una notación gráfica (ver

Figura 16), las restricciones se especifican través de reglas expresadas en lenguaje natural. En este sentido, utilizar distinto tipo de notaciones es señalado en [36] como uno de los problemas a enfrentar al momento de transferir lenguajes arquitectónicos a la industria del software.

De la misma forma, la variabilidad interna en FVS también es modelada a través de reglas, profundizando el concepto de homogeneidad. En la Figura 24 se introduce la regla donde se selecciona el nivel de seguridad deseado en un determinado producto. Esto le brinda la posibilidad al arquitecto de software de expresar el comportamiento arquitectónico tanto de la familia de productos, de la variabilidad interna de sus componentes y las restricciones entre las distintas características bajo una única notación: reglas FVS.

Otro punto interesante a mencionar es la flexibilidad de la notación para poder razonar y explorar el comportamiento a nivel arquitectónico evitando la introducir artefactos más cercanos al código, lo cual puede llevar a tomar decisiones de carácter implementativo de manera prematura. Por ejemplo, al modelar la variabilidad interna del componente *Comunicación* en la Figura 22, la regla FVS es lo suficientemente flexible para no imponer un orden entre las funcionalidades de encriptar y checksum. En caso de utilizar artefactos más cercanos al código, el arquitecto de software quizás se vea forzado a imponer un orden arbitrario entre estas acciones. Sin embargo, la razón detrás de esta decisión no es una decisión basada en comportamiento arquitectónico, sino que fue tomada por cuestiones implementativas. De esta manera, se pierde el nivel de abstracción necesario para especificar el comportamiento arquitectónico. FVS permite mantener el nivel de abstracción pudiendo expresar con flexibilidad el comportamiento esperado. Más aún, como puede observarse en la Figura 23, es posible establecer un orden dado entre las funcionalidades si así fuera requerido.

Lo mismo puede observarse en la figura 19. La regla FVS permite que el comportamiento de almacenar en un log ocurra entre la ocurrencia de otros dos eventos, y no específicamente en un momento dado. Con notaciones cercanas al código es posible que el arquitecto deba tomar una decisión implementativa y forzar a que el log ocurra en un instante dado, antes o después de la invocación a un método.

La variabilidad interna de un componente es modelada de manera natural y directa en FVS a través de reglas. La posibilidad de modelar distinto comportamiento esperado para un consecuente se traduce directamente en las distintas variantes de un producto. Luego, al momento de especificar un producto dado, se elige únicamente

el comportamiento esperado, y se introduce la regla correspondiente. Un hecho importante para destacar es que también es posible especificar que una misma decisión arquitectónica sea consistente en todos los lugares donde un componente exhibe variabilidad. En el caso de estudio presentado se especifica que una misma decisión arquitectónica sea aplicada en los dos servicios que expone el componente *Comunicación*. La regla especificada en la regla 24 expresa justamente este requerimiento. Nuevamente, con notaciones donde se pierde el nivel de abstracción y se introducen artefactos cercanos al código es mucho más difícil expresar tal requerimiento.

Dado este panorama se puede observar que FVS exhibe buenas propiedades para ser utilizado al momento de explorar el comportamiento arquitectónico en etapas tempranas a través de una notación simple, homogénea, donde es posible mantener un alto nivel de abstracción, el cual necesario para razonar en un mundo arquitectónico. En una etapa posterior, con el comportamiento arquitectónico ya definido o al menos especificado con mayor madurez, FVS se podría combinar con otras técnicas e introducir cuestiones cercanas al código.

7. Trabajo futuro

La presente tesis deja planteado diversos tópicos que podrían continuar la presente investigación.

En primer término, es necesario continuar la validación de FVS en casos de estudio de relevancia industrial. La exposición a casos de estudio con mayor complejidad es uno de los desafíos a cubrir en el futuro.

Otro aspecto por cubrir es la integración de FVS con otras herramientas, de manera de ampliar su potencial. Dado que FVS está enfocado en el comportamiento inicial, es posible combinarlo con otras notaciones que incorporen código como una etapa posterior. Una posibilidad en este sentido sería ver la posibilidad de combinar reglas FVS con componentes *PPC* [6]. Adicionalmente, dado que las reglas FVS, pueden también traducirse en autómatas [37], y sería interesante utilizarlos en otras herramientas de razonamiento arquitectónico. Estos dos objetivos apuntan a consolidar un framework declarativo y completo para expresar y validar el comportamiento arquitectónico de una familia de productos.

Respecto a la utilización de herramientas, mejorar soporte de software actual de FVS es también un punto a mejorar en el corto plazo. Finalmente, también sería interesante contemplar la posibilidad de introducir mecanismos que permitan la trazabilidad de los escenarios FVS con la especificación de los atributos de calidad mediante escenarios.

Con respecto a FVS como lenguaje, como mejora futura también es posible que este lenguaje tenga el potencial expresivo para poder expandirse a un nivel más de abstracción para por ejemplo, denotar al mismo tiempo el modelado de features y variabilidad, todo en mismo “esquema” o regla.

8. Resumen y Conclusiones

En el presente trabajo se presenta un análisis detallado del concepto de Familia de Productos y de las técnicas existentes para su modelado arquitectónico. En este sentido se han detectado diversas limitaciones en los enfoques actuales.

La primera está relacionada con el concepto de variabilidad, el cual no ha sido tratado como un aspecto de primer nivel, causando así debilidades en la arquitectura resultante. La mayoría de las aproximaciones se enfoca en manejar variabilidad únicamente a nivel externo, removiendo/sacando/modificando componentes o conectores. Sin embargo, estos enfoques dejan de lado la variabilidad interna de un componente.

Una segunda limitación está relacionada con el nivel de abstracción necesario en una descripción arquitectónica. Muchas aproximaciones terminan utilizando artefactos de software cercanos al código, lo cual puede llevar a tomar decisiones de manera prematura.

Finalmente, la falta de una notación homogénea para describir la arquitectura también es señalado como un problema.

Dado este contexto, se definieron características deseables que debe contar un lenguaje para la descripción arquitectónica de familias de productos. Debe poder especificar variabilidad interna de componentes, debe permitir explorar el comportamiento arquitectónico manteniendo altos niveles de abstracción, y exhibir una notación homogénea.

Como primer paso en lograr tal lenguaje se exploró el lenguaje declarativo FVS. **FVS posee exhibe propiedades que lo hacen atractivo en un dominio arquitectónico**. Cuenta con una notación sólida, flexible y compacta, con una semántica clara, y es lo suficientemente expresivo. Además, es particularmente útil para explorar comportamiento en etapas tempranas, propiedad necesaria en dominios arquitectónicos. Se presentó el modelado arquitectónico de un caso de estudio que tuvo resultados satisfactorios.

Finalmente, el trabajo futuro menciona futuras líneas de investigación que buscan consolidar FVS como lenguaje de modelado arquitectónico en familias de productos.

Referencias

[1] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture", ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

[2] D. Garlan, "Formal Modelling and Analysis of Software Architecture: Components, Connectors, and Events", Formal Methods for Software Architectures, LNCS 2804, September 2003.

[3] Dewayne E. Perry - Alexander L. Wolf "Foundations for the Study of Software Architecture" AT&T Bell Laboratories 600 Mountain Avenue Murray Hill, New Jersey 07974 y Department of Computer Science University of Colorado Boulder, Colorado 80309

[4] David Garlan, "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events"Carnegie Mellon University, Pittsburgh PA 15213, USA,

[5] Marcos Raul Cordoba Boyas, "Lineas de Productos de Software", Facultad de Ingeniería de Sistemas, escuela Politécnica Nacional, Quito Ecuador

[6] Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: Proceedings of WICSA/ECSA 2009, pp. 221–230. IEEE (2009).

[7] K. Pohl, G. Böckle, F. van der Linder, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag, 2005.

[8] M. Razavian and R. Khosravi, "Modeling variability in the component and connector view of architecture using UML", Intern. Conference on Computer Systems and Applications (AICCSA), IEEE/ACS, pp.801-809, 2008.

[9] F. Bachmann, L. Bass, "Managing Variability in Software Architectures", SSR'01, Canada, May 2001.

[10] Díaz, J., Pérez, J. , Garbajosa, J. Wolf, A., "A process for documenting variability design rationale of flexible and adaptive PLAs". In the Move to Meaningful Internet Systems: OTM 2011 Workshops, pages= 612--621, 2011, Springer.

[11] IEEE (1990). Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990.

[12]S. Robertson and J. Robertson (1999). Mastering the Requirements Process. ACM Press.

[13] A. Davis (1993). Software Requirements: Objects, Functions and States. Prentice Hall.

[14] Axel Van Lamsweerde. 2001. Goal-Oriented Requirements Engineering: A Guided Tour. InProceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE '01). IEEE Computer Society, Washington, DC, USA, 249-.

[15] M. Glinz. On non-functional requirements. In Proc. RE, volume 7, pages 21–26. Springer, 2007.

[16] Ivar Jacobson, Magnus Christerson, Patrik Jonsson & Gunnar Overgaard 1992. Object-Oriented Software Engineering: A Use Case Driven Approach (ACM Press). Addison-Wesley, 1992, ISBN 0-201-54435-0

[17] Len Bass, Paul Clements, and Rick Kazman. 1998. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[18] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. 2002. *Documenting Software Architectures: Views and Beyond*. Pearson Education.

[19] Krueger, Ch. Introduction to Software Product Lines. 2006. En http://www.softwareproductlines.com/

[20] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. 2004.

[21] Sametinger, J. (1997). Software engineering with reusable components. Springer Science & Business Media.. [22] Van Group J., and BoschJ., Design Erosion: Problems and Causes, Journal of Systems & Software, 2002.

[23] Pohl K., Bockle G., and Van der Linden F. Software Product Line Engineering: Foundations, Principles and Techniques. Spring 2005.

[24] Garcia F.J., Barras J.A., Laguna M.A. y Marques J.M., Líneas de Producto, Componentes, Frameworks y Mecanos. Informe Técnico DPTOIA-IT-2002-04, Universidad de Valladolid, España, 2002.

[25] Bosch J., Design & Use of Software Architectures: Adopting and Involving a Product-Line Approach. Addison-Wesley, 2000.

[26] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, 5(1), 143-168.

[27] Czarnecki K., Helsen S., and Eisenecker U., Staged Configuration Through Specialization Multi-Level Configuration of Feature Models. Software Process Improvement and Practice, special issue on "Software Variability: Process and Management", 10(2), 2005, pp.143 – 169.

[28] Feature Oriented Domain Analysis (FODA) Feasibility Study, Kyo C. Kang Sholom G. Cohen James A. Hess William E. Novak A. Spencer Peterson, Noviembre 1990.

[29] B. Magro, J. Garbajosa, J. Pérez, "The Development of a Software Product Line for Validation Environments Software", Product Line Engineering, Ed. Taylor y Francis, 2009.

[30] U. Assmann, Invasive Software Composition, Springer- Verlag, Heidelberg, 2003.

[31] G. Kiczales et al., "An Overview of AspectJ", El 15° ECOOP, LNCS 2072, Budapest, Hungria.

[32] Pérez, J., Ali, N., Carsí, J. A., & Ramos, I. (2006). Designing software architectures with an aspect-oriented architecture description language. In Component-Based Software Engineering (pp. 123-138). Springer Berlin Heidelberg.

[33] Rashid, A., Sawyer, P., Moreira, A., & Araújo, J. (2002). Early aspects: A model for aspect-oriented requirements engineering. In Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on (pp. 199-202). IEEE.

[34] Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., & Kulesza, U. (2006, March). Towards an integrated aspect-oriented modeling approach for software architecture design. In 8th Workshop on Aspect-Oriented Modelling (AOM. 06), AOSD (Vol. 6).

[35] Rashid, A., Moreira, A., & Tekinerdogan, B. (2004). Special Issue of Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design.IEE Proceedings: Software, 151(4), 153-156.

[36] Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA

[37] Asteasuain, F., & Braberman, V. Specification patterns: formal and easy. International Journal of Software Engineering and Knowledge Engineering (IJSEKE). Print ISSN: 0218-1940. En prensa. 2015.

[38] Autili, M., Inverardi, P., & Pelliccione, P. (2007). Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engineering, 14(3), 293-340.

[39] Smith, M. H., Holzmann, G. J., & Etessami, K. (2001). Events and constraints: A graphical editor for capturing logic requirements of programs. In Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on (pp. 14-22). IEEE.

[40] Bass, L. (2007). Software architecture in practice. Pearson Education India.

[41] Raúl Puerta Sánchez (Julio 2011) Soporte a la Trazabilidad en el desarrollo de Líneas de Productos de Software - UNIVERSIDAD POLITÉCNICA DE MADRID

[42] Brown, A. W. (2000). Large-scale, component-based development (Vol. 1). Englewood Cliffs: Prentice Hall PTR.

[43] Asteasuain, F., & Braberman, V. (2010). Specification patterns can be formal and also easy. In The 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE).

[44] I.Montero, S.Segura,Programa de doctorado: Tecnología e Ingeniería del Software. Universidad de Sevilla. Departamento de Lenguajes y Sistemas Informáticos. 2006.

[45] Lee, K., Kang, K. C., Kim, M., & Park, S. (2006). Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In Software Product Line Conference, 2006 10th International (pp. 10-pp). IEEE.

[46] Griss, M. L., Favaro, J., & d'Alessandro, M. (1998, June). Integrating feature modeling with the RSEB. In Software Reuse, 1998. Proceedings. Fifth International Conference on (pp. 76-85). IEEE.

[47] Van Lamsweerde, A. (2003). From system goals to software architecture. In Formal Methods for Software Architectures (pp. 25-43). Springer Berlin Heidelberg.

[48] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., & Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. Software Engineering, IEEE Transactions on, 21(4), 314-335.

[49] Jonás A. Montilva C., Ph.D. IEEE Member Universidad de Los Andes Facultad de Ingeniería Departamento de Computación Mérida – Venezuela. Desarrollo de Software Basado en Líneas de Productos de Software.