DESARROLLO DE FAMILIAS DE PRODUCTOS BASADO EN MDA PARA SISTEMAS TELEMÁTICOS

Yuli Garcés Bolaños
Alejandra Reyes Reina

Monografía presentada para optar al título de Ingeniero en Electrónica y Telecomunicaciones

Director
Dr. Rodrigo Cerón

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Línea de investigación en Ingeniería de Sistemas Telemáticos
Departamento de Telemática
Popayán, Enero de 2010

NOTA DE ACEPTACIÓN

_____

_____

_____

_____

PRESIDENTE DEL JURADO

_____

JURADO

Popayán, 2009

ÍNDICE DE CONTENIDO

Página

ÍNDICE DE FIGURAS

Página

ÍNDICE DE TABLAS

Página

CAPÍTULO 1.

INTRODUCCIÓN

1.1. Contexto

La tendencia actual del mundo hacia la globalización plantea retos económicos que llevan a las empresas a buscar las mejores oportunidades que les posibiliten obtener una mayor competitividad y productividad. Para el logro de estos objetivos hay cuatro aspectos básicos a ser considerados: una nueva división del trabajo, mejor utilización de los recursos disponibles, el aprovechamiento de economías de alcance y mayor incorporación de tecnología. En este contexto, el desarrollo tecnológico y, en particular, el software, constituyen opciones viables para mejorar la productividad y competitividad de las empresas. Hoy está en auge la integración de aplicaciones basadas en software a los diferentes modelos de negocio, incluso las pequeñas y medianas empresas cuentan con la posibilidad de contratar soluciones software a la medida, éstas posibilitan su crecimiento permitiendo que los montos invertidos puedan recuperarse en el corto plazo, obteniendo el retorno máximo de la inversión en tecnología [1].

El software es una de las principales bases de soporte para todo tipo de negocios. En muchas empresas pasó de automatizar unas tareas básicas de contabilidad, a desempeñar otras tareas más complejas y variadas, como realizar operaciones transaccionales, servir como sistema de apoyo a la toma de decisiones y facilitar el proceso de innovación a través de sistemas estratégicos [2]. Sin embargo, no siempre es un éxito, a veces los resultados obtenidos no satisfacen a los usuarios finales o no consiguen los beneficios económicos esperados por las empresas que lo desarrollan. Algunas de las razones por las cuales esto sucede son que los productos software no finalizan en el plazo fijado, no se ajustan al presupuesto inicial, no cumplen con las especificaciones, los resultados finales son de baja calidad [3] o simplemente puede deberse a que la industria del software se caracteriza por la rapidez con la que cambian y surgen nuevas tecnologías y/o plataformas, a las cuales las empresas se tienen que ir adaptando y que en muchas ocasiones se ven obligadas a adoptar, aun cuando éstas sobrepasan en gran medida la capacidad de las mismas para manejarlas. El ritmo acelerado al que evolucionan las tecnologías software y el deseo apremiante de la industria por absorber esos cambios, ha planteado nuevos retos en el desarrollo software. Generalmente las aplicaciones requieren implementaciones que se adapten a los constantes cambios, que faciliten su integración con otros sistemas y que resuelvan problemas relacionados con calidad. Frente a esta necesidad, la realidad es que el precio que las empresas pagan por la posibilidad de migración, interoperabilidad e independencia de plataforma para sus aplicaciones, se traduce actualmente en un aumento desmesurado en los costos de desarrollo [4].

La Arquitectura Dirigida por Modelos o MDA (Model Driven Development) es una iniciativa del OMG (Object Management Group) orientada a lograr dichos objetivos; intenta proteger parte de la inversión de las empresas en el desarrollo del software, aislando la lógica de la aplicación de las reglas del negocio, de la tecnología y las plataformas sobre las cuales el sistema será implementado. La iniciativa MDA cubre un amplio espectro de áreas de investigación, como creación de metamodelos basados en estándares del OMG, construcción de modelos, transformaciones de modelos, definición de lenguajes de transformación, construcción de herramientas de soporte, etc. y aunque algunas de estas áreas están bien fundamentadas y en la práctica se aplican con éxito, otras están todavía en proceso de definición. En este contexto es necesario que el ámbito académico y empresarial unan sus esfuerzos y conviertan a MDA junto con sus conceptos y metodologías relacionadas, en un enfoque coherente, basado en

estándares abiertos, y soportado por técnicas y herramientas maduras, que realmente aumenten el nivel de abstracción.

1.2. Escenarios de motivación

En el contexto planteado en la sección anterior la reutilización se convierte en una de las mejores opciones para alcanzar algunos de los objetivos que plantea MDA, como disminuir los plazos de entrega de los productos software, mejorar su calidad y reducir los costos de desarrollo y mantenimiento. La reutilización es una alternativa para el desarrollo de productos software de una forma rápida, eficiente y productiva, debido a que las aplicaciones no inician desde cero. Entre las ventajas que ofrece MDA y el concepto de reutilización de software, se puede destacar el aumento de la confiabilidad hacia el usuario final, gracias a la experiencia adquirida a través de la implantación de software de proyectos anteriores en proyecto actuales; la reducción de los riesgos en el proceso de desarrollo y la disminución de la incertidumbre sobre el costo de desarrollo, debido a que ésta se limita solo a lo nuevo, lo cual reduce el margen de error en los procesos de estimación. Por último, cabe agregar que la reutilización libera a los desarrolladores del trabajo repetitivo y permite que dediquen su tiempo a tareas más productivas y novedosas [5].

Aunque algunas estimaciones realizadas en los años 80 pronosticaban que el 60% de una aplicación informática se desarrollaría ensamblando componentes reutilizables, el nivel de reutilización alcanzado hoy día es bastante inferior. Por ejemplo, un estudio publicado en 2005 reveló que el porcentaje de reutilización logrado en 25 proyectos de la NASA (National Aeronautics and Space Administration) era del 32%. Muchos autores consideran que este fracaso se debe a que es un modelo de desarrollo de software que requiere un entorno soportado por tecnología que aún no está madura. Además, la mayoría de los procesos de desarrollo de software, persiguen la construcción de productos aislados. Al no disponerse de escenarios suficientemente amplios como para detectar con precisión qué elementos son reutilizables y cuáles son las situaciones donde pueden aprovecharse, se desemboca en una reutilización oportunista del software. Para que la reutilización del software sea sistemática, los procesos de desarrollo deberían abordar la construcción colectiva de Líneas de Productos Software o SPL (Software Product Line) relacionadas por un dominio [5].

J. Greenfield y K. Short [5] han llegado a conclusiones similares sobre las líneas de productos al tratar de aplicar en el desarrollo de software los principios de economía de escala y de alcance. Al respecto afirman que la fabricación industrial de un producto consta fundamentalmente de dos etapas, la fase de desarrollo, donde se crean el diseño del producto y unos pocos prototipos para la validación del diseño y la fase de producción, donde se crean de forma masiva ejemplares del producto. La economía de escala ocurre sobre todo durante la fase de producción. En ella se realiza la fabricación de múltiples unidades de un mismo producto. Cuanto más se produce menores son los costos debido a diversas causas, entre ellas la repartición de los costos fijos entre las unidades producidas y la mejora tecnológica. La economía de alcance ocurre específicamente durante la fase de desarrollo. En ella se realiza la fabricación colectiva de productos similares. Permite ahorrar recursos y costos como consecuencia de producir dos o más productos de forma conjunta, resolver problemas comunes en la fabricación de los diversos productos una sola vez y compartir activos tangibles (por ejemplo, la tecnología) e intangibles (por ejemplo, la marca) entre los diferentes productos [6]. La economía de alcance es aplicada al concepto de SPL, ya que como señala P. Wegner [6], la naturaleza esencialmente lógica del software hace que los costos estén concentrados en esta etapa, porque el costo de producir las copias de un desarrollo software, es despreciable comparado con el costo de desarrollo del mismo.

Las SPL, al igual que MDA, proponen aumentar el nivel de abstracción, aunque con un enfoque diferente, orientado más hacia metas económicas (economías de alcance) antes que a metas técnicas, este hecho sumado al escenario descrito anteriormente, pone en evidencia las grandes ventajas de orientar las organizaciones hacia la producción de familias de sistemas y no a la producción de productos software aislados (ver figura 1).

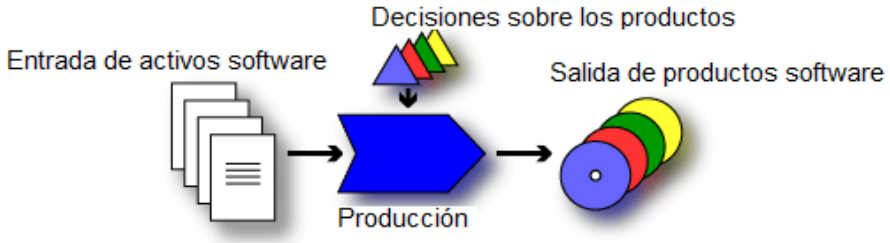

Figura 1. Cambio en la producción, de productos software aislados a SPL.

La historia del desarrollo de software es una historia de aumentos en los niveles de abstracción. Por lo general los desarrolladores trabajan continuamente con diferentes lenguajes, y así como se cambian de un lenguaje a otro, también cambian e incrementan el nivel de abstracción en el cual trabajan, lo que implica que el desarrollador aprende un nuevo lenguaje de más alto nivel que podría ser trasladado a lenguajes de niveles más bajos.

En principio, cada uno de los nuevos niveles más altos de abstracción son introducidos solo como un concepto. Los primeros lenguajes de ensamblador fueron, sin duda alguna, inventados sin el beneficio de un ensamblador automático que convirtiera las órdenes en bits, y los desarrolladores fueron agrupando funciones junto con los datos que ellos encapsulaban, mucho antes de que hubiese un esfuerzo por automatizar este concepto. Similarmente, los conceptos de programación estructurada fueron pensados antes que los lenguajes de programación estructurada fueran de uso extendido. Con el tiempo, los nuevos niveles de abstracción fueron formalizados y las herramientas tales como ensambladores, pre-procesadores y compiladores, fueron construidas para soportar estos conceptos, lo cual tiene el efecto de esconder los detalles de los niveles más bajos que solo unos pocos expertos (por ejemplo, los desarrolladores de compiladores) necesitan conocer. Con el transcurso del tiempo es posible aprender cómo aplicar un conjunto de convenciones para usar dicho lenguaje, estas convenciones son formalizadas y un lenguaje de alto nivel nace y es trasladado automáticamente al lenguaje de más bajo nivel, y así sucesivamente. Claramente esto forma un patrón, el conocimiento de una aplicación se formaliza en un nivel de abstracción tan alto, como el lenguaje permita hacerlo [7].

Este trabajo representa un estudio y analiza el estado del arte de los dos enfoques de desarrollo de software mencionados, la Arquitectura Dirigida por Modelos y las Líneas de Productos Software, que han sido pensados por sus creadores y promotores para atender necesidades específicas que son acomodadas perfectamente a los escenarios planteados, los cuales corresponden a un contexto real y actual. Haciendo uso de los conceptos, metodologías, tecnologías y técnicas que proponen, será presentada una arquitectura en un nivel de abstracción tan alto como sea necesario (de acuerdo a lo expresado anteriormente), que de soporte a las familias de sistemas basadas en un enfoque MDA.

1.2. Definición del problema

La economía global es en los últimos días una fuerza dominante del mundo, las posturas ideológicas y la acción política influyen en un alto grado en las relaciones internacionales y en

el orden económico imperante en la actualidad. La globalización entendida como la ausencia de los estados en el contexto mundial, permite que exista un crecimiento económico como respuesta a las necesidades humanas apoyadas en la acción individual que les permite conseguir aquello que es aparentemente necesario [1].

El crecimiento continuo de la demanda de productos software, debido a la fuerza de la economía global, muestra la necesidad existente en los usuarios de adquirir sistemas de calidad y útiles en cuanto a los servicios que prestan. Gracias a ello es posible encontrar software, en el mercado, que intenta satisfacer las necesidades de los usuarios; ellos son afectados por la falta de calidad y por ello, demandan más y mejores productos a largo plazo, intentan alcanzar beneficios pero corren riesgo de no satisfacer sus necesidades iniciales. Problemas como la transparencia al usuario, las pérdidas innumerables de tiempo, los costos y la calidad están presentes en los proyectos que a largo, mediano o corto plazo pierden en forma parcial o total su funcionalidad para el mercado o entorno para el cual fueron desarrollados. El software cumple un rol en la infraestructura social; por ello la facilidad de uso de los sistemas interactivos, entendidos como un atributo de calidad fundamental [2], la aparición de nuevas tecnologías, dispositivos móviles y servicios inteligentes suponen el inicio de la exigencia de desarrollos software de calidad, reutilizables, fáciles de usar, multiplataforma, seguros y, por supuesto, medibles en cuanto a tiempo y costos de desarrollo se refiere [3].

De acuerdo con el Standish Group [4], en Estados Unidos, de 175000 proyectos cada año, el 16.2% finaliza a tiempo y bajo el presupuesto planteado, el 31.1% son cancelados por problemas de calidad y el otro 52.7% excede su presupuesto original en un 189% en promedio. Además solo un 42% de los proyectos terminados satisfactoriamente cumplen los requisitos planteados originalmente, datos asombrosos que muestran que en la actualidad existen más fracasos que hace unos años atrás [5].

Aunque parece que el desarrollo de software no cuenta ni contará con capacidad para abastecer la demanda total de los usuarios, es desde finales de los 90's que se visualizan otros enfoques dando inicio al uso de técnicas, modelos y arquitecturas que permiten hacer viable la reutilización de software, diseñando sistemas con un grado de calidad cada vez mayor. Dentro de los enfoques con mayor y mejor aceptación entre los desarrolladores se encuentra la Arquitectura Dirigida por Modelos o MDA (Model Driven Architecture); apoyada por OMG (Object Managment Group) [6], cuyo énfasis es el uso eficiente de modelos en el proceso de desarrollo de software, y la creación de grupos de sistemas lo que permite la reutilización de software. MDA es una forma de organizar y administrar la arquitectura, es soportada por herramientas automatizadas y ofrece servicios tanto para la definición de los modelos así como para las transformaciones entre sus diferentes tipos [6]. Otra técnica novedosa, adoptada por los desarrolladores, es conocida como familia de productos, Línea de Productos Software o familia de sistemas. En ella las organizaciones obtienen un alto grado de eficiencia cuando las características comunes, o los patrones de productos, pueden definirse por medio de profesionales de alta calificación. Promueve la reutilización proactiva de activos, sobre todo en áreas como las Tecnologías de la Información (IT), ayuda a mejorar el nivel de automatización de las organizaciones dando solución a gran cantidad de problemas relacionados con los sistemas bajo desarrollo [7].

Las familias o líneas de producto son un grupo de sistemas relacionados, los cuales vistos en conjunto cubren un segmento de mercado determinado, están enfocados en muchas aplicaciones y son guiados por análisis de mercado y planes de negocio. En ellas es importante determinar los aspectos comunes y variables de la familia, los requisitos comunes y específicos

de la misma, consiguiendo de esta forma construir familias de sistemas que compartan un conjunto de componentes denominados núcleo que tienen un costo inicial de establecimiento y permiten, a su vez, reducir el tiempo en el desarrollo de nuevos productos gracias a su soporte en componentes reutilizables [8].

De lo expuesto anteriormente es posible concluir que es necesario obtener abstracciones compactas e independientes de plataforma, con facilidad de migración (portability), de calidad, seguras y eficientes, que permitan desarrollar productos software reutilizables, medibles en tiempo, costos y primordialmente, que permitan abastecer la demanda exponencial de los clientes y, a la vez, lograr su satisfacción; debe estar en la mira la productividad de los desarrolladores debido a que la necesidades crecientes de esta industria lo imponen como una de sus restricciones [3], [9].

El presente trabajo de grado pretende unir el enfoque de desarrollo de sistemas haciendo uso de las familias de productos con la base generada por MDA. Presenta información detallada de las ventajas, desventajas, estándares y características de la implantación de esta nueva tendencia de desarrollo de software; y permite responder en forma parcial las siguientes preguntas de investigación:

- ¿Es favorable la aplicación del enfoque de desarrollo de familias de productos para los nuevos desarrollos de sistemas en el que el Software sea uno de sus componentes principales?
- ¿Es aplicable MDA en el desarrollo de familias de productos?
- ¿Es viable el desarrollo de extensiones a herramientas que permitan generar software base para familias de productos?

1.3. Solución propuesta

Para dar respuesta a las tres preguntas de investigación planteadas en la sección anterior (definición del problema), este trabajo de grado desarrolló un modulo funcional que siguiendo el enfoque de la Arquitectura Dirigida por Modelos, realiza la transformación de Modelos Independientes de Plataforma o PIM (Plataform Independent Model) a PIM más detallados, y permite la gestión de la variabilidad de diagramas de clases de una Línea de Productos Software de entrada, para obtener a la salida un diagrama de clases de un producto en especifico, de acuerdo al mecanismo de implementación de variabilidad conocido con el nombre de derivación arquitectónica. El diagrama de clases es modelado con una herramienta ofrecida para Eclipse en su versión 3.5 (Galileo) y anteriores, conocida como eUML2, compatible con UML 2.1, XMI 2.0 y java para Eclipse 3.4. (Ganymede). El modelo es exportado al bloque funcional en un archivo JAVA/XMI que es editado por una herramienta o plugin creado sobre Eclipse 3.4, insertando un valor según la variabilidad de la clase en cuestión sobre su etiqueta-valor, de acuerdo con unas reglas lógicas y de selección previamente definidas y a las decisiones aportadas por el usuario. Esta herramienta deriva estas clases y finalmente entrega un diagrama de clases de un producto específico.

1.4. Contribuciones

El presente trabajo de grado introducirá dos nuevas metodologías de desarrollo de software, las Líneas de Productos Software y la Arquitectura Dirigida por Modelos y las analizará como alternativas viables al desarrollo clásico. Las contribuciones aportadas a la comunidad de desarrollo de software y al estado de arte de estos dos enfoques son las siguientes:

- Estudio y análisis del estado del arte de SPL y MDA.

- Definición de una arquitectura que de soporte a las familias de sistemas basadas en un enfoque que utilice MDA.
- Verificación de la arquitectura definida con el desarrollo de un sistema telemático basado en tecnología de componentes (EJB, CCM, etc.)

1.5. Estructura del documento

El presente documento está estructurado de tal forma que describe el proceso llevado a cabo para la realización de este proyecto de grado. Los conceptos estudiados y analizados son explicados paso a paso, al igual que las tareas desarrolladas para la obtención del resultado final, detallando los aspectos más significativos del trabajo realizado.

En primer lugar, en el capítulo siguiente son repasados brevemente los conceptos teóricos más relevantes, sobre todo los que tienen que ver con los enfoques de desarrollo de software llamados Arquitectura Dirigida por Modelos y Líneas de Productos Software; también son descritos los principales proyectos relacionados con estos dos enfoques, bien sea aquellos que han sido desarrollados o que actualmente están en desarrollo, los objetivos que persiguen, los aportes que realizan al estado del arte de estos enfoques y al trabajo de grado.

El tercer capítulo estudia y analiza el estado del arte de los aspectos fundamentales de las Líneas de Productos Software, desde diferentes puntos de vista. Son descritos los principales avances logrados en cuanto a la definición de conceptos, metodologías y procesos para la implementación de una Línea de Productos Software.

El cuarto capítulo estudia y analiza el estado del arte de los aspectos fundamentales de la Arquitectura Dirigida por Modelos. Describe los estándares, procesos, herramientas y métodos que propone para el desarrollo del software y los avances logrados en cuanto a la definición de un entorno que permita escribir especificaciones basadas en un modelo independiente de plataforma.

Para reflejar todo el trabajo realizado de una forma práctica, el quinto capítulo plantea y describe una arquitectura para dar soporte a las familias de sistemas basadas en un enfoque que utilice MDA.

El sexto capítulo muestra la verificación de la arquitectura definida en el capítulo anterior con el desarrollo de un sistema telemático (videojuegos) que sirva de ejemplo tangible de la aplicación del proyecto.

Finalmente, el séptimo capítulo expone las distintas líneas de trabajo que quedan abiertas a partir del presente proyecto y que pueden ser abordadas en trabajos futuros, y a la vez, resume las conclusiones obtenidas con la realización del proyecto.

Adjunto a este documento, en la parte de anexos, están incluidos el modelado de la aplicación construida (ver anexo A), unas indicaciones sobre el modelado de arquitecturas en el contexto de líneas de productos (ver anexo B) y el manual de usuario de la aplicación construida (ver anexo C).

CAPÍTULO 2.

ESTADO DEL ARTE

2.1. Introducción

La arquitectura del software es el común denominador de los dos procesos de desarrollo de software que serán abordados en este proyecto de grado, las Líneas de Productos Software (ver sección 2.2.) y la Arquitectura Dirigida por Modelos (ver sección 2.3.). Ambos enfoques proponen una nueva orientación en el proceso de desarrollo, en ellos las arquitecturas están enfocadas a los objetivos del software a diseñar, antes que a las capacidades del producto que lo contiene; la intención es que no se construyan arquitecturas rígidas que definen requisitos estrictos y que solo dan lugar al desarrollo de unidades de software individuales.
Las siguientes secciones describen, a grandes rasgos, las propuestas que introducen los dos enfoques mencionados, los trabajos relacionados a estos enfoques y las aportaciones que ellos tienen sobre el presente trabajo.

2.2. Líneas de Productos Software

Una familia de productos software o familia de sistemas o Línea de Productos Software o SPL[1] (Software Product Line) es un conjunto de productos construido a partir de unos activos comunes que resuelve las necesidades de un dominio en específico [13], [14].
Debido a que la arquitectura de un sistema software es un elemento básico para el diseño del mismo y es fundamental para medir el éxito de un proyecto de desarrollo, debido a que es el primer elemento facilitador y/o limitante de sus características futuras, el proceso de construcción de una SPL está enfocado en diseñar una arquitectura abierta, configurable y modificable, donde los conjuntos de sistemas que compartan unos objetivos comunes sean desarrollados, más no pensando únicamente en las unidades de software individuales.
Este diseño suele llevarse a cabo mediante una aproximación arquitectónica conocida con el nombre de arriba-abajo. El diseño parte de los requisitos, con ellos una arquitectura de alto nivel es definida y refinada en etapas sucesivas, hasta dar lugar a sus componentes más pequeños. El proceso permite identificar los productos que forman parte de la SPL y a su vez ellos hacen posible identificar los activos comunes (núcleo de la SPL) y los explícitos (variantes), es decir, que no están presentes en todos los productos de la SPL y que son denominados componentes o elementos variables de la arquitectura [15].

Cabe agregar que en cada etapa de esta aproximación arquitectónica, es posible comprobar la eficacia del diseño, pero solo hasta la etapa final, los desarrolladores tienen algo que un cliente puede verificar.

2.3. Arquitectura dirigida por modelos

La arquitectura dirigida por modelos o MDA[2] (Model Driven Development) abarca un conjunto de estándares propuestos por OMG[3] (Object Management Group). Su propósito principal es lograr que el proceso de desarrollo de software esté basado en los modelos del sistema a desarrollar, como característica relevante, ellos parten de los sistemas abstractos

[1] Para más información sobre los conceptos fundamentales relacionados con SPL, remitirse al capítulo 3.
[2] Para más información sobre los conceptos fundamentales relacionados con MDA, remitirse al capítulo 4.
[3] Para más información sobre el OMG, remitirse al capítulo 4.

independientes de la plataforma en la cual va a funcionar. Los procesos de transformación permiten obtener modelos específicos de plataforma y finalmente el código fuente del software del sistema [16].

La arquitectura de MDA, al igual que todos los estándares que promueve OMG, está compuesta por cuatro niveles de abstracción que recogen algunos de los principales estándares del mencionado grupo. Trabaja en todos los aspectos importantes del ciclo de vida del software de los sistemas, desde el análisis de requisitos, hasta la generación del código fuente y el producto final. Los productos finales son obtenidos mediante transformaciones entre los diferentes modelos y los diferentes niveles de abstracción que propone MDA, de tal forma que al igual que en las SPL, el diseño siga una aproximación arquitectónica de arriba-abajo que separa la lógica de negocio de los detalles de implementación.

2.4. Trabajos relacionados

A continuación es posible observar una descripción breve de los principales proyectos relacionados con los dos enfoques de desarrollo de software, en los cuales está centrado el presente trabajo, es decir, SPL y MDA. Para ambos enfoques, los aportes de dichos proyectos al estado del arte y los puntos débiles, son mencionados.

2.4.1. Derivación de modelos en UML para líneas de productos

Este proyecto [15] realiza un estudio práctico de las posibilidades que ofrecen los sistemas basados en el principio de derivación, para el manejo de la variabilidad en líneas de productos. Los aportes que entrega este proyecto son los siguientes:

1. Una revisión sobre los conceptos fundamentales de MDA, del lenguaje unificado de modelado o UML (Unified Modeling Languaje, ver sección 4.4.5.) y del lenguaje extensible de etiquetas para el intercambio de metadatos o XMI (XML Metadata Interchange.).
2. Una herramienta software que pretende automatizar la derivación de modelos UML de una línea de productos, mediante un conjunto de reglas que faciliten la gestión de la variabilidad en las SPL.

La revisión sobre MDA fue realizada de manera muy general, no profundizó en ningún aspecto en particular, ni mencionó conceptos fundamentales relacionados con este enfoque, como la arquitectura con la que trabaja, los modelos que propone para dirigir el desarrollo del software, el metamodelado, las transformaciones, las herramientas y los estándares que promueve.
La herramienta desarrollada solo recibe modelos UML y solo genera modelos UML, es decir, los elementos de entrada y salida de la aplicación son archivos UML. El formato de archivos de entrada es XMI o en su defecto uno propietario que admita conversión a XMI.
Para el control de la aplicación, la herramienta cuenta con una interfaz gráfica basada en ventanas que permite controlar todas las funcionalidades de la aplicación.
La derivación del modelo arquitectónico la realiza incorporando información de variabilidad al mismo, para lo cual añade a los elementos del diagrama, metadatos acerca de la variabilidad, mediante el uso de los valores etiquetados que forman parte de los mecanismos de extensión estándar de UML. Las restricciones a la derivación son un conjunto de reglas lógicas o construcciones semánticas entre elementos del modelo, que permiten analizar si una combinación de elementos variantes, forman parte de una derivación, o no.
Para la programación, compilación, depuración, documentación y ejecución de la aplicación utilizaron la versión 1.4.1. del JDK de Sun Microsystems (Java Development Kit). El entorno de desarrollo principal utilizado, fue IBM Eclipse 2.1. Este entorno no cuenta con una herramienta

de diseño y construcción de interfaces gráficas, razón por la cual, este proyecto recurrió al entorno Sun One Studio, que ofrece soporte para la construcción de interfaces basadas en Swing. Los diagramas UML de entrada y salida de la aplicación fueron construidos utilizando la herramienta Poseidón UML, en las versiones 1.4, 1.5 y 1.6. Poseidón presenta una gran desventaja ya que es una herramienta de modelador propietario y aun cuando existe una versión gratuita del mismo, su funcionalidad es muy limitada.

2.4.2. Automatización en el manejo de modelos UML de arquitectura del software

Este proyecto [14] implementa un entorno de desarrollo para trabajar dentro del ámbito de la arquitectura de software, utilizando el lenguaje de modelado UML. Uno de los objetivos que persigue es reducir el costo en el desarrollo de sistemas, al crear oportunidades reales para el concepto de reutilización de activos, a través de las Líneas de Productos Software.
Los aportes que entrega este proyecto son los siguientes:

1. Revisión sobre el estado del arte de las arquitecturas de software, las líneas de productos, las métricas y atributos de calidad de las arquitecturas de sistemas software, las herramientas de modelado y el lenguaje UML.
2. Un entorno de desarrollo para trabajar con la arquitectura del software.
3. Soporte mínimo para la evaluación de la calidad del software

La revisión sobre el estado del arte de las arquitecturas software y las herramientas de modelado está desactualizada, ya que el desarrollo del trabajo tuvo lugar en el año 2002. En cuanto a las arquitecturas, no menciona el enfoque de MDA y en cuanto a las herramientas, tan solo menciona tres, cuando actualmente existen muchas alternativas en el mercado (ver capítulo 4).
El entorno de desarrollo fue creado utilizando la herramienta Rational Rose desarrollada por la empresa Rational Corporation y el lenguaje de modelado UML 1.4. Ella permite trabajar con la gran mayoría de las características especificadas en el estándar de UML. Sin embargo, solo existe soporte para los estereotipos; ello implica la falta de soporte explícito al resto de mecanismos dentro de su interfaz de usuario, para la creación de los modelos representativos de la arquitectura del sistema.
El soporte para la evaluación de la calidad del software es planteado a partir de un conjunto de métricas de software orientado a objetos y atributos de calidad. Si bien las primeras no necesitan otro sustento que la definición del modelo del sistema sobre el cual aplicar unas "mediciones" que entregarán una estimación de "lo bueno" que es el producto; en el caso de los atributos de calidad, es necesario dotar de nuevas extensiones a los modelos, tarea que en este proyecto abordan recurriendo a los valores etiquetados como mecanismo de extensibilidad para conseguir agregar esta nueva funcionalidad a los modelos UML. Los mecanismos de extensibilidad permiten dar soporte a los diferentes mecanismos de variabilidad que la versión de UML utilizada no brinda.

2.4.3. Engineering Software Architectures, Processes and Platforms for System-Families - ESAPS

ESAPS [17] se trata de un proyecto dentro del programa Eureka[4] ∑! 2023 que hace parte del programa ITEA[5] (Information Technology for European Advancement) proyecto 99005.

[4] Eureka es una red europea orientada al mercado e industria de la investigación y desarrollo o I+D, fue creada en 1985 como una iniciativa intergubernamental y tiene como objetivo mejorar la competitividad europea prestando soporte a los negocios, centros de investigación y universidades que trabajan en proyectos europeos de desarrollo de innovadores productos, procesos y servicios [18].

Tiene como objetivo principal proporcionar un enfoque adaptado a las familias de sistemas, mediante la combinación de áreas tecnológicas como el análisis, definición y evolución en las organizaciones participantes. Las mejoras están enfocadas en distintos dominios de aplicación como salud, comunicación, gestión de servicios públicos, automóviles y equipo de soporte a las oficinas.

Los aportes de este proyecto en los dominios mencionados son los siguientes:
1. Procesos de ingeniería adaptados a familias de sistemas.
2. Métodos y herramientas de ingeniería adaptados a familias de sistemas.
3. Un componente basado en plataformas adaptadas a dominios específicos para familias de sistemas.
4. Requerimientos para que los proveedores de herramientas las adapten a las necesidades de las familias de sistemas.
5. Requerimientos para proveedores de mediadores de dominios específicos o genéricos.

La estructura del trabajo comprende una serie de actividades divididas en seis grandes partes:
1. Análisis y modelado de familias de sistemas.
2. Definición y descripción de familias de sistemas.
3. Derivación de productos y evolución de activos de familias de sistemas.
4. Validación de la tecnología.
5. Divulgación.
6. Gestión.

A cargo del proyecto está un consorcio de destacadas compañías europeas, instituciones de investigación en tecnologías y universidades, dedicadas a cada una de las seis grandes partes en las cuales está estructurado el trabajo. Existen grupos de investigación formados por las compañías que hacen parte del consorcio, ellos están a cargo de los sub-proyectos (más de 20 sub-proyectos), que resultan poco prácticos de abordar (todos) en el presente trabajo. Sin embargo, en este documento es posible encontrar información general de los proyectos sucesores, los más grandes, de ESAPS; ellos son CAFÉ (From Concepts to Application in System-Family Engineering) y FAMILIES (FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering). Para ampliar los conocimientos sobre los sub-proyectos, existe la página oficial de ESAPS [17], en ellas es posible revisar la sección de resultados públicos, la cual contiene información final de cada uno de los trabajos realizados.

Como fue mencionado, el consorcio ESAPS fue estructurado alrededor de numerosas compañías, cada una ofreció un conjunto complementario de posibilidades al proyecto y compartió una necesidad común con las demás por adaptar la tecnología a las familias de sistemas. Esta tecnología ha sido investigada por las compañías en proyectos exitosos de cooperación europea, ellos transfieren esta experiencia a la práctica industrial, actuando como centros regionales de excelencia. Muchas universidades y otras compañías están asociadas con un contratista principal, formando un conglomerado regional, cada grupo actúa como una red, utilizando el conocimiento especializado de cada miembro. El proyecto ESAPS estuvo basado en el concepto de “la industria como laboratorio” y los conglomerados regionales facilitan a las universidades el acceso a los laboratorios. Estos grupos también forman la base

[5] ITEA es un programa estratégico europeo para estimular, coordinar y adaptar la competitividad lograda con la I+D en software a servicios y sistemas software, juntando la industria, las universidades y los institutos de investigación, en proyectos estratégicos. Proporciona y facilita la conexión entre el financiamiento, la tecnología y la ingeniería [19].

para la divulgación local de los resultados del proyecto, así como también el ESI[6] (European Software Institute), que tiene un papel especial en el proyecto, como divulgador de la experiencia a otras compañías europeas.

La divulgación de los resultados conseguidos a través del proyecto fue realizada de tres formas diferentes:

1. La divulgación interna dirigida a empleados de la compañía, miembros del proyecto, ejecutada y controlada por cada compañía.
2. La divulgación externa dirigida a toda la cadena de proveedores de Nokia, Thomson-CSF, Philips y Siemens.
3. ESI a cargo de la divulgación en la industria europea de familias de sistemas, en particular el sector que comprende dominios de negocios con un alto potencial de reutilización.

2.4.3.1. From Concepts to Application in System-Family Engineering - CAFÉ

CAFÉ [20] es un proyecto dentro del programa Eureka ∑! 2023 que hace parte del programa ITEA proyecto ip00004 y que toma los conceptos desarrollados en ESAPS y los lleva a su madurez, de tal forma que éstos pueden ser aplicados en proyectos concretos a partir de los cuales pueden ser derivados métodos y procedimientos de estos conceptos. Este proyecto definió la estructura de los activos, métodos y procesos, para su utilización en herramientas y aplicaciones. CAFÉ es un proyecto en una secuencia, los primeros fueron ARES (Architectural Reasoning for Embedded Systems) y PRAISE (Product-line Realisation and Assessment in Industrial Settings), después ESAPS y finalmente CAFÉ, ver figura 2

Figura 2. Proyectos previos a CAFÉ

Con el fin de ubicar los proyectos en un contexto real, y llevar los conceptos de ESAPS a la aplicación, CAFÉ introduce el modelo BAPO (Business Architecture Process Organisation) que agrupa en cuatro categorías, las principales áreas que influencian el desarrollo del software, de la siguiente forma:

- Área de negocios: la forma en que se logran ganancias con los productos resultantes.
- Área de arquitectura: la tecnología necesaria para construir el sistema.
- Área de organización: la organización en la cual el software es desarrollado.
- Área de procesos: las responsabilidades y dependencias durante el desarrollo software.

El proyecto ESAPS desarrolla conceptos para la ingeniería de línea de productos, basado en un núcleo de proceso que ya fue descrito y que consiste en seis actividades principales. CAFÉ está basado en el mismo núcleo de procesos pero enfocado en especial en las primeras y las últimas actividades, las cuales aborda en trece sub-proyectos que se pueden consultar en la página oficial de CAFÉ [20], en la sección de resultados públicos.

[6] ESI es una organización ubicada en el centro tecnológico privado Tecnalia (ubicada en Zamudio - País Vasco Español), es una fundación sin ánimo de lucro creada en 1993 por la comisión europea con el soporte del gobierno Vasco, su misión es contribuir al desarrollo de la sociedad de la información e incrementar la competitividad industrial por medio del conocimiento, innovación, mejoramiento continuo, promoción y divulgación de la información sobre la tecnología. Las principales actividades del ESI están centradas en el apoyo al logro de los objetivos de la industria del software de alcanzar desarrollos de mayor calidad, a tiempo, de la mejor forma y al menor costo [21].

Los resultados del proyecto CAFÉ, abarcan la estructura completa de los activos y el conocimiento sobre métodos y procedimientos que serán utilizados para desarrollar herramientas y aplicaciones. La divulgación de los resultados de CAFÉ obedece al mismo procedimiento planteado para ESAPS. Los aportes que entrega este proyecto son los siguientes:

1. Aplicación de los conceptos de ESAPS.
2. Desarrollo de métodos y procedimientos para los conceptos de ESAPS.
3. Desarrollo de herramientas y aplicaciones para la ingeniería de línea de productos.

La estructura del trabajo comprende una serie de actividades divididas en cinco grandes partes:

1. Adopción de las familias de sistemas.
2. Construcción de activos.
3. Utilización de activos.
4. Validación y pruebas.
5. Divulgación

2.4.3.2. FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering - FAMILIES

FAMILIES [22] es un proyecto dentro del programa Eureka ∑! 2023 que hace parte del programa ITEA proyecto ip02009. En FAMILIES se consolidan los resultados de ESAPS y CAFÉ. El modelo BAPO es analizado y algunos problemas pendientes son solucionados. Este proyecto define un entorno para analizar los aspectos económicos de la reutilización, un modelo de madurez para los procesos de familias de sistemas, aspectos arquitectónicos relacionados con la solución de los requisitos de calidad que tienen que ver con las características del negocio y una metodología que soporta la separación de los aspectos de dominio, técnicos y tecnológicos, en el marco de estandarización de MDA.
FAMILIES es el proyecto final en una secuencia de proyectos de ITEA, es el siguiente a CAFÉ (ver figura 3). Ha logrado que la comunidad europea sea reconocida en el tema de la ingeniería de familias de sistemas. En la actualidad, la comunidad europea está a la cabeza en este campo, incluso sobre su contraparte americana, la iniciativa de líneas de productos del Carnegie Mellon SEI[7] (Software Engineering Institute).

El proyecto FAMILIES dirige sus esfuerzos al crecimiento de la comunidad, la consolidación de los resultados basados en hechos de gestión para las prácticas de FAMILIES y sus proyectos precedentes, y en la exploración de los campos que no se cubrieron en los proyectos previos, con el fin de completar el entorno para las líneas de productos.

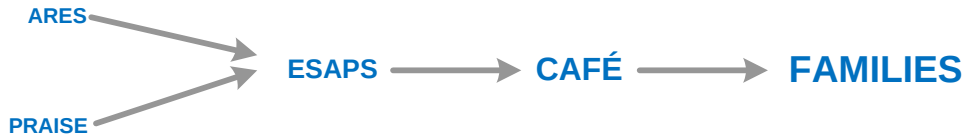

Figura 3. Proyectos previos a FAMILIES

FAMILIES comprende dieciséis sub-proyectos que pueden ser consultados en la página oficial de FAMILIES [22] en la sección de resultados públicos. La divulgación de los resultados de

[7] El Carnegie Mellon Software Engineering Institute es un instituto de Estados Unidos de América que sirve a dicho país como centro de desarrollo e investigación financiado por el gobierno federal o FFRDC (Federally Funded Research and Development Center). El SEI ha avanzado en los principios y prácticas de la ingeniería del software, trabajando en la seguridad computacional y en la mejora de procesos. Como parte de la Universidad Carnegie Mellon el SEI es líder en la innovación técnica. El SEI trabaja de la mano de organizaciones gubernamentales, de defensa, de la industria y de la academia [23].

FAMILIES obedece al mismo procedimiento planteado para ESAPS. Los aportes que entrega este proyecto son los siguientes:

1. Un entorno para las familias de sistemas.
2. Un modelo para las familias de sistemas.
3. Patrones, estilos y reglas, relacionados con la calidad de las familias.
4. Una metodología (procedimientos, herramientas, directrices y ejemplos) para soportar la separación entre los aspectos de dominio, los aspectos técnicos y los aspectos tecnológicos en modelos coherentes, acorde con los principios de MDA.
5. Extensión de los conceptos de reutilización a las organizaciones.

La estructura del trabajo comprende una serie de actividades divididas en seis grandes partes:

1. Transición a familias de sistemas y reducción de costos con la reutilización.
2. Madurez de la familia de sistemas.
3. Calidad de la familia.
4. Ingeniería de familias dirigida por modelos.
5. Integración de familias.
6. Divulgación.

2.4.4. MDA e Ingeniería de Requisitos para Líneas de Producto

Este trabajo [24] explora el papel que puede jugar MDA en la construcción de modelos de dominio (expresados como grafos de características) y su posible transformación en modelos arquitectónicos. Presenta un enfoque que tiene como punto de partida un modelo de metas, de tal forma que la imagen global es una secuencia de transformaciones (no necesariamente automáticas) de modelos: de metas a características y de ambos a una arquitectura inicial para una línea de productos. La trazabilidad de estas transformaciones representa al mismo tiempo una necesidad y un beneficio adicional.

El planteamiento de este trabajo pretende suavizar el paso de un modelo de desarrollo convencional a otro que aproveche las ventajas de las líneas de productos, utilizando un modelo representado con SPEM[8] (Software Process Engineering Metamodel), recoge las técnicas específicas de la ingeniería de líneas de productos, en un proceso paralelo al de la ingeniería de aplicaciones. La propuesta consiste en utilizar un enfoque de ingeniería de requisitos orientada a metas, para modelar explícitamente la intencionalidad del sistema.

Los aportes que entrega este proyecto son los siguientes:

1. Exploración de las posibilidades de incorporación de una visión MDA en el proceso de definición, análisis e implementación de la variabilidad en un contexto de desarrollo de Líneas de Productos Software.
2. Un proceso para la determinación de la variabilidad de las líneas de productos.

Este trabajo se basa en un modelo de reutilización que utiliza una estructura compleja basada en tres niveles de abstracción: requisitos, diseño e implementación. Estas tres categorías están en correspondencia con los modelos que propone MDA, así incorpora este enfoque a las líneas de productos. El nivel de requisitos está representado por el modelo de dominio o de negocio, los requisitos de la línea de productos y el modelo de variabilidad.

En el proceso propuesto en este trabajo, utilizan los conceptos de “meta” y “meta menor” para la determinación de la variabilidad de las línea de productos. La ventaja de esta propuesta es que permite analizar los requisitos desde el punto de vista funcional (meta) y no funcional (meta menor), lo que evidentemente se adapta de forma natural al desarrollo de línea de productos.

[8] Para más información sobre SPEM visitar la página http://www.omr.org/technology/documents/formal/spem.htm

Utiliza para ello árboles con conectivas lógicas "y/o" que expresan puntos de variabilidad, es decir, diversas formas de alcanzar la misma meta. Los modelos de variabilidad propuestos para el análisis y desarrollo de la línea de productos son candidatos naturales a ser considerados como PIM en este proyecto.

2.4.5. MDA vs. Factorías de software

Este trabajo [25] plantea e intenta responder algunas cuestiones respecto a la relación entre dos enfoques para el desarrollo de software, MDA y las Factorías de Software[9]. Discute las principales diferencias entre ambas propuestas, sus respectivas ventajas e inconvenientes, la posibilidad de integración e identifica algunos aspectos de interés en el ámbito de la investigación. Las argumentaciones presentadas son ilustradas mediante la aplicación de estas propuestas para la definición de un método de desarrollo de sistemas basados en computación ubicua.

Los aportes que entrega este proyecto son los siguientes:

1. Responder a algunas preguntas que surgen respecto a la relación entre los enfoques MDA y factorías de software.
2. Definir las ventajas e inconvenientes que presentan los enfoques.
3. Desarrollo de un método que integre MDA y las factorías de software.
4. Plantear oportunidades de investigación aplicables a estas propuestas.

Este trabajo hace énfasis en que ambas iniciativas cuentan con el apoyo de actores muy importantes en la industria del software ya que MDA es promovido por las empresas que hacen parte del OMG y las factorías de software son promovidas por Microsoft, y quizá por ello, ambos enfoques parecen estar enfrentados, debido a que los promotores de una, en muchos aspectos son los detractores de la otra, pero que en realidad no se trata de enfoques tan incompatibles. Al tratar de definir las ventajas e inconvenientes de seguir alguno de los dos enfoques, este trabajo propone que la decisión de uno u otro sea tomada en función de las características requeridas por el proyecto a desarrollar. Recomienda emplear MDA cuando la interoperabilidad con otras herramientas, o el uso de herramientas existentes (que sigan lo estándares OMG) sea un factor clave y aconseja emplear factorías de software, cuando existe la intención de construir una serie de sistemas similares y/o se va a trabajar dentro de un dominio determinado.

El método que propone este trabajo hace uso de la estrategia metodológica que proponen las factorías de software y propone construir un lenguaje de dominio específico o DSL (Domain Specific Languaje). De acuerdo a ello esto sería:

- Desarrollo de una línea de productos.
- Construcción de un entorno de implementación para los sistemas que serán desarrollados siguiendo la línea de productos.
- Definición de un DSL que permita capturar los requisitos específicos de cada uno de los sistemas, utilizando las primitivas conceptuales más adecuadas para este tipo de sistemas.

La construcción del DSL se haría de tal manera que proporcione primitivas conceptuales independientes de plataforma y especifique siguiendo las técnicas que propone OMG (creación de un perfil de UML o definición de un metamodelo con MOF).

[9] Las factorías de software constituyen un enfoque para el desarrollo del software promovido principalmente por Microsoft, propone una línea de productos software que configura herramientas extensibles, procesos y contenido para automatizar el desarrollo y mantenimiento de variantes de un producto arquetípico mediante la adaptación, ensamblaje y configuración de componentes basados en frameworks [25].

Siguiendo esta estrategia mixta, el método para sistemas basados en computación ubicua proporciona a PervML[10] como lenguaje para la especificación de dichos sistemas y define el metamodelo MOF (Meta-Object Facility, ver sección 4.4.1.) de dicho lenguaje, un entorno para la implementación de los sistemas y una implementación de PervML utilizando la extensión EMF[11] (Eclipse Modeling Framework) de eclipse.
A partir de esta implementación platean desarrollar un motor de transformación de especificaciones PervML a código, que extienda el entorno. El motor de transformación, a falta del estándar definitivo de OMG, se basa en la herramienta AGG[12], que permite especificar y ejecutar transformaciones de modelos mediante gramáticas de grafos y el motor de plantillas FreeMaker[13]. Este proyecto sigue una estrategia mixta porque los autores consideran que las gramáticas de grafos son una técnica adecuada para definir transformaciones entre metamodelos, mientras que consideran que las plantillas resultan más cómodas para la generación de los archivos de código finales.

Entre los trabajos futuros que propone este proyecto, plantea la posibilidad de crear métodos que sigan las pautas indicadas por estos enfoques, aplicar estos métodos a entornos reales de producción, demostrar que estos enfoques son mejores que los tradicionales, y proporcionar técnicas y herramientas para dar soporte a estos enfoques.

2.4.6. Un entorno para la aplicación de las líneas de productos software

El entorno para la aplicación de Líneas de Productos Software [26] es el resultado del esfuerzo del SEI y su programa RTSS (Research, Technology, and System Solutions Program) por aportar conocimiento a la comunidad del software en la iniciativa de líneas de productos. El documento que tiene consignado los resultados de dichos esfuerzos está disponible en la Web y cuenta con varias versiones. Cada versión representa un intento de crecimiento gradual y captura la última información sobre prácticas exitosas del concepto de SPL.
El SEI ha defendido las ideas que plantea el enfoque de líneas de productos para el desarrollo del software y ha impulsado muchas de las prácticas que caracterizan a este enfoque. Actualmente está trabajando por hacer de la implementación de las SPL una práctica fiable, de bajo riesgo y benéfica, que combine la orientación de los negocios y la técnica, para lograr el éxito. El SEI es parte de una creciente comunidad de investigadores interesados en madurar la ingeniería de líneas de productos.

Además del entorno, hay otros aportes que el SEI pone a disposición del público general:

- Un libro que se titula “Software product lines: practices y patterns” [27] que contiene una versión inicial del entorno, más tres casos de estudio de organizaciones que han implementado líneas de productos y un gran conjunto de aplicaciones de líneas que ayudan a la adopción de este enfoque.
- Un sitio Web de aplicaciones de líneas de productos del SEI, en el es posible descargar las últimas publicaciones, casos de estudio y, a la vez, estar al tanto de los próximos eventos relacionados.

[10] PervML es un lenguaje que proporciona las primitivas conceptuales necesarias para describir los sistemas basados en computación ubicua, desde su análisis (utilizando primitivas como servicio e interacción) hasta su diseño (proporcionando primitivas como dispositivo). Pretende describir fácilmente las necesidades del usuario mediante unos sencillo diagramas [25].

[11] EMF es un framework que soporta la funcionalidad necesaria para el tratamiento de modelos al estilo definido por MOF. Está concebido como un entorno basado en metamodelado de la capa superior (M3) y es llamado Ecore. Sustituye eficazmente al lenguaje subconjunto de MOF 2.0 conocido en su especificación como EMOF (Essential MOF) porque soporta lectura y escritura transparente de serializaciones EMOF y la conversión de este tipo de modelos a Ecore [29].

[12] Para más información sobre AGG visitar la página http://tfs.cs.tu.berlin.de/agg/

[13] Para más información sobre FreeMaker visitar la página http://freemarker.sf.net

- Un documento que se titula “Software Product Line Acquisition - A Companion to Framework for Software Product Line Practice, Version 3.0” [28] para orientar la compra de SPL para aquellas organizaciones que desean adquirir una línea en lugar de desarrollar la propia. Este documento proporciona una amplia información sobre los conceptos fundamentales sobre SPL y sirve como interpretación del entorno desde la perspectiva estricta de las adquisiciones que realiza el gobierno de Estados Unidos de América.
- Un plan de estudios basado en la vasta experiencia del SEI en desarrollo, compra e investigación en líneas de productos, comprende cinco cursos que tienen una duración de dos días, por cada uno de ellos. Su intención es entregar a los profesionales del software, el estado del arte de aplicaciones que están relacionadas con este enfoque, de tal forma que ellos puedan probar dichas aplicaciones, crear las propias y alcanzar metas orientadas a negocios o a la reutilización estratégica del software.

En la version 5.0 del entorno, existen cambios significativos que reflejan las tendencias actuales en la ingeniería del software (el movimiento “código abierto”, los sistemas distribuidos, las arquitecturas orientadas a servicios, el desarrollo dirigido por modelos, etc.) así como también la nueva ola de experiencias en SPL que han traído nuevas prácticas y nuevas referencias. Algunos de los cambios presentes en esta versión del entorno son:

- Una discusión sobre los factores contextuales que influencian el desarrollo del núcleo de activos, las restricciones del producto, restricciones de la producción, estrategia de producción y activos que previamente existen.
- Un estudio más profundo sobre el concepto de plan de producción.
- Preguntas frecuentes o FAQ (Frequently Asked Questions).
- Algunas modificaciones en los nombres dados a conceptos relacionados con áreas prácticas del entorno.

2.4.7. ModelWare

Este trabajo [30] hace parte del IST[14] (Infromation Society Technologies), pretende definir y desarrollar la completa infraestructura requerida para la implementación a gran escala de estrategias del Desarrollo Dirigido por Modelos o MDD (Model Driven Development, ver capítulo 4) y su validación en diferentes dominios de negocios. El proyecto combina innovación en tecnologías de modelado, procesos y metodologías para la ingeniería, desarrollo de herramientas (en dominios específicos y genéricos), estandarización, experimentación y cambios en gestión. ModelWare es una iniciativa soportada por la Conferencia Europea Anual sobre MDA – Fundamentos y Aplicaciones o ECMDA-FA[15] (European Conference on Model Driven Architecture - Fondations and Applications) con la ayuda del OMG.

La adopción exitosa de MDD busca ser garantizada a través de comunidades de proveedores de herramientas, del OMG, colaboradores del “código abierto” y de usuarios finales, entre otros.

[14] IST es una de las temáticas prioritarias en el sexto programa framework de la Unión Europea para la investigación o FP6 (Sixth Framework Programme) y desarrollo tecnológico para el período 2002-2006. Las actividades de investigación europeas están estructuradas alrededor de programas consecutivos llamados Programas Frameworks. Las prioridades son identificadas en base a un conjunto de criterios que reflejan las principales preocupaciones de la Unión Europea en cuanto a aumentar la competitividad industrial y la calidad de vida para los ciudadanos europeos en una sociedad global de la información. Las actividades de investigación en este framework están orientadas a aspectos técnicos (hardware, software, conocimiento) y desafíos de la sociedad (comunicación, trabajo colaborativo, educación) [31].

[15] ECMDA-FA está dedicada a fomentar los conceptos de MDA y a auspiciar la industrialización de la metodología de MDA. Concentra sus esfuerzos en conducir a las figuras principales de la industria e investigación europea, a un diálogo que tenga como resultado una industria más eficiente y fuerte, que produzca software más confiable, basándose en el estado del arte de los resultados de las investigaciones [32].

Al respecto, ModelWare lanzó el proyecto Eclipse MDDi (Model Driven Develpment integration) dirigido a la construcción de una plataforma MDD abierta, que ofrezca las facilidades de integración necesarias para aplicar este enfoque de desarrollo de software.
Este proyecto produce un entorno extensible y herramientas dedicadas a la integración de herramientas en Eclipse, diseñadas para soportar varios lenguajes de modelado (UML, DSL) y metodologías. Espera que la plataforma de MDDi también facilite el soporte de otros espacios tecnológicos, para crear un ambiente completamente adecuado para MDD. Este entorno simplifica las interacciones entre herramientas para formalizar las operaciones aplicadas sobre modelos, como las transformaciones de modelos o validación de modelos.

La solución ModelWare sigue el enfoque MDA de tal forma que mueve el desarrollo de software de un enfoque centrado en código y documentos, a uno centrado en modelos; se basa en el aprovechamiento de niveles más altos de abstracción en la especificación y diseño de sistemas software, proporcionando un entorno que introduce mejoras en los procesos ingenieriles. De esta manera "automatiza" la producción de la mayoría de los activos software (pruebas, documentación, código, etc.) a partir de modelos y permite una mejor capitalización del conocimiento.

Los aportes que entrega este proyecto son los siguientes:
1. Desarrollar una solución basada en técnicas MDD, capaz de entregar un 15 a 20% de aumento de la productividad en el desarrollo de sistemas software en comparación con dos años atrás.
2. Liderar la industrialización de la solución.
3. Asegurar la adopción exitosa de la solución por la industria.
4. El proyecto Eclipse MDDi.

La estructura del trabajo comprende una serie de actividades divididas en seis grandes partes:
1. Tecnologías de modelado.
2. Procesos y metodologías.
3. Infraestructura de herramientas.
4. Adopción exitosa de MDD.
5. MDD industrial.
a. MDD industrial – integración de negocios.
b. MDD industrial – sistemas distribuidos a gran escala.
6. Gestión.

2.5. Aportes de los trabajos descritos al presente trabajo

La comunidad de la ingeniería del software presta especial interés a los nuevos enfoques de desarrollo de software que surgen, tales como la MDA o SPL. Con el fin de facilitar la entrada de estos dos enfoques, de aprovechar las ventajas que ambos ofrecen y de conseguir la aceptación y adopción de los mismos, varios proyectos que proponen diferentes procesos, métodos, técnicas, arquitecturas y tecnologías han sido desarrollados. Ellos han sido descritos en la sección anterior y de éstos se tomarán algunos aportes para el presente trabajo.

El trabajo "MDA vs. Factorías de software" [25] propone que la elección de un enfoque u otro sea tomada en función de las características del método y plantea dos escenarios diferentes para los cuales resulta apropiado el empleo de un enfoque, mientras que el otro no y viceversa. El objetivo general del presente trabajo, es definir la arquitectura que debe utilizar un entorno de desarrollo para soportar el trabajo con familias de sistemas bajo el enfoque propuesto por MDA; de acuerdo a este objetivo, es importante trabajar en los dos escenarios planteados en

[25], razón por la cual ninguno de los dos enfoques es descartado, por el contrario, es propuesta una solución mixta que hace uso de lo mejor de cada uno de éstos. Cabe notar que esta situación es similar para numerosos métodos y proyectos, por lo que todo lo que se diga aplicado al presente trabajo, será fácilmente generalizable.
De [25] se retoma la intención de desarrollar una línea de productos y la idea de capturar los requisitos específicos de cada uno de los sistemas, siguiendo las técnicas que propone el OMG con la iniciativa MDA.

El trabajo "derivación de modelos en UML para líneas de productos" [15] desarrolla una herramienta software que pretende facilitar la gestión de la variabilidad en SPL, permitiendo realizar derivaciones semiautomáticas de productos particulares, a partir de modelos arquitectónicos de familias de productos, basados en el lenguaje UML. El presente trabajo, siguiendo los objetivos específicos del proyecto, retoma la idea de obtener productos planteando reglas lógicas y de selección que implementen el concepto de derivación (el mismo conjunto de reglas propuesto en ese trabajo será empleado), y también la idea de basarse en el lenguaje de modelado UML; pero a diferencia de [15], los conceptos tomados de la arquitectura MDA son empleados en la transformación de modelos (ver capítulo 4) PIM (Plataform Independent Model) a modelos PIM más detallados, implementando una aplicación creada en la plataforma de desarrollo eclipse con herramientas que ofrecen compatibilidad entre el lenguaje UML versión 2.2. (y anteriores), XMI 2.1. y Java.

CAPÍTULO 3

LÍNEAS DE PRODUCTOS SOFTWARE

3.1. Definición

Una familia de productos software o línea de productos software[16] o SPL (Software Product Line) es un conjunto de productos construido a partir de unos activos[17] comunes; dicho conjunto está dirigido a un segmento particular del mercado, cumple con tareas específicas y resuelve las necesidades de un dominio en específico [13], [14].

3.2. Introducción

Las SPL están emergiendo rápidamente como un paradigma de desarrollo de software viable e importante; permiten a las compañías realizar grandes mejoras en tiempo, costos, productividad y calidad, además, facilitan la entrada rápida y flexible a un mercado, así como la personalización masiva de productos. A la vez, permiten la construcción de software a medida, desarrollado para satisfacer las necesidades de usuarios particulares o de grupos de ellos [13].

A diferencia de los enfoques tradicionales de reutilización enfocados sólo en el código, las SPL abarcan todos los activos relevantes durante el ciclo de vida del desarrollo de software. Ellos cubren las diferentes actividades que comprende el proceso de desarrollo de los productos software, desde la etapa de requerimientos hasta la validación, lo cual define un amplio rango de activos. Es precisamente este conjunto el que define la infraestructura de la SPL [33].

En los enfoques tradicionales de reutilización, no se dispone de contextos lo suficientemente amplios como para detectar con precisión qué elementos son reutilizables y cuáles son las situaciones donde se pueden aprovechar, esto desemboca en una reutilización oportunista del software. Para que sea sistemática, los procesos de desarrollo deberían abordar la construcción colectiva de SPL relacionadas por un dominio [2]. El enfoque de reutilización de las SPL es una alternativa para el desarrollo de produ/ctos software de una forma rápida, eficiente y productiva, ya que las aplicaciones no inician desde cero. Entre sus ventajas se puede destacar el aumento de la confiabilidad hacia el usuario final, gracias a la experiencia adquirida a través de su implantación en proyectos anteriores; la reducción de los riesgos en el proceso de desarrollo; la reducción del margen de error en los procesos de estimación, gracias a que la incertidumbre sobre el costo de desarrollo se limita solo a lo nuevo; y por último, cabe agregar que libera a los desarrolladores del trabajo repetitivo permitiendo que dediquen su tiempo a tareas más productivas y novedosas [6].

La principal diferencia entre el desarrollo tradicional de sistemas únicos y la ingeniería de SPL es un cambio fundamental de enfoque, ya no es la construcción de productos aislados. Esto implica un cambio en la estrategia: la transformación a una visión estratégica del mundo de los negocios [33], donde el éxito de la SPL es debido, en gran medida, a que las características en común compartidas por los productos software pueden ser aprovechadas para lograr

[16] Los términos familia de productos software y línea de productos software o SPL suelen emplearse indistintamente. En este documento se continuará usando las siglas SPL.

[17] En este documento se usará el término activo para hacer referencia al software asset, es decir, a un elemento software reutilizable que puede ser cualquier producto software obtenido durante el ciclo de vida del desarrollo de software. Ejemplos de activos son los requerimientos, propuestas, especificaciones, diseños, manuales de usuario, códigos, documentación, pruebas, etc.

economías de alcance[18], ya que los productos individuales son construidos a partir de un conjunto común de activos preestablecidos, de tal forma que construir un nuevo producto viene siendo una tarea más de configuración o generación, antes que de creación. La actividad predominante es la integración y no la programación, debido a que para cada línea de productos software existe una guía predefinida o un plan que específica cómo va a ser la construcción de los productos [13].

Desafortunadamente las SPL requieren por lo general de una inversión inicial extra en la organización que pretenda implementarlas. La inversión es requerida entre otras cosas, para la construcción de activos reutilizables y para la transformación de la organización. Existen diferentes estrategias para hacer la inversión, pero en general todas consideran que el punto a partir del cual se obtienen ganancias, es alcanzado después de tres productos, algunas veces antes [34] (Ver figura 4).

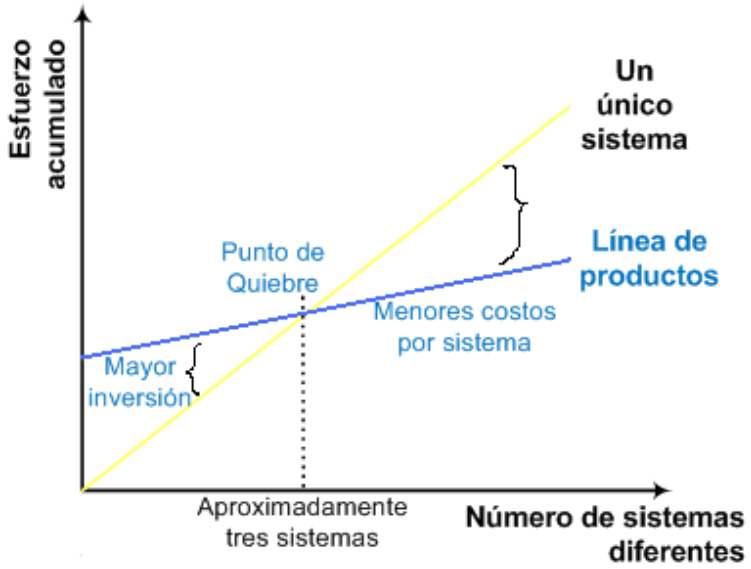

Figura 4. Equilibrio económico de la ingeniería de SPL.

Usualmente, junto con la reducción en los costos de desarrollo también es posible una rebaja en su mantenimiento. Muchos aspectos contribuyen, el hecho más notable es que el código y la documentación bajo mantenimiento disminuyen radicalmente, y a la vez, el riesgo que acompaña a todos los proyectos [33]. Las SPL también tienen un gran impacto en la calidad del software resultante, ya que una nueva aplicación consiste en gran medida de componentes probados y maduros. Esto implica que la densidad de errores esperada en nuevos productos puede ser mucho más baja que en aquellos que son desarrollados desde cero. Lo anterior permite que los sistemas sean más confiables y seguros. Además, las SPL impactan positivamente en aspectos como la usabilidad del producto final, al mejorar la coherencia de la interfaz de usuario. Esto es logrado usando los mismos bloques de construcción para la implementación de la misma clase de interacción, por ejemplo, teniendo un único componente de registro de usuario para todo un conjunto de productos, en vez de tener uno específico para cada producto. En algunos casos la demanda de este tipo de unificación ha sido el principio para la introducción del enfoque de SPL en una organización [35].

[18] La economía de alcance es el ahorro de recursos y costos que obtienen ciertas empresas como consecuencia de producir dos o más bienes o servicios de forma conjunta; se da en la fabricación colectiva de productos similares y se consiguen al compartir activos (assets) tangibles e intangibles.

Según lo dicho, es posible observar que mientras típicamente los argumentos para implementar el enfoque de SPL hacen referencia a los costos, para otras organizaciones el principal argumento para su implementación es el aumento de calidad y usabilidad de los productos [33].

3.3. Fundamentos económicos

Con el fin de explicar la relación entre las SPL y sus fundamentos económicos, este punto trata sobre las características de los mercados que son relevantes para las SPL.

3.3.1. Estrategia de definición de productos

Por medio de la estrategia de definición de productos, puede ser establecida una base para la posterior definición de un portafolio de productos. Existen dos clases principales de estrategias, una orientada al cliente y otra orientada al fabricante.
En la primera, los productos son definidos por demandas de usuarios existentes y futuros. Los productos finales son individualizados para sus necesidades específicas. Cuando existe un número muy grande de ellas se le conoce como personalización masiva, bajo esta situación resulta difícil identificar por anticipado los requerimientos específicos de cada uno de los productos y por lo tanto es muy importante que la plataforma de la SPL proporcione una base flexible para favorecer el desarrollo de los productos. En la segunda estrategia, los productos son definidos por la organización. Usualmente este caso ocurre cuando los sistemas son desarrollados para grandes mercados; en él, cada variante de un producto es vendida a cientos o miles de clientes; ella puede subdividirse a la vez, en estrategias guiadas por el mercado o la tecnología. En la primera, los productos utilizados para formar el portafolio son determinados basándose en análisis de segmentos potenciales de mercado. La segunda toma como punto de partida las capacidades tecnológicas de la organización, bien sea las que tiene, puede desarrollar o adquirir [33].
En la práctica, las estrategias de definición de productos son usualmente una mezcla de los tipos explicados anteriormente. La ingeniería de SPL puede soportar todo estos enfoques, pero sus beneficios relativos varían en relación a la forma seleccionada.

3.3.2 Estrategias de Mercado

Determinan como una organización quiere ser conocida en el mercado. Porter [36] las clasifica en:

- Bajos costos: busca ofrecer sus productos al menor precio posible.
- Diferenciación: pretende dar características específicas diferentes a sus productos con respectos a las de sus competidores, por ejemplo, servicio y marca, entre otras.
- Enfoque: la organización se enfoca en un nicho del mercado en específico.

La ingeniería de SPL soporta todas estas estrategias, incluso la combinación de éstas.

3.3.3. El ciclo de vida de la línea de productos

En el mercado los productos tienen un ciclo de vida típicamente caracterizado por las siguientes etapas [37]:

- Introducción: un nuevo producto es lanzado el mercado. Las ventas inicialmente son bajas porque el producto es desconocido.
- Crecimiento: el producto y sus cualidades competitivas son cada vez más conocidas en el mercado y como consecuencia las ventas se incrementan y las utilidades son positivas.

- Maduración: las ventas disminuyen. Los precios deben ser reducidos para ganar en un mercado competitivo.
- Saturación: es alcanzado el nivel máximo de ventas. En esta etapa es común la competencia agresiva.
- Degeneración: el producto es sustituido aceleradamente por otros productos, tal vez de la misma organización. Las ganancias disminuyen significativamente.

Este ciclo de vida de un único producto, da una visión general para uno de ellos. En el contexto de una línea de productos, es necesario tener en consideración la combinación de productos y para ello es necesario diferenciar dos situaciones:

- Productos diferentes de la misma SPL que están compitiendo en el mercado, al mismo tiempo. Los productos reducen mutuamente el mercado de los otros en un proceso conocido como canibalización de la SPL [40].
- Productos diferentes de la misma línea de productos se suplantan entre sí durante el tiempo que están en el mercado.

3.3.4. Relación entre estrategia de mercado y línea de productos

La estrategia de mercado está fuertemente relacionada con las SPL, debido a que son capaces de reducir considerablemente la cantidad de esfuerzo, costos y tiempo de comercialización requerido para que un nuevo producto sea exitoso. Son idóneas para organizaciones que elijan una estrategia enfocada en liderar el mercado con precios bajos en sus productos, ya que éstas permiten producirlos a costos más bajos (al menos si producen más de tres). Debido a que el tiempo de comercialización es reducido considerablemente con las SPL, también es posible lograr una gran diferenciación de los demás competidores en términos de rapidez. Las SPL y la estrategia de mercado deben encajar y complementarse.

3.3.5. Resultados económicos de las líneas de productos

Aunque las SPL no impactan necesariamente la funcionalidad de los productos que entregan, sí influencian otras propiedades que pueden ser clasificadas como características del producto o del proceso (figura 5).

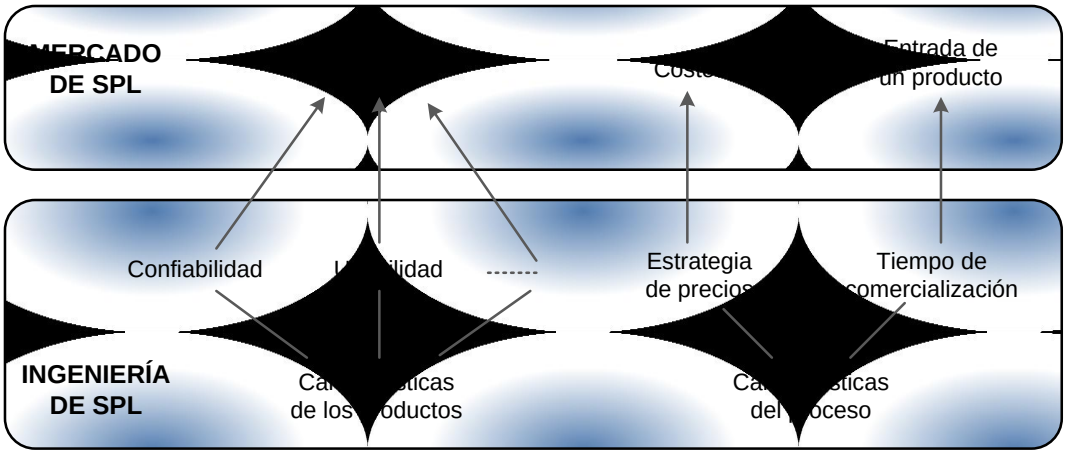

Figura 5. Cualidades en la ingeniería y mercadeo de la línea de productos.

Las características de los procesos son en gran medida afectadas por las SPL. Algunos ejemplos son [33]:

- Costos de desarrollo: como grandes partes de las funcionalidades del sistema son llevadas a término por la arquitectura, el desarrollo de nuevos productos puede ser reducido sustancialmente en tamaño y complejidad; como consecuencia, los costos de desarrollo son a menudo reducidos en una escala similar.
- Tiempo de desarrollo: como consecuencia de las reducciones en los esfuerzos de desarrollo, el tiempo de desarrollo para nuevos productos disminuye de la misma manera.

Las características del producto pueden ser evaluadas por medio de ellos mismos o mediante la observación del desarrollo y mantenimiento del sistema. Algunos ejemplos son [33]:

- Confiabilidad del producto: las SPL no son por sí mismas una estrategia de mejora de la confiabilidad, pero al reutilizar componentes de productos de calidad, ella ha sido probada hasta cierto punto, situación ventajosa frente al desarrollo de sistemas únicos.
- Usabilidad: las SPL pueden inducir a que se incremente la coherencia entre las interfaces de diferentes usuarios, siempre y cuando para implementarlas, los mismos componentes de la línea de productos sean reutilizados.
- Posibilidad de migración: es considerada simplemente como una forma de variabilidad en la arquitectura. Siempre y cuando se haya planificado, es un ejercicio simple.
- Mantenimiento: las SPL soportan el mantenimiento básicamente de dos formas, cuando el cambio ha sido considerado como parte de la arquitectura o cuando deben hacerse cambios explícitos a su infraestructura. En el primer caso el mantenimiento es simple. En el segundo caso puede resultar más difícil y costoso cambiar la arquitectura de la SPL que la de un único producto, sin embargo, los beneficios del cambio aplican para toda ella.

3.3.6. Gestión y delimitación del producto

Para el éxito de la línea de productos es importante la integración de la técnica con la planeación orientada al mercado, esto es conocido como delimitación de la línea de productos y puede ocurrir en tres niveles:

1. Delimitación del portafolio del producto: determina el rango de productos que será soportado.
2. Delimitación del dominio: identifica las principales áreas funcionales (dominios) que son relevantes a la línea de productos.
3. Delimitación de los activos: define las funcionalidades precisas que los componentes reutilizables deberían soportar.

Los tres tipos de delimitación pueden ser vistos como tres niveles diferentes, construidos uno encima del otro. De esta manera, la delimitación del portafolio de productos es una base para la delimitación de dominios, la cual a su vez es una base para la delimitación de los activos o de la infraestructura de la línea de productos [33].

3.3.6.1. Gestión del portafolio de productos

El portafolio de productos puede ser fácilmente caracterizado por una simple lista de productos junto con sus características y funcionalidades principales, esta lista es a veces llamada mapa de productos [41]. La decisión sobre cuáles productos incluir y cuáles no, puede ser reevaluada más adelante, sin embargo, en esta etapa debe ser realizada una primera evaluación.

El modelo Kano (figura 6) es un enfoque usado ampliamente para definir nuevos productos. En este modelo los requerimientos son subdivididos en las siguientes categorías [33]:

- Más que satisfechos: estos requerimientos van más allá de las expectativas de los clientes estándar y de esta forma también más allá de la competencia. Un producto exitoso debería tener algunos requerimientos más que satisfechos[19]
- Satisfechos: categoría en la que la satisfacción del cliente es aproximadamente proporcional al grado de satisfacción de los requerimientos.
- Requerimientos básicos: esta categoría corresponde a expectativas fundamentales de los clientes. Un producto exitoso debe cumplir a cabalidad con estos requerimientos.

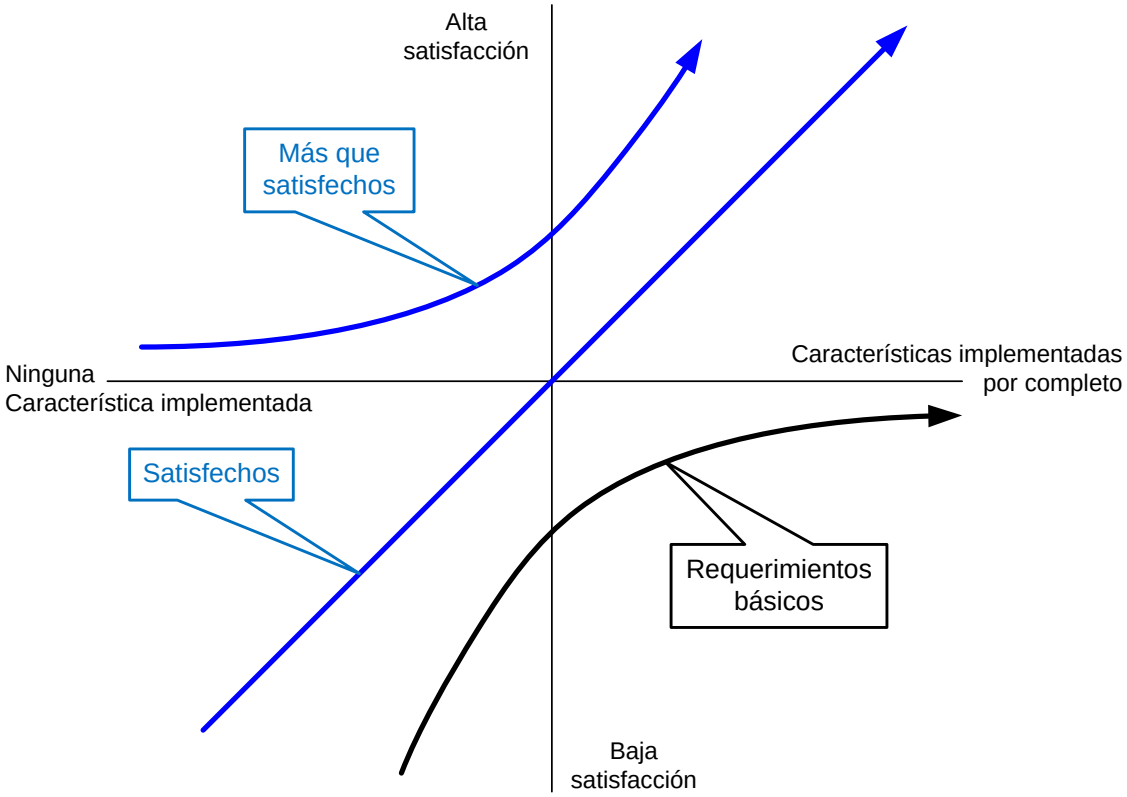

Figura 6. Requerimientos en el modelo Kano.

Para la completa definición del portafolio, es necesario analizar la interrelación entre los productos, ya que éstos pueden competir unos contra otros (canibalización de la SPL) o pueden soportar los unos a los otros (un producto diseñado a un nivel bajo, para conseguir gente que compre algo de la SPL, combinado con productos diseñados a niveles mucho más altos, para hacer que los clientes migren a lo largo de la SPL [33]. Sólo cuando los diferentes productos están armonizados, puede decirse que la definición del portafolio ha sido la apropiada.

3.3.6.2. Análisis de dominios potenciales

Todo los conceptos modernos de análisis de dominios potenciales están basados en una valoración [42], [43]. En el enfoque PulSE-ECO [44] las dimensiones de esta valoración son

[19] Los requerimientos más que satisfechos también son conocidos con el nombre de deligthers, tal y como son llamados en el libro Software Product Line in Action de Frank J. van der Linden y Klaus Schmid Eelco Rommes.

divididas en dos grandes categorías: viabilidad y comportamiento. A continuación es posible observar la lista de las dimensiones usadas en este enfoque [33].

La dimensión de la viabilidad está enfocada en determinar si la SPL puede ser establecida exitosamente en el dominio:

- Madurez: el dominio debe ser lo suficientemente maduro para que los conceptos necesarios estén establecidos y puedan ser codificados de una forma apropiada.
- Estabilidad: el dominio debe ser lo suficientemente estable para que la infraestructura de la línea de productos pueda ser estable también.
- Restricciones de recursos: los recursos necesarios para establecer la línea de productos, deben estar disponibles.
- Restricciones organizacionales: la organización debe ser la adecuada para una línea de productos, o al menos debe ser lo suficientemente flexible para ser adaptada.

La dimensión del comportamiento describe cuan exitosa se puede esperar que sea la SPL:

- Mercados potenciales: para que la línea de productos sea exitosa, es necesario que existan mercados potenciales. Idealmente, esto ya está claro en la fase de delimitación del portafolio de productos.
- Similitudes/variabilidades: constituyen la base de la línea de productos.
- Cohesión/acoplamiento: impactan la adecuación de componentes reutilizables. Si el acoplamiento con otros dominios es fuerte, es muy difícil desarrollar activos reutilizables.
- Activos existentes/ herencia: los activos existentes son una ventaja al servir como ejemplo para el desarrollo de nuevos componentes. Los productos existentes que preceden a la SPL, son una desventaja si requieren de un gran esfuerzo para su mantenimiento.

3.3.6.3. Delimitación de los activos

La delimitación de los activos proporciona la agregación inicial de funcionalidades, éstas son definidas basándose en una perspectiva de retorno de inversión y desde el punto de vista económico, pueden ser un punto de partida para la arquitectura del software.
En el enfoque PulSE-Eco la delimitación de los activos es realizada por el llamado componente de reutilización de infraestructura de alcance. Este componente comprende los siguientes cuatro pasos:

1. Formalizar la meta de reutilización como función económica.
2. Identificar detalladamente la funcionalidad relevante.
3. Caracterizar la funcionalidad en términos de funciones económicas.
4. Obtener posibles activos como resultado de la evaluación.

Por lo general solo se ponen en práctica los dos primeros pasos, en estos casos, no se consiguen los beneficios de un análisis cuantitativo.

3.4. Proceso

La ingeniería de SPL comprende dos ciclos de vida, la ingeniería de dominios y la ingeniería de aplicaciones. En la primera, la plataforma común, un conjunto de requerimientos, una arquitectura de referencia y un conjunto de componentes reutilizables son desarrollados y mantenidos; ellos forman la mayor parte de esta plataforma, es la base de trabajo para la ingeniería de aplicaciones, está última es la encargada de desarrollar los productos de la línea.
La disciplina en el proceso de desarrollo, es muy importante en el enfoque de SPL, ya que se requiere de más coordinación y disciplina entre los participantes en el proceso, que cuando es desarrollado un sistema único. Las dependencias dentro de la organización son mayores y por

lo tanto la calidad, la gestión y la ejecución de los procesos deben realizarse en forma organizada.

3.4.1. Modelo general de la ingeniería de línea de productos

La figura 7 muestra un modelo general para la ingeniería de SPL y la divide en dos ciclos de vida. En ellos hay nueve sub-procesos, ocho forman cuatro parejas fuertemente conectadas para ambos dominios de la ingeniería y son: requerimientos, diseño, construcción y evaluación.

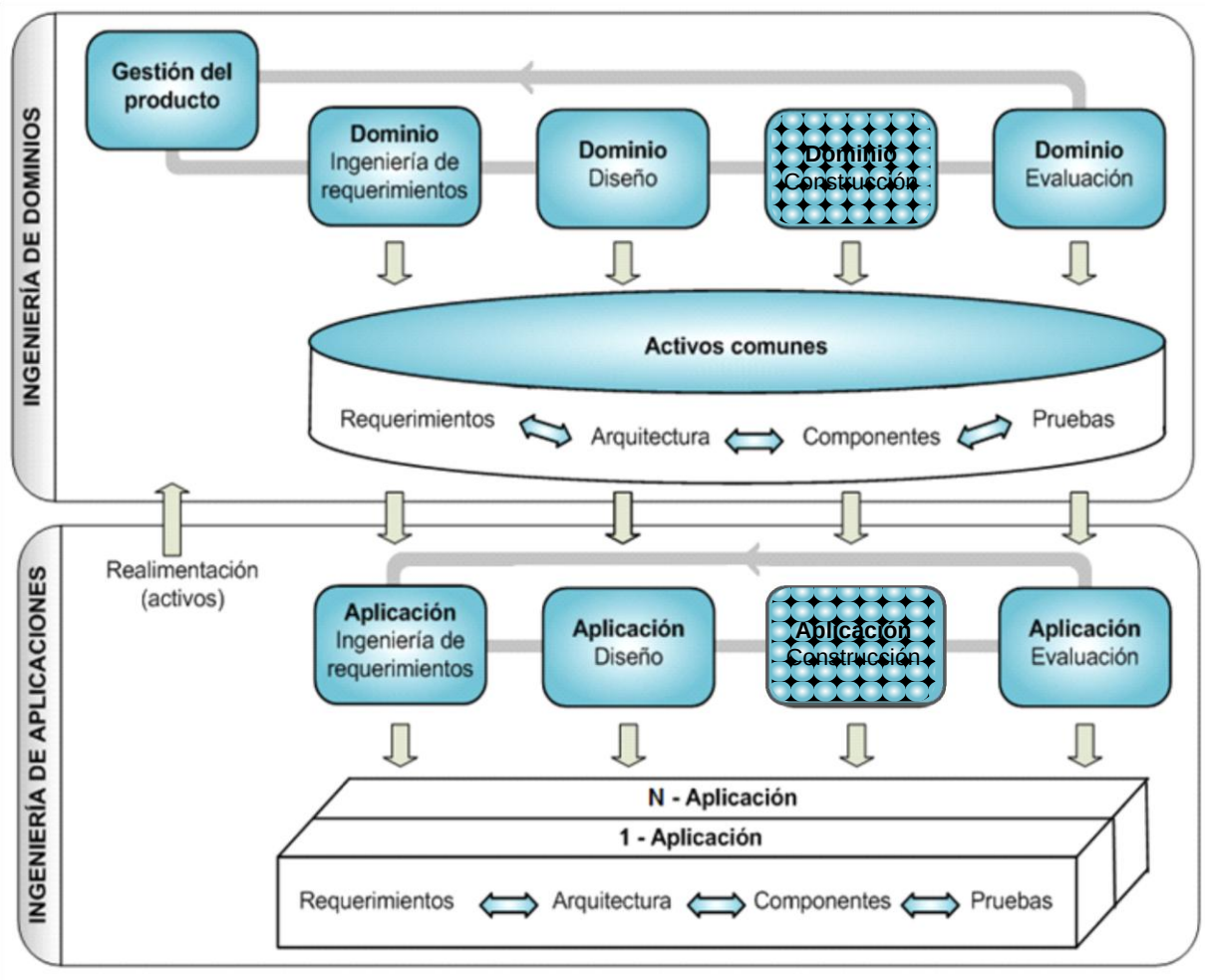

Figura 7. Framework – Ingeniería de línea de productos.

Los subprocesos en la ingeniería de dominios entregan activos comunes que son usados por su equivalente en la ingeniería de aplicaciones para la creación de los productos. A cambio, los subprocesos en la ingeniería de aplicaciones generan realimentación utilizada en la ingeniería de dominios para mejorar los activos comunes. Este ciclo es esencial para asegurar que la plataforma sea adecuada para la producción eficiente de los productos finales.

3.4.1.1. Ingeniería de dominios

1. Ingeniería de requerimientos de dominio [37]

Es el proceso de creación y gestión de requerimientos para la arquitectura de referencia y su implementación. Los requerimientos de referencia resultantes deben cubrir los requisitos de todas las aplicaciones (previsibles) dentro del alcance de la línea de productos, esto incluye los comunes y los puntos de variación que la arquitectura de la línea de productos debe soportar.
A grandes rasgos es posible decir decir que la ingeniería de requerimientos consta de cinco etapas básicas, de acuerdo con AMIR-ST (Aproximación Metodológica para la Ingeniería de Requisitos de Sistemas Telemáticos)[20] [38]:

- Preparación y gestión: elaborar un plan de trabajo para la captura de requisitos, gestionar los riesgos y la documentación asociados a él y controlar el desarrollo y ejecución de las actividades relacionadas con la especificación de requisitos de todo proyecto.
- Recolección y obtención: estudiar el dominio del problema para obtener y registrar información sobre las necesidades de los clientes y usuarios.
- Análisis y negociación: profundizar en el conocimiento de la organización para asegurar un acuerdo con el cliente sobre los procesos dentro de la organización que el sistema telemático a construir va a dar soporte.
- Especificación: realizar una descripción formal de los requisitos que finalmente el sistema a construir va a cumplir.
- Validación y verificación: comprobar los documentos y modelos para detectar omisiones, conflictos y ambigüedades, así como comprobar que los requisitos siguen normas de calidad establecidas y actualizar el repositorio de requisitos del grupo desarrollador.

2. Diseño del dominio
Toma los requerimientos de referencia como entrada y crea una arquitectura de referencia para la plataforma. Ella sirve como base para el diseño de productos durante el diseño de la aplicación.

3. Construcción del dominio
Los activos comunes son diseñados y creados a partir de implementaciones existentes y aplicando el concepto de reutilización, o partiendo desde cero. Para cada activo es necesario decidir si es construido/comprado/reutilizado/encargado, aunque frecuentemente esta decisión es inconsciente [13]:

- Construir: el activo es construido en la organización. Ella tiene control total sobre la especificación e implementación del activo y está limitada solamente por sus propias capacidades.
- Comprar: el activo es comprado como un producto fuera de la arquitectura. Los activos que fácilmente están disponibles en el mercado suelen ser económicos cuando son comprados a terceros.
- Reutilizar: el activo es reutilizado de un sistema existente dentro de la organización. El límite es impuesto por el rango de activos que estén disponibles y su adaptabilidad a la arquitectura. En algunos casos un activo de una aplicación específica puede ser tomado y convertido en un activo común. Si el proceso de ingeniería de aplicaciones es responsable de que los activos permanezcan en funcionamiento, esto puede ser relativamente sencillo.
- Encargar: las especificaciones de los activos son creadas en la organización, pero la construcción de éste se asigna a terceros. Esto puede llevar a que la implementación no esté en concordancia con la intención original del activo. El riesgo debe ser manejado apropiadamente, por ejemplo, concentrando esfuerzos en la creación de especificaciones

[20] AMIR-ST [38] es una aproximación metodológica propuesta por el Grupo de Ingeniería Telemática (GIT) de la Universidad del Cauca que pretende contribuir a que los equipos desarrolladores de sistemas telemáticos logren al menos un nivel dos en Ingeniería de Requisitos [39].

de muy alta calidad o estableciendo mecanismos extras de comunicación y ciclos de realimentación cortos.

4. Evaluación del dominio
La variabilidad dificulta realizar una evaluación minuciosa de la arquitectura. Incluso un número pequeño de puntos de variación, rápidamente llevarán a un número inmenso de configuraciones posibles, haciendo muy difícil evaluar todos los productos potenciales en una línea. La arquitectura no puede ser evaluada para cada producto, porque no es posible predecir con precisión, cuáles serán los nuevos productos. Tan solo un número limitado de configuraciones pueden ser evaluadas [33].

3.4.1.2. Ingeniería de aplicaciones

1. Ingeniería de requerimientos de aplicaciones
Los requerimientos provienen de los interesados (clientes y usuarios) en el producto bajo desarrollo; por ejemplo: los usuarios finales, gerentes de productos o ingenieros de servicios. El modelo de variabilidad y los requerimientos de dominio, comunes y variables, pueden ser una valiosa herramienta para conseguir los requerimientos de los participantes [37].
La ingeniería de aplicaciones debe analizar que está disponible y que es necesitado. Si existe un requerimiento no satisfecho debe decidir si continuar y darle solución con una aplicación específica, o abandonarlo por completo, o posponerlo para una versión futura de la aplicación. El último caso suele ocurrir cuando éste es compartido por muchas aplicaciones; la realimentación dada por el ciclo de vida de la ingeniería de aplicaciones, puede proporcionar valiosa información a los ingenieros de dominio para que dicho requerimiento sea cumplido en una próxima versión de la arquitectura.

2. Diseño de la aplicación
Una arquitectura de producto es derivada a partir de la arquitectura de referencia. Para su obtención los puntos de variación son solucionados. Ella debe trabajar con todos los requerimientos del producto seleccionado. La arquitectura resultante es extendida con los nuevos componentes e interfaces, o componentes particulares son reemplazados, en función de satisfacer todos los requerimientos de las aplicaciones específicas que no son cubiertos por la arquitectura de referencia [37].

3. Construcción de la aplicación
La meta del proceso de construcción de la aplicación, es implementar productos. Los activos comunes son entregados por el proceso de construcción del dominio y son usados para minimizar el esfuerzo y tiempo necesario para el desarrollo.
Las interfaces de dominio pueden ser reutilizadas sin cambios, pero los componentes variables, aquellos con puntos de variación internos, deben ser adicionados. Las dos opciones para hacer esto más eficiente son la configuración de componentes especializada o las herramientas de soporte.
Los activos comunes son complementados por componentes de aplicaciones específicas, en muchos casos, tales componentes proporcionarán interfaces especificadas por la arquitectura de dominio. La creación de componentes de aplicaciones específicas es llevada a cabo, casi siempre, de la misma forma como se hace el desarrollo de un único sistema [37].

4. Evaluación de la aplicación
Aunque los componentes comunes fueron probados en la ingeniería de dominios no significa que no deban ser probados nuevamente, debido a ellos tienen variabilidades internas que fueron solucionadas para hacerlos útiles a la aplicación. Como fue mencionado anteriormente,

es imposible evaluar todas las combinaciones de configuración, por lo tanto, los componentes de dominio configurados deben ser probados nuevamente en el contexto de la aplicación específica para filtrar otros errores. Para los requerimientos de aplicaciones que son idénticos a los de dominio, las pruebas de dominio aplicadas en este contexto pueden ser repetidas. Otras pruebas llevarán puntos de variación que deben ser solucionados para ajustar la aplicación. Para los requerimientos de aplicaciones específicas es necesairo desarrollar y ejecutar nuevas pruebas [37].

3.5. Arquitectura

La arquitectura de la SPL delimita, en líneas generales, el alcance de la funcionalidad que un conjunto de sistemas puede manejar, captura las decisiones de diseño de más alto nivel, incluyendo la organización en componentes y su interacción, así como principios y lineamientos para su implementación y evolución. Es conocida como arquitectura de referencia, contribuye con la necesidad de no destacar un único producto, sino muchos y diferentes, para que compartan eficientemente grandes partes de su implementación.

La arquitectura de referencia es especialmente importante para la línea de productos porque facilita que los productos compartan activos. Ellos son creados pensando en dicha arquitectura, esto permite a los ingenieros de dominio hacer suposiciones sobre su contexto, en el cual serán usados los activos desarrollados y, como resultado, la creación de activos es convertido en un proceso más sencillo y económico, además, también facilita el trabajo de los ingenieros de aplicaciones siempre y cuando sean guiados por ella.

3.5.1. Consideraciones sobre la Arquitectura de una SPL

La arquitectura de una SPL tiene cuatro puntos básicos [45]:

1. Requerimientos arquitectónicos importantes: son aquellos que tienen gran impacto en la arquitectura. En teoría, la ingeniería de requerimientos es un prerrequisito para diseñar, de tal forma que los requerimientos para la arquitectura sean establecidos desde el principio, sin embargo, en la práctica, a menudo los arquitectos no pueden permitirse esperar hasta que las especificaciones de ellos estén estables y disponibles. En su lugar, tienen que trabajar con requerimientos cambiantes e incompletos. Es posible distinguir tres tipos de requerimientos de acuerdo a AMIR-ST [38]: de información, funcionales y no funcionales, en el capítulo 5 es realizada y desarrollada una breve descripción de éstos para el caso de estudio del presente trabajo. Para las SPL, la variabilidad es una de las directrices más importantes en la arquitectura, debe ser soportada tanto estructural como conceptualmente.

2. Arquitectura conceptual: determina los conceptos importantes a abstraer para detalles de la implementación, describe los principales conceptos que controlan el trabajo de los sistemas, sin entrar en los detalles de su implementación. Sirve como un modelo conceptual[21] o mental que permite entender y simplificar el problema [46], además proporciona un medio eficaz para la comunicación entre los participantes en su desarrollo.

3. Estructura: captura la descomposición del sistema en componentes y sus relaciones. Los componentes pueden ser genéricos o aplicaciones específicas, pueden ser diseñados y

[21] Un modelo conceptual es una descripción de conceptos en el dominio de interés. Muestra los conceptos más importantes y sus relaciones.

desarrollados dentro de la organización, desarrollados por terceras partes de acuerdo a un diseño dado, o comprados como productos comerciales fuera de la arquitectura.

4. Textura: es una colección de reglas para la implementación de la arquitectura y su evolución en el tiempo. Estas reglas pueden ser expresadas como convenciones de codificación, patrones de diseño y estilos arquitectónicos. La textura de una arquitectura guía a diseñadores y arquitectos. Sus lineamientos y reglas les dicen cómo trabajar en detalle fuera de la arquitectura, y como implementarla. Es usada durante todo el diseño del sistema, garantizando un enfoque consistente para interpretar la arquitectura y resolver problemas recurrentes. Guía a los arquitectos en la evolución que sufre la arquitectura con el paso del tiempo sin destruir sus principales conceptos y captura las ideas básicas o su filosofía principal y mantiene una totalidad coherente. La textura es llamada, a veces, meta-arquitectura [47].

3.5.2. Diseño de una línea de productos, gestión de la variabilidad

El principal cambio en el diseño de arquitecturas de referencia para SPL, está basado en el hecho de que el arquitecto tiene que tratar con muchos productos diferentes al mismo tiempo. En algunos casos, especialmente en compañías grandes, cada producto tiene muchas personas involucradas. A diferencia de los diseños enfocados en un único sistema y en los cuales no es necesario establecer buenos canales de comunicación entre las personas involucradas en el desarrollo, con las SPL, la comunicación es un asunto de gran importancia.
Obtener los requerimientos correctos para todos los productos es difícil, especialmente si existieran conflictos entre ellos. Deben ser resueltos en la arquitectura. Cancelar productos para que no existan conflictos es una salida fácil y conduce a desarrollar una línea de productos pobre. Para que esto no suceda, el soporte a la variabilidad debe ser una parte integral de la arquitectura de referencia [48].

3.5.2.1. Tipos de variabilidad

1. Similitud: es una característica (funcional o no funcional) común a todos los productos en la SPL, la cual es implementada como parte de la plataforma.
2. Variabilidad: es una característica que puede ser común a algunos productos, pero no a todos. Debe ser modelada explícitamente como una variación posible implementada de tal manera que permita aplicarse solamente a ciertos productos.
3. Producto específico: una característica puede ser parte de solamente un producto, al menos para el futuro visible. Si estas variaciones no están integradas dentro de la plataforma, debe existir soporte en ella [13].

Una variación específica puede cambiar de tipo durante el ciclo de vida de la línea de productos. Mientras que las similitudes y variabilidades son manejadas por lo general en la ingeniería de dominios, las partes de productos específicos son manejadas exclusivamente en la ingeniería de aplicaciones.

3.5.2.2. Representación de la variabilidad

Esta información puede categorizar los componentes de las arquitecturas de diversas formas. A grandes rasgos los componentes de un diagrama pueden ser [15]:

- Componentes obligatorios: aquellos que deben aparecer en todos los productos instanciados a partir de un diseño arquitectónico de la SPL.

- Componentes opcionales: aquellos que pueden aparecer o no, sin que por ello el producto instanciado se considere "incompleto". Son por tanto, como su propio nombre indica, aquellos que aportan una funcionalidad añadida, opcional a los ejemplares de productos.
- Componentes intercambiables: son aquellos componentes obligatorios dentro de los ejemplares de productos, pero para los cuales la arquitectura de la SPL provee un número de diseños o implementaciones diferentes entre los que el diseñador puede elegir. En este sentido, los componentes intercambiables son agrupados según su funcionalidad: todos los de un mismo grupo tienen un mismo objetivo y realizan una misma función en el producto final, pero pueden hacerlo de maneras muy diversas. Como regla básica, dentro de los componentes que conforman un grupo de selección, sólo uno puede aparecer en el diseño del producto final.

El modelo de variabilidad por sí solo no la puede representar por completo en una SPL, debido a que la separación de los componentes no es tan clara en un producto real como puede serlo en teoría. Las características de los productos no siempre son fáciles de asignar a un único componente, ya que en ocasiones la funcionalidad de un producto depende de las relaciones entre sus características, por ejemplo, en un cliente de correo electrónico, en un principio el componente de edición de textos y el componente de envío de mensajes son distintos y no tienen porque compartir funcionalidades, sin embargo, en un análisis más detallado, puede verse como aspectos de la funcionalidad de uno de los componentes dependen de la funcionalidad que provea el otro. De nada sirve que el componente de edición permita incluir imágenes en los mensajes, si el componente de envío no es capaz de enviarlas junto con el texto [15].
Además del modelo de variabilidad son necesarios los puntos de vista tradicionales sobre requerimientos, diseño, etc. y la relación entre estos puntos de vista y la variabilidad, de tal forma que sea conocido como impactará la variabilidad sobre estos puntos de vista individuales.

3.5.2.3. Ingeniería de aplicaciones y variabilidad

Para cada nuevo requerimiento capturado durante la ingeniería de aplicaciones se debe decidir si formará parte de la plataforma, una variación, o si será delegado para el desarrollo del producto. El caso más simple es cuando un nuevo requerimiento es capturado y la infraestructura de la línea de productos ya lo soporta. En este caso solo es necesario revisar que el tiempo de vinculación[22] en la infraestructura también lo cumple. Si el requerimiento no es soportado por la infraestructura, existen tres posibilidades diferentes [33]:

1. Tratar de negociarlo, descartarlo o reemplazarlo. Aunque esto puede sonar extraño desde una perspectiva de satisfacción del cliente, estas tres opciones deben ser evaluadas ya que en el contexto de una línea de productos, entre más variabilidades se deban soportar, más se dificultará la evolución de la infraestructura a conseguir.
2. El nuevo requerimiento será integrado con la infraestructura de la línea de productos.
3. El nuevo requerimiento será integrado en una aplicación de manera específica.

La segunda posibilidad lleva a que la ingeniería de aplicaciones trabaje junto a la ingeniería de dominios sobre él. La tercera posibilidad lleva a que solamente la ingeniería de aplicaciones se ocupe de él.

[22] Tiempo de vinculación describe cuando debe ser seleccionada una variante. Algunos valores típicos son el tiempo de compilación, tiempo de enlace, tiempo de ejecución, etc.

3.5.2.4. Técnicas básicas para trabajar la variabilidad

Hay cuatro técnicas básicas para representar la variación en una arquitectura [33]: adaptación, sustitución, extensión y derivación. En la adaptación, existe una única implementación disponible para cierto componente, pero las interfaces para ajustar el comportamiento son ofrecidas. Cada una puede tomar la forma de un archivo de configuración, de parámetros en tiempo de ejecución o incluso de parches para el código fuente de los componentes, por nombrar tan solo algunas opciones.
Con la sustitución, están disponibles muchas implementaciones de un componente. Cada implementación cumple con las especificaciones de los componentes como es descrito en la arquitectura. En la ingeniería de aplicaciones una de las implementaciones es seleccionada, o la implementación de un producto específico es desarrollada de nuevo, siguiendo las especificaciones dadas.
La extensión solicita a la arquitectura el suministro de interfaces que permitan adicionar nuevos componentes que pueden o no ser productos específicos. La técnica de extensión es diferente de la sustitución por el hecho de tener disponibles solamente interfaces genéricas para adicionar componentes. Con la técnica de sustitución, la interfaz especifica exactamente qué debería hacer el componente y solo varía como es hecho el componente. Con la extensión cierto número de componentes pueden ser adicionados usando las mismas interfaces, mientras que con la sustitución un único componente es reemplazado por algún otro. La figura 8 bosqueja gráficamente estas tres técnicas.

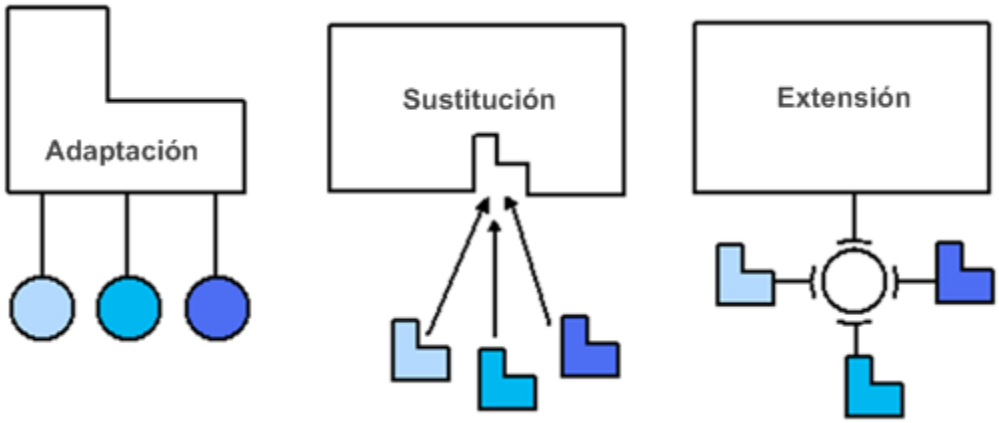

Figura 8. Tres técnicas básicas para implementar la variabilidad en una arquitectura.

Las SPL buscan obtener productos específicos a partir de su diseño; al incluir todos los componentes comunes, y seleccionar aquel conjunto de componentes específicos que permiten dar forma al producto final y que cumplen con los requisitos específicos que le caracterizan. Este concepto de obtener un producto después de escoger entre todos los posibles que permite el diseño de una SPL, recibe el nombre de derivación o instanciación de producto [15]. Las arquitecturas de software tradicionales están enfocadas al diseño de un único producto, y por lo tanto definen características específicas referentes a la estructura y el comportamiento del sistema que representan. La forma de generar un ejemplar de este modelo de diseño, consiste en implementar el código fuente que da lugar al sistema que representa. En este mismo sentido, la arquitectura de la SPL permite diferenciar entre elementos comunes y variantes, hace posible especificar las derivaciones permitidas y sus condiciones, y ofrece mecanismos de variabilidad para ejecutar dichas derivaciones. En una arquitectura de software tradicional diferentes implementaciones prácticas pueden corresponder a un mismo diseño arquitectónico [15].

3.5.2.5. Mecanismos concretos de variación

En la arquitectura de referencia estos mecanismos están desplegados en el modelo de los puntos de variación. Algunos ejemplos de tales mecanismos son los siguientes:

Mecanismos de la técnica de derivación [15]

- Compilación condicional: si el diseño arquitectónico se limita a especificar un conjunto de limitaciones basadas en condiciones básicas y delega la derivación hasta el tiempo de compilación. Es un mecanismo de bajo nivel, clásico y muy básico, más orientado a componentes que a la arquitectura, dificulta la configuración y el mantenimiento del código, penaliza la evolución y la reutilización de los productos.
- Configuración de parámetros de componentes: permite configurar parámetros en el interior de los componentes que serán usados, y en distinta forma según el producto final a desarrollar. Es un mecanismo de variación específico de aquellos casos en los que la variabilidad es muy pequeña, o en los que los mecanismos no pueden utilizarse hasta el tiempo de ejecución.
- Especialización: enfocado especialmente a los diseños orientados a objetos a través del concepto de herencia, posibilita especializar una clase de diferente manera según el producto final en el que vaya a aparecer. La superclase o clase padre contienen la funcionalidad que es común a todos los productos de la SPL, y las características particulares de cada producto individual aparecen en las subclases o clases hijas.
- Elección de componentes: permite especificar si un componente variable está o no incluido en un producto particular dentro de la SPL. De esta forma, productos con funcionalidad más reducida pueden derivarse dejando fuera bloques completos de componentes, mientras que otros de mayor funcionalidad si los incluirán. Es un mecanismo de variabilidad de tipo general y alto nivel.
- Selección de componentes: permite especificar un conjunto de componentes variables que son complementaros entre sí, es decir, los componentes de un grupo reemplazan unos a otros en los diferentes productos, y solo uno de cada conjunto puede formar parte del producto final.
- Selección lógica: permite especificar un conjunto de restricciones lógicas sobre los componentes variables de la línea. Los productos específicos son aquellas derivaciones de la línea principal que cumplen el conjunto de restricciones lógicas que los caracterizan.

Mecanismos de la técnica de adaptación [49] [50]

- Herencia: dada una clase y su implementación, una sub-clase es introducida para cambiar algunos de los comportamientos defectuosos, según necesite la aplicación.
- Parchado: si el código fuente de un componente está disponible, esta puede ser una forma efectiva para cambiar parte de su comportamiento sin asumir la carga de mantener el componente entero.
- Configuración en tiempo de compilación: los compiladores pueden ofrecer mecanismos para variar un componente en tiempo de compilación. Preprocesadores y macros, son formas de lograr la variabilidad. Archivos listos pueden compilar un componente en muchas variantes binarias o seleccionar los componentes correctos para conectar un ejecutable.
- Configuración: en este caso una implementación de un componente tiene internamente diferentes variaciones y proporciona una interfaz para seleccionar entre las diversas posibilidades. Un archivo de configuración simple puede hacer lo necesario; algunos ejemplos son los parámetros y las llamadas a procedimientos.

Mecanismos de la técnica de sustitución [49], [50]

- Generación de código: un generador de código lee algunos tipos de especificaciones de alto nivel, por ejemplo un modelo o un argumento de proceso (script), y genera el código requerido para cierto componente o incluso para todo un producto.
- Sustitución de componentes: la implementación defectuosa de un componente es reemplazada con algún otro.

Mecanismo de la técnica de extensión [50]
- Enchufar: la arquitectura ofrece interfaces que permiten que componentes de enchufado pueden ser adicionados al sistema. Ellos proporcionan ciertas funcionalidades y pueden ser comunes o específicos a una aplicación.

3.5.3. Evolución

Una arquitectura inevitablemente enfrentará requerimientos imprevistos, de hecho, su éxito bien podría depender de cómo les hace frente; si ha sido bien diseñada debería poder manejar nuevos requerimientos, mientras éstos hayan sido esperados. Es muy posible que con el paso del tiempo la arquitectura encuentre requerimientos que no puede soportar en su forma actual y por consiguiente deba cambiar. Esto es llamado evolución intencional.

Algunas de las razones por las cuales surgen nuevos requerimientos son las siguientes [33]:
- El mercado demanda nuevas características o mejoras en la calidad. Las características y propiedades pueden terminar en nuevas versiones de los productos existentes o en todo un nuevo producto.
- Por otro lado, las características y los productos pueden volverse redundantes. Por ejemplo, la mayoría de los computadores personales no están actualmente equipados con la unidad de disco de 3 ½. Cada cambio debe ser una oportunidad para simplificar la arquitectura y eliminar algo innecesario.
- Cuando una organización adquiere otra organización, todo un rango de productos existentes puede ser adicionado a una línea de productos.
- Adelantos tecnológicos. Por ejemplo, una solución interna (a nivel de la organización) a un problema de tiempo atrás, puede ahora ser reemplazada por un componente disponible comercialmente.

La evolución también ocurre sin propósito alguno, en tal caso es llamada evolución no intencional. Incluso si una arquitectura documentada permanece estable, su implementación cambia a causa del mantenimiento. Si este proceso no es guiado y monitoreado puede llevar a dos problemas: una incompatibilidad entre la arquitectura y su implementación y al deterioro de la arquitectura, también conocido como entropía del software o "software *rot*"[23]. Con el tiempo, la arquitectura y su implementación caminan en forma separada. Si no se hace algo al respecto, en algún punto, esta brecha será tan grande que leer la documentación de la arquitectura no ayudará a entender el software.
El deterioro de la arquitectura significa que el diseño del software es llevado al fracaso, sin intención, debido a una serie de pequeños cambios. La textura de la arquitectura puede desempeñar un rol importante para disminuir este tipo de problemas. Sus lineamientos aplican no solo durante la implementación inicial sino también en el mantenimiento. La refactorización[24] es otra forma importante de luchar contra el deterioro del software [51], significa que el diseño

[23] El término software rot hace referencia al deterioro que sufre el software con el tiempo y que eventualmente lo llevará caer en desuso.
[24] La refactorización es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna, sin cambiar su comportamiento externo.

es mejorado sin cambiar la funcionalidad del sistema. Originalmente fue planteada para usar en un nivel de diseño detallado pero puede ser aplicada a todos los niveles de diseño incluyendo la arquitectura.

La vida útil de una arquitectura depende del dominio del problema en que trabaja. Algunos sistemas, como equipos profesionales para médicos pueden trabajar por décadas si existe un mantenimiento continuo, mientras que otros, como los teléfonos pueden tener una nueva arquitectura, cada tres años [33]. Su evolución para soportar nuevos requerimientos puede volverse muy costosa o riesgosa y es en este momento que ha alcanzado el fin de su vida útil. Esto no significa que carezca de valor ya que pueden existir productos en el dominio para el que fue construida que necesiten mantenimiento; además, la arquitectura representa conocimiento y experiencia valiosa en el dominio del problema. Sus requerimientos, conceptos, estructura y textura pueden ser activos reutilizables o pueden ser fuente de inspiración para una nueva arquitectura de referencia.

CAPÍTULO 4.

ARQUITECTURA DIRIGIDA POR MODELOS

4.1. Introducción

El desarrollo dirigido por modelos, o MDD (Model Driven Development), es una aproximación al desarrollo de software basado en el modelado del sistema y su generación a partir de modelos, ha ganado popularidad gracias al esfuerzo de OMG (Object Management Group) y, en particular, a su propuesta de una arquitectura dirigida por modelos o MDA (Model Driven Architecture). MDD pretende que la comunidad de ingeniería del software pase de métodos de desarrollo basados en lenguajes de programación a métodos basados en modelos conceptuales [25].
MDA está enfocado en la funcionalidad y comportamiento del sistema, antes que en la plataforma o plataformas de tecnología sobre las cuales será implementado, separa los detalles de la implementación de la lógica de negocio, de esta manera no es necesario repetir el proceso de definir una aplicación o funcionalidad y comportamiento del sistema cada vez que una nueva tecnología surge. Otras arquitecturas están generalmente atadas a la tecnología en particular, con MDA, la funcionalidad y comportamiento, son modelados una vez y solamente una vez [16].

4.2. Modelado, metamodelado y transformaciones

Esta sección describe algunos términos y conceptos relacionados con el enfoque MDA, expone la arquitectura en la cual trabaja MDA y todos los estándares que promueve OMG (ver sección 4.2.1.); realiza una breve descripción de los modelos de diseño e implementación que propone MDA para abordar el proceso de desarrollo de software desde diferentes niveles de abstracción.

4.2.1. Object Management Group

Es una organización internacional abierta, sin ánimo de lucro, definida como un consorcio de la industria informática. Centra sus esfuerzos en integrar normas para una amplia gama de tecnologías y un conjunto amplio de industrias; mantiene la marca registrada sobre MDA, así como sobre varios conceptos similares que están relacionados con MDD [16], [24].
Maneja un proceso abierto, con una posición neutral, en el cual propone tecnologías e invita a realizar propuestas y realimentarlas por cualquiera de sus miembros, antes de llegar a un consenso sobre una especificación final, convirtiéndola a un estándar adoptado, “de facto” [16].

4.2.2. Desarrollo dirigido por modelos

Es una aproximación al desarrollo de software basado en el modelado del sistema software y su generación a partir de modelos. Al ser únicamente una aproximación, proporciona una estrategia general a seguir en el desarrollo de software, pero no define técnicas a utilizar, ni fases del proceso, ni ningún tipo de guía metodológica [25].
Un modelo es una simplificación de la realidad, es una descripción de un sistema, o parte de él, que si está escrito en un lenguaje formal[25] resulta muy apropiado para la interpretación

[25] Un lenguaje formal es un lenguaje con una forma (sintaxis) y propósito (semántica) bien definidos.

automática de un computador [52]. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores que no son relevantes para el nivel de abstracción dado, además, sirve como medio de comunicación entre clientes, arquitectos, diseñadores y programadores; es más barato de construir que un sistema real y sirve de soporte para la implementación [6].

MDD propone un nuevo nivel de abstracción en el cual los modelos independientes de la plataforma software son construidos. La figura 9 representa gráficamente que la independencia de la plataforma software es análoga a la independencia de la parte hardware; por ejemplo Java, permite escribir una especificación que puede ejecutarse sobre una variedad de plataformas hardware, sin necesidad de cambios; así, una especificación independiente de la plataforma software podría ser trasladada a un ambiente como CORBA (Common Object Request Broker Architecture) multiprocesador/multitarea, o a un ambiente de bases relacionales cliente/servidor, sin cambios en el modelo [52].

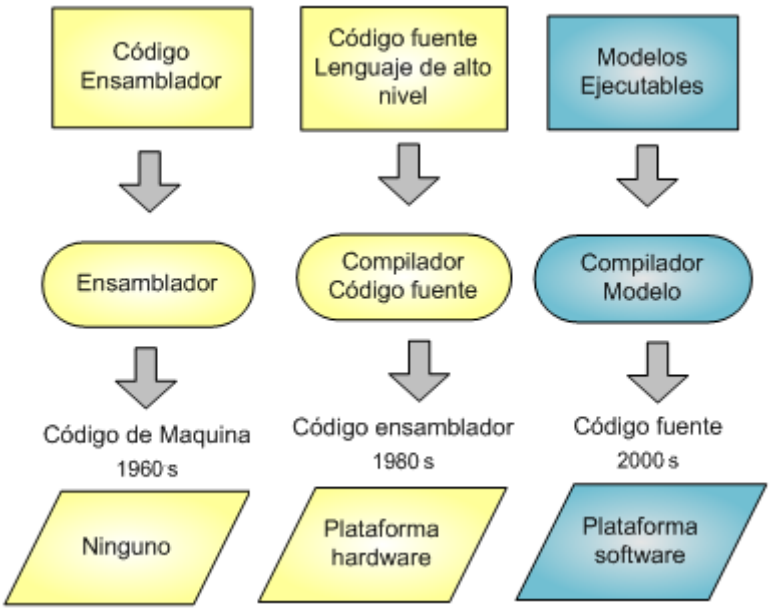

Figura 9. Aumento del nivel de abstracción

4.2.3. Arquitectura dirigida por modelos

Comprende un conjunto de estándares de OMG que promueven MDD y agrupa varios lenguajes que pueden usarse para definir métodos que sigan este enfoque, sin embargo, solo proporciona la infraestructura tecnológica y conceptual con la que construir estos métodos [25].

Es una nueva manera de escribir especificaciones basadas en un modelo independiente de la plataforma. Es un acercamiento al diseño de software, propuesto y patrocinado por OMG, concebido para dar soporte a la ingeniería dirigida por modelos de los sistemas software [15], está enfocada primordialmente en el funcionamiento y el comportamiento de las aplicaciones distribuidas o de sistemas, no en la tecnología sobre la cual va a ser implementada, separando los detalles de la implementación, de las funciones de negocio, así, no es necesario repetir el proceso de modelado de una aplicación o comportamiento o funcionalidad del sistema cada vez que una nueva tecnología aparece [53].

El ciclo de vida de desarrollo de software de MDA no es muy diferente del tradicional, ya que se identifican prácticamente las mismas fases, el factor diferenciador principal son los modelos creados durante el proceso de desarrollo y la naturaleza de éstos, ver la figura 10.

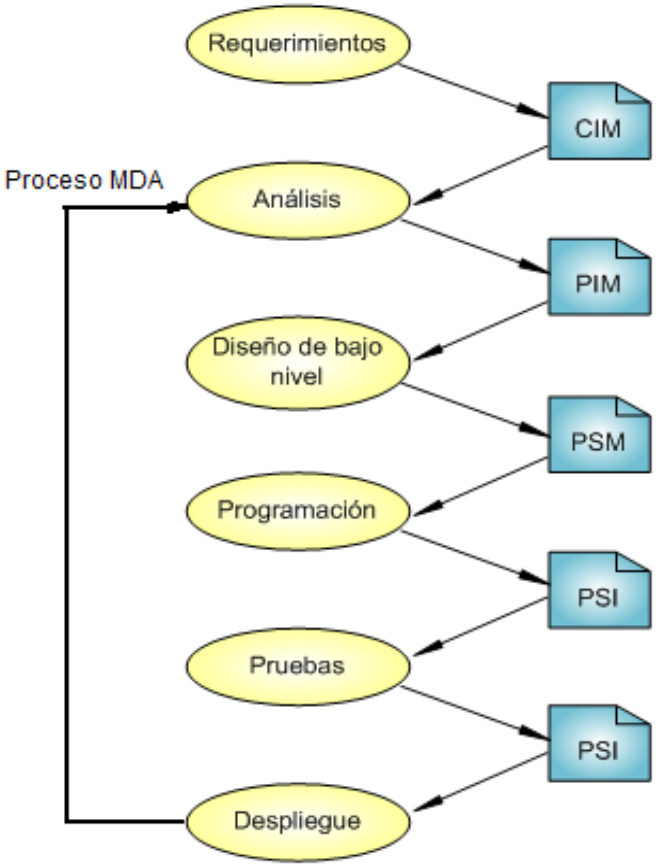

Figura 10. El ciclo de vida del desarrollo software de MDA

Aunque las siglas MDA incluyen el término arquitectura, no significa que MDA defina una arquitectura software en particular o un estilo arquitectónico, en realidad es una construcción conceptual amplia que describe un conjunto de enfoques de desarrollo de software centrados en la utilización de modelos. Ya no son creados únicamente con fines de documentación o como guía para la implementación, ahora son el núcleo de todo el proceso de desarrollo.

Los principales objetivos de MDA son: la optimización de la productividad, posibilidad de migración, operación conjunta y reutilización de activos. Los beneficios más importantes de MDA son: la reducción de los costos durante todo el ciclo de vida de la aplicación, reducción en el tiempo de desarrollo para nuevas aplicaciones, mejoras en su calidad, aumento en el retorno de inversión en tecnología e inclusión rápida de los beneficios de tecnología emergente en sistemas existentes [16].

En cuanto a la reutilización de activos, es posible utilizar cualquier tipo de activo desarrollado con anterioridad en un nuevo proyecto, conocido con el nombre de reutilización de software [54]. La reutilización de software persigue el objetivo de volver a utilizar elementos y componentes software en lugar de tener que desarrollarlos desde el principio. Es posible afirmar que es un modelo de desarrollo de software [55], que requiere un entorno soportado por tecnología que aún no está del todo madura y un modelo de proceso que defina cuándo, cómo

y dónde reutilizar, o cómo desarrollar el elemento de forma tal que no se caiga a un asunto netamente oportunista, ir a aspectos sistemáticos [56].

La reutilización de código abarca la idea de lograr utilizar parcial o completamente un programa de computador escrito alguna vez, para ser usado en otro escrito posteriormente. Es una técnica común que busca ahorrar tiempo y energía a los programadores al reducir el trabajo redundante. Por esta razón el activo más popular para la reutilización es el código [57].

Es común que un proyecto complejo o de una envergadura considerable sea abordado dividiendo el trabajo entre diferentes equipos de desarrollo; construyen componentes que pueden ser conectados en sus fases finales. Por supuesto, esto no suele funcionar de una manera tan simple; los equipos pueden tener diferentes interpretaciones de las especificaciones de cada uno de los componentes o incluso puede suceder que los grupos terminen construyendo básicamente los mismos componentes del sistema, pero en tecnologías diferentes. El costo de construir sistemas de esta forma es enorme ya que prácticamente el mismo sistema, o un simple subconjunto de éste, es implementado tantas veces como equipos de trabajo existan. Ello aumenta considerablemente los costos, teniendo en cuenta que el mantenimiento del código producido es proporcional a la cantidad de sistemas construidos por cada equipo, estos aumentan aun más [57].

Esta situación contrasta con la visión promulgada por MDA, la arquitectura del sistema es definida solo en el último momento, es decir, las dependencias entre niveles son externalizadas y adicionadas solo cuando el sistema es desplegado, de tal forma que cada modelo sea reutilizable y constituye, en sí mismo, el principal activo en el proceso desarrollo de software. Una breve descripción de los modelos que propone MDA es presentada a continuación.

4.2.3.1. Modelo independiente de computación

Está formado por los modelos que reflejan la arquitectura de la organización sin identificar los conceptos que serán parte del sistema a desarrollar; es un modelo independiente del software, utilizado para describir un sistema de negocios, ciertas partes pueden ser soportadas por sistemas software, pero por si solo es independiente de él [16], [52].
El CIM modela los requisitos del sistema describiendo la situación en la que será utilizado y las aplicaciones que llevará a cabo, sirve de ayuda para entender el problema [15].

4.2.3.2. Modelo independiente de plataforma

Es una vista del sistema desde el punto de vista independiente de la plataforma, permitirá aplicarlo de igual modo sobre diferentes plataformas, por ejemplo J2EE, .NET, CORBA, etc. Representa la estructura, funcionalidad y restricciones del sistema, este modelo es la base para todo el proceso de desarrollo y es el único que debe ser creado íntegramente por el desarrollador [16], [52]. Describe un sistema software que soporta algún negocio sin conocimiento alguno de la plataforma de implementación final dentro de éste, el sistema es modelado teniendo en consideración el mejor soporte al negocio.

4.2.3.3. Modelo específico de plataforma

Es una vista del sistema con detalles específicos de la plataforma elegida para su implementación, es descrito con pleno conocimiento de la plataforma de implementación final.

Es el resultado de una o varias transformaciones del PIM ya que para cada plataforma de tecnología específica debe generarse un PSM [16].

El PSM está mucho más cercano al código que el PIM, pero puede incluir más o menos detalle dependiendo de su propósito. Expone un conjunto de conceptos técnicos que representan las diferentes formas o partes que componen un sistema y los servicios que provee, también expone conceptos que explican los diferentes elementos que suministra una plataforma para implementar una aplicación en un sistema y los requerimientos de conexión y uso de las partes que la integran y la conexión de las aplicaciones a ella [14]. La mayoría de los sistemas actuales comprenden muchas tecnologías, por ello es común tener muchos PSM's por cada PIM.

4.2.3.4. Implementación específica de la plataforma

Es el código del sistema a desarrollar. El código será generado automática o semiautomáticamente, en su totalidad o en parte, mediante reglas de transformación [14], [16].

4.2.4. Metamodelado

Es simplemente un modelo de un lenguaje de modelado que define la estructura, semántica y restricciones para una familia de modelos[26]. Un modelo es capturado por un metamodelo particular, por ejemplo, un modelo que emplea diagramas UML es capturado por el metamodelo UML, el describe como los modelos UML pueden ser estructurados, los elementos que puede contener y las propiedades de estos elementos. La figura 11 muestra estas relaciones [57].

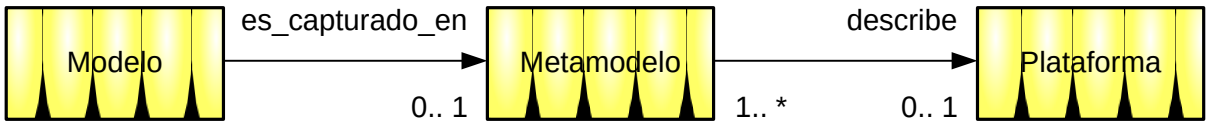

Figura 11. Modelos, metamodelos y plataformas

El metamodelado es un mecanismo para definir lenguajes de modelado que resulta muy útil en el contexto de MDA debido a que estos no tienen que estar basados en texto y de hecho a menudo no lo están; pueden tener, por ejemplo, una sintaxis gráfica, como UML. Esta situación descalifica a la gramática BNF[27] (Backus Naur Form) de uso muy común, que aunque resulta apropiada para definir lenguajes basados en texto, está restringida a solo este tipo de lenguajes, mientras que el metamodelado no [52].

Debido a que un metamodelo es también un modelo, por sí solo debe ser escrito en un lenguaje de especificación formal el cual es llamado metalenguaje, de tal forma que BNF es un metalenguaje (ver figura 12).

[26] Una familia de modelos es un grupo de modelos que comparten una sintaxis y semántica comunes

[27] El BNF (Backus-Naur form) es una metasintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir lenguajes formales, se utiliza extensamente como notación para las gramáticas de los lenguajes de programación basados en texto, de los sistemas de comando y de los protocolos de comunicación.

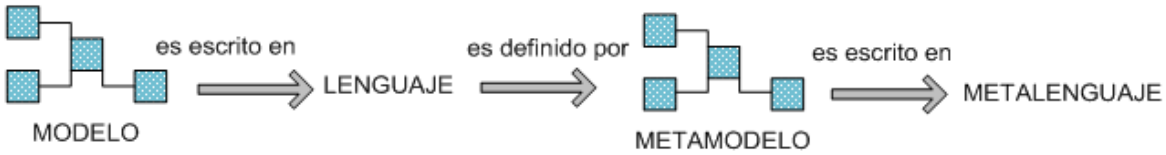

Figura 12. Modelos, lenguajes, metamodelos y metalenguajes.

El metamodelo define completamente el lenguaje y no es necesario o útil hacer la distinción entre el lenguaje y el metamodelo que define el lenguaje ya que para todos los propósitos prácticos son equivalentes. Debido a que un metalenguaje es un lenguaje en sí, puede ser definido por un metamodelo escrito en otro metalenguaje.

En teoría, hay un número infinito de niveles de relaciones de modelo-lenguaje-metalenguaje, sin embargo, los estándares definidos por OMG usan una arquitectura de tan solo cuatro niveles ya que no es muy útil seguir adicionando niveles. En vez de definir un nivel M4, OMG estableció que todos los elementos del nivel M3 deben ser definidos como ejemplares de conceptos del mismo nivel. De hecho, la separación entre los 4 niveles de la arquitectura, sirve simplemente como un mecanismo estructurado que facilita trabajar con los modelos y las clasificaciones; lo realmente importante es que cada elemento tenga un metaelemento clasificado, a través del cual un metadato pueda ser accedido, de tal forma que cualquier modelo pueda ser construido y cualquier sistema pueda ser descrito, como se explica a continuación [52].

4.2.4.1. Nivel de modelado M0, las instancias

En el nivel M0 se encuentra el sistema en ejecución, contiene los datos de la aplicación. En este nivel existen ejemplares “reales”, todos con sus propios datos y que pueden existir de diferentes formas, como un dato en una base de datos o un objeto de un sistema que está en ejecución. En el momento de modelado del negocio las instancias en el nivel M0 son los ítems del negocio en sí, por ejemplo, los clientes, las facturas y los productos. Cuando se está modelando software, los ejemplares son las representaciones software de los ítems del mundo real, por ejemplo, la versión computarizada de las facturas o las órdenes, la información del producto y los datos personales [16] [52].

4.2.4.2. Nivel de modelado M1, el modelo del sistema

Contiene modelos, por ejemplo, un modelo UML de un sistema software. Existe una relación definida entre los niveles M0 y M1, los conceptos en el nivel M1 son todos categorizaciones o clasificaciones de los ejemplares en el nivel M0; así mismo, cada elemento en el nivel M0 es siempre un ejemplar de un elemento en el nivel M1. Los elementos M1 directamente especifican que instancias se ven en el nivel M0. Los elementos que existen en el nivel M1 (clases, atributos y otros elementos de modelos) son por si solos ejemplares de clases en M2, el próximo nivel. Este nivel contiene la aplicación, las clases de un sistema orientado a objetos, o la tabla de definiciones de una base de datos relacional. Este es el nivel en el cual tiene lugar el modelado de la aplicación [16] [52].

4.2.4.3. Nivel de metamodelo M2, el modelo del modelo

El modelo que reside en el nivel M2 es llamado metamodelo. Cada modelo UML en el nivel M1 es una instancia del metamodelo UML como fue definido en la especificación UML 2 [58]. Los

elementos en el nivel M2 especifican aquellos del nivel M1. La misma relación que está presente entre elementos de niveles M0 y M1 existe entre elementos de M1 y M2 ya que cada elemento en M1 es un ejemplar de los elementos en M2 y los elementos de M2 categorizan los de M1. Una clase M1 define ejemplares en el nivel M0, y una clase M2 define lo correspondiente en el nivel M1, por ejemplo, una clase o una asociación.
Por lo general, la notación que es empleada para bosquejar un metamodelo es la misma usada para bosquejar un modelo. Este es el nivel en el cual las herramientas operan, donde es posible crear y cambiar clases y otros elementos de los modelos. Desde el punto de vista del modelador y la herramienta de modelado, las clases y otros elementos del modelo, son los ejemplares de trabajo [16] [52].

4.2.4.4. Nivel de meta-metamodelado M3, el modelo de M2

Representa las entidades fundamentales de cualquier sistema, así como las construcciones básicas. Este es el nivel en el cual se encuentran los lenguajes de modelado y los metamodelos funcionales. Siguiendo la misma línea de los niveles anteriores, un elemento en el nivel M2 es visto como un ejemplar de un elemento en el nivel M3 o meta-nivel. Nuevamente, la misma relación que está presente entre elementos de niveles M0 y M1, y M1 y M2, existe entre elementos de M2 y M3. Cada elemento en M2 es un ejemplar de un elemento en M3 y cada elemento en M3 categoriza elementos M2.

Por lo general la notación que es utilizada para bosquejar un meta-metamodelo, es la misma notación utilizada para bosquejar un metamodelo y un modelo. Para OMG, MOF (Meta-Object Facility) es el lenguaje estándar del nivel M3, todos los lenguajes de modelado como UML (Unified Modeling Languaje), CWM (Common Warehouse Metamodel), etc., son ejemplares de MOF [16] [52].

4.2.5. Transformaciones

En la sección 4.2.3., que habla de MDA, es descrito el papel que desempeñan los modelos dentro del proceso de desarrollo de MDA y a la vez es mencionado, que para pasar de un modelo a otro deben realizarse una o varias transformaciones. Estas transformaciones son esenciales en aquellos procesos de desarrollo de software que siguen un enfoque MDA. La idea es que estas transformaciones sean realizadas de forma automática o semiautomática por herramientas, que tomen un modelo como entrada y produzcan un segundo modelo o varios modelos como salida.

En general, cuando se habla sobre transformaciones, los siguientes términos son manejados:

- Regla de transformación: es una descripción de cómo uno o más constructores, en un lenguaje inicial, pueden ser transformados en uno o más constructores, en un lenguaje final. Las reglas deben ser especificadas sin ambigüedad, de tal forma que un modelo, o parte de él, pueda ser utilizado para crear otro o una parte.
- Definición de transformación: es un conjunto de reglas que describen como un modelo, en un lenguaje inicial, puede ser transformado en un modelo, en un lenguaje final.
- Transformación: es la generación automática o semiautomática de un modelo final a partir de un modelo inicial, de acuerdo a una definición de transformación [52].

Existen dos tipos de transformaciones, las transformaciones verticales, realizadas entre diferentes niveles de abstracción (CIM, PIM, PSM, PSI), y las transformaciones horizontales, realizadas entre los diferentes modelos de un mismo nivel de abstracción. Una transformación

debe caracterizarse por conservar el sentido original entre el modelo inicial y el modelo final, claro está que el significado original de un modelo solo puede ser conservado en la medida en que éste pueda ser expresado en ambos modelos, por ejemplo, la especificación de comportamiento puede ser parte de un modelo UML, pero no de un modelo entidad relación o ER (Entity-Relationship). Aun así, el modelo UML puede ser transformado en un modelo ER, conservando las características estructurales del sistema solamente [52].

Cabe agregar que las definiciones expuestas anteriormente no limitan de forma alguna el lenguaje inicial y final. Ello significa que el modelo inicial y el modelo final pueden ser escritos en el mismo lenguaje o en diferentes.

4.4. Estándares

Los conceptos fundamentales de MDA pueden ser aplicados sin el uso de estándares; sin embargo, para que la comunidad del software adopte más fácilmente sus lineamientos propuestos, es necesario tener un conjunto de estándares sobre modelado que permitan a la industria el desarrollo de herramientas MDA que faciliten la operación conjunta de soluciones y de las mismas. Algunos de los más importantes estándares de modelado son definidos por OMG, gracias a su interés particular en MDD. Esta sección describe los estándares más relevantes y la forma en que éstos trabajan juntos.

4.4.1. Meta Object Facility

Es un estándar OMG para metamodelado que define el lenguaje para definir lenguajes de modelado. Está en el nivel más alto, el nivel M3. MOF es el lenguaje en el que las definiciones de CWM y UML (ver numeral 4.4.4. y 4.4.5. respectivamente) son escritas, de hecho, MOF es definido usándose a sí mismo [58].

Además de definir lenguajes de modelado, también permite la construcción de herramientas para la definición de dichos lenguajes, para lo cual proporciona algunas funcionalidades adicionales, como una especificación de la interfaz para un repositorio que permite conseguir información sobre modelos M1 a partir de un repositorio basado en MOF y un modelo de intercambio con el cual define una forma estándar de generar un formato de intercambio para modelos, que está basado en el lenguaje de marcas extensible o XML (Extensible Markup Language) y es llamado XML de intercambio de metadatos o XMI (XML Metadata Interchange) [59]. Gracias a que MOF es definido usándose a sí mismo, XMI también puede ser utilizado para generar estándares de formatos de intercambio para metamodelos.

El papel de MOF dentro de MDA es ofrecer conceptos y herramientas para trabajar con los lenguajes de modelado. Usando la definición de MOF es posible definir transformaciones entre lenguajes de modelado.

4.4.2. Query, Views, and Transformations

Es un estándar compatible con el conjunto de recomendaciones de MDA (MOF, UML, etc.), creado para la transformación de modelos. Es de naturaleza híbrida “declarativa/imperativa”, especificado entre dos dimensiones ortogonales, la dimensión del lenguaje y la dimensión de la interoperabilidad.

Para QVT, el modelo inicial, el modelo final y las transformaciones, deben estar acorde con cualquier metamodelo MOF, lo que en últimas quiere decir que la sintaxis abstracta de QVT debería estar acorde con un metamodelo MOF 2.0. El lenguaje QVT integra el estándar OCL 2.0. (Object Constraint Language, ver numeral 4.4.3.) además de definir lenguajes específicos de dominio llamados, “relaciones”, “núcleo” y “traslaciones operacionales” [60].

4.4.3. Object Constraint Language

Es un lenguaje declarativo en el cual es posible escribir expresiones sobre modelos, por ejemplo, derivación de reglas para atributos, el cuerpo de operaciones de solicitud, constantes, precondiciones y poscondiciones. Cualquier expresión OCL evalúa un valor y describe cual valor es, pero no estipula como debería ser calculada la expresión. Por supuesto, las expresiones pueden ser traducidas a lenguajes de programación (por ejemplo, java) para especificar cómo es ejecutada. Ellas son “puras” en el sentido de que no tienen efectos laterales [61].

OCL extiende el poder expresivo de UML y MOF al permitir crear modelos de un sistema más precisos y completos. En el contexto de MDA significa que el modelo inicial de una transformación se vuelve más rico, ello hace posible generar un modelo final mucho más completo.

Como OCL puede ser utilizado en el nivel MOF, permite escribir expresiones sobre metamodelos, de hecho, el primer uso de OCL con MOF ha sido en la definición del metamodelo UML. El mismo enfoque es tomado en la definición de otros estándares OMG, tales como CWM y él mismo MOF, situación que lleva a obtener las mismas ventajas que el uso de OCL da a modelos UML, es decir, los metamodelos se vuelven más precisos y completos, constituyendo una mejor especificación, con menor ambigüedad. Además de traer más precisión a los modelos y a las definiciones de lenguajes, puede ser utilizado muy efectivamente en la definición de transformaciones, ya que muchas de éstas solo pueden ser aplicadas bajo ciertas condiciones que OCL puede especificar [52] [61].

4.4.4. Common Warehouse Metamodel

CWM es un lenguaje de modelado que pretende modelar aplicaciones de almacenamiento de datos. Tiene mucho en común con el metamodelo UML, pero a diferencia de éste, tiene algunas metaclases especiales, por ejemplo, para modelar bases de datos relacionales. Los desarrolladores de CWM han eliminado de UML todo aquello que no era necesario para su propósito y han adicionado los detalles específicos para almacenamiento de datos. Las partes de comportamiento del metamodelo UML, como la máquina de estados, no está en CWM.

Debido a que los datos de almacenamiento son una tecnología que combina información de muchas fuentes diferentes, el metamodelo CWM tiene un alcance muy amplio e incluye metamodelos simples para ciertas cosas como las bases de datos relacionales, XML, visualización de información, minería de datos, entre otras [62].

4.4.5. Lenguaje de modelado unificado

Es el lenguaje de modelado estándar en el nivel M2 de más amplio uso (la gran mayoría de los modelos que son creados, son modelos UML), es definido usando MOF. La principal causa que motivó su aparición fue la de poder disponer de un proceso unificado de modelado para

grandes sistemas, que fuese escalable y utilizable tanto por personas como por computadores [15]. UML es un lenguaje gráfico de modelado de sistemas software respaldado por OMG, sirve para visualizar, especificar, construir y documentar un sistema. Ofrece un estándar para describir aspectos conceptuales tales como procesos de negocio y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables. UML permite modelar arquitecturas, objetos, interacciones entre objetos, datos y aspectos del ciclo de vida de una aplicación, así como otros aspectos más relacionados con el diseño de componentes incluyendo su construcción y despliegue [63].

El metamodelo UML es una instancia del modelo MOF que tiene la misma estructura de éste, ya que reutiliza parte de su definición. Debido a que UML es utilizado para modelar mucho más que metamodelos, contiene muchas más metaclases que MOF [63].

4.4.6. Las semánticas de acción UML

Son una extensión de UML que proporcionan el metamodelo para un "lenguaje de acción". Dicho lenguaje tiene como propósito proporcionar los fundamentos necesarios para semánticas dinámicas en UML. En principio, ellas pueden ser utilizadas para escribir directamente modelos UML ejecutables. En muchos sentidos, parece que las semánticas de acción consideran a UML más como un lenguaje útil para un ensamblador, antes que un lenguaje útil para un modelador [52] [63].

4.4.7. Perfiles UML

Son mecanismos de especialización que hace parte de UML y definen una forma específica de usar UML, por ejemplo, el perfil java para UML define una forma para modelar código Java en UML. Los perfiles son definidos por un conjunto de estereotipos, un conjunto de restricciones relacionadas y un conjunto de valores etiquetados. Al reutilizar el metamodelo UML, un perfil define un nuevo lenguaje, que puede ser específico a ciertas plataformas, como Java o C++. Un modelo con un perfil y su estereotipo aplicado puede ser utilizado para definir un PSM.

Una alternativa a los perfiles, es definir un nuevo metamodelo que cumpla con el propósito específico para el cual el perfil define una variante especializada de UML [52] [63].

Un estereotipo extiende el vocabulario básico de UML. Pueden ser adjuntados a un elemento de un modelo para significar que el elemento es de alguna manera diferente de otros. Definir un estereotipo es similar a crear una subclase de un tipo UML existente que puede incluir atributos que se comporten como etiquetas [52].

Los estándares XML y XMI son descritos en las secciones 5.1.4.2 y 5.1.4.3, respectivamente.

4.5. Herramientas

En la actualidad varios grupos de trabajo están interesados en desarrollar herramientas que permitan construir software siguiendo un enfoque MDA. Aunque son muchos los vendedores que atribuyen a su herramienta la capacidad de manejar los conceptos fundamentales de MDA, la verdad es que al respecto aún falta mucho por hacer, afortunadamente, MDA cuenta con el gran apoyo que brinda OMG y sus miembros, entre los cuales están actores muy importantes en la industria del software, para promocionar e incentivar el desarrollo de este tipo de herramientas de las cuales trata esta sección.

4.5.1. AndroMDA

Es un marco generador extensible que se adhiere a MDA. Para la generación de código la herramienta de software libre toma modelos UML en formato XMI como entrada y los transforma en código. Permite la generación de código en diferentes lenguajes de programación. Está compuesta por "cartuchos" que pueden ser considerados como un tipo especial de *plugin,* en ellos están definidos los metamodelos y las reglas de transformación utilizadas para convertir elementos del modelo de acuerdo al metamodelo.
Soporta diseño modular, contiene todo el metamodelado UML, valida la entrada de diferentes modelos y da una mayor abstracción en la transformación modelo a modelo (MDA), mientras que al mismo tiempo permite generar cualquier tipo de salida de texto utilizando plantillas [64].

4.5.2. ArgoUML

Es una herramienta propuesta por la comunidad de software libre, diseñada para el desarrollo de sistemas de información. Nace a finales de la década de los 90. Fue concebida como una herramienta CASE[28] para realizar el análisis y el diseño en el desarrollo de sistemas orientados a objetos y evolucionó hasta los sistemas de información de la Web (SI Web). Fue desarrollada utilizando el lenguaje Java, cubre todas las etapas del ciclo de vida de un sistema software, por lo que con ésta es posible definir modelos para todos los niveles de abstracción que propone MDA; genera código Java por defecto, aunque existen módulos auxiliares que permiten generar código en otros lenguajes como C#, C++, PHP4, etc. Las transformaciones con esta herramienta deben definirse usando Java como lenguaje de programación, solo puede realizar transformaciones verticales entre los modelos y el usuario debe participar activamente en la definición de tales modelos [65].

4.5.3. Xcarecrows

Es un plugin de Eclipse que tiene versiones de prueba y de software libre, proporciona una herramienta gráfica utilizada para el diseño de modelos de sistemas o aplicaciones de acuerdo al proceso definido por MDA de OMG. Está enfocada en la calidad del modelado de comportamiento y convierte, automáticamente, modelos en componentes ejecutables para interfaces bien definidas del entorno [66].

4.5.4. Atlas Transformation Languaje

Es una herramienta de software libre propuesta por el grupo de investigación ATLAS INRIA & LINA desarrollado por OBEO e INRIA en respuesta al RFP[29] (Request For Proposal) de QVT. Es un lenguaje de transformación de modelos especificado tanto como un metamodelo como una sintaxis textual concreta. El lenguaje es híbrido, declarativo e imperativo. El estilo más utilizado es el declarativo, el cual permite expresar transformaciones fácilmente. Actualmente ATL forma parte del subproyecto M2M[30] dentro del Eclipse Modeling Project [67].

[28] Las herramientas CASE (Computer Aided Software Engineering) son todo un conjunto global de herramientas que pueden utilizarse para facilitar los procesos de desarrollo software, su definición es muy genérica y abarca casi cualquier programa que pueda utilizarse durante el desarrollo de software.
[29] RFP (Request For Proposal) es una invitación para que proveedores o demás personas o grupos interesados envíen una propuesta sobre un estándar, servicio, metodología, etc., en específico. El proceso de RFP define una estructura para la toma de decisiones y permite que los riesgos y beneficios sean definidos claramente por adelantado.
[30] Para más información sobre M2M visitar la página http://www.eclipse.org/m2m/

4.5.5. Acceleo

Es una nueva tendencia para la generación de código fuente abierto, es diseñado para aplicar eficazmente MDA y está enfocado en la mejora de la productividad del desarrollo de software. Es una herramienta nativa integrada con Eclipse. Incluye, a la vez, herramientas y editores para que sea fácil de aprender y adaptable a cualquier tipo de proyecto o tecnología. Proporciona un modelo de ingeniería de innovaciones: generación incremental, metamodelos de interoperabilidad, árboles de sintaxis y plantillas de personalización. Además, implementa un conjunto de módulos listos para uso disponible para tecnologías JEE, C#, Java, PHP, etc. [68].

4.5.6. Blu age Build

Es una herramienta software propietaria, compatible con MDA, basada en Eclipse y financiada por la Unión Europea. Es capaz de transformar modelos de aplicaciones empresariales J2EE y .NET. El módulo de edición Blu Age 2009 es un entorno integrado para construir y depurar modelos, a la vez que generar aplicaciones JEE usando el lenguaje ATL [69].

4.5.7. QVT Operacional de Borland

Es una implementación del estándar QVT realizada por Borland para su herramienta Together. No soporta relaciones declarativas, tan solo transformaciones imperativas. Actualmente ha sido aprobada su liberación por Borland y está en fase de integración en el subproyecto M2M dentro del Eclipse Modeling Project, al igual que ATL [67].

4.5.8. motionModeling

Es una herramienta de software libre que proporciona un entorno MDA completo, está integrando con AndroMDA para la generación de código. La herramienta está diseñada para permitir ser ampliada con Java Script, de tal forma que pueda implementar nuevas aplicaciones o nuevos metamodelos; a la vez soporta UML 1.4 [70].

4.5.9. SmartQVT

Es una herramienta de software libre desarrollada por France Telecom R&D. Implementa el lenguaje MOF 2.0. Utiliza QVT-Operational, un lenguaje híbrido con una estructura de reglas declarativas y un flujo de ejecución imperativo. Requiere de un intérprete externo de Python, debido a que el analizador de QVT ha sido desarrollado utilizando este lenguaje [67].

4.5.10. OptimalJ

Es una herramienta MDA, desarrollada por Compuware, utiliza MOF para soportar estándares como UML y XMI. Es un entorno de desarrollo que permite generar aplicaciones J2EE completas a partir de un PIM. En OptimalJ existen tres tipos de modelos [71], el modelo del dominio, que describe el sistema a un alto nivel de abstracción, sin detalles de implementación y que corresponde al PIM de la aplicación y su elemento principal es un modelo de clases del negocio; el modelo de la aplicación que describe el sistema desde el punto de vista de una tecnología determinada (J2EE) y que contiene los PSM de la aplicación que se generan automáticamente a partir de un modelo del dominio y el modelo de código, que es el código generado a partir de un modelo de la aplicación [72].

OptimalJ distingue varios tipos de patrones: “patrones de transformación” entre modelos (para transformar un PIM en PSM y un PSM en código) y “patrones funcionales” que aplican transformaciones dentro de un modelo.
En el 2008 Compuware decidió descontinuar OptimalJ debido a reestructuraciones internas de la empresa, aun cuando se le considera un ambiente de desarrollo muy superior a nivel de soluciones técnicas es relativamente costoso y, por lo tanto, la empresa encontró difícil ganar un mercado que compartía con la comunidad de desarrollo Java y los productos que ofrece.

4.5.11. ArcStyler

Es una herramienta MDA, desarrollada por iO-Software, utiliza MOF para soportar estándares como UML y XMI, además de JMI para el acceso al repositorio de modelos. Integra herramientas de modelado (UML) y desarrollo (ingeniería inversa, explorador de modelos basado en MOF, construcción y despliegue) con la arquitectura CARAT[31]. Ella permite la creación, edición y mantenimiento de “cartuchos MDA” (MDA-Cartridge) que definen transformaciones. También incluye herramientas relacionadas con el modelado del negocio y el modelado de requisitos por lo que cubre todo el ciclo de vida. Un cartucho contiene un conjunto de reglas de transformación y se instala como un *plugin*, puede incluir en los modelos aspectos específicos de una plataforma utilizando perfiles UML [72], [73].

4.5.12. Eclipse

Es una comunidad que desarrolla proyectos de código abierto, están centrados en la construcción de una plataforma de desarrollo abierto compuesta de marcos extensibles, herramientas y ejecutables para construir, desplegar y manejar el desarrollo de software, en todo su ciclo de vida. De esa forma, define e implementa un entorno integrado de desarrollo de código con capacidades multiplataforma, su objetivo final es la ayuda a la comunidad relacionada con el desarrollo de diverso tipo de aplicaciones. Los proyectos desarrollados con Eclipse están construidos sobre el entorno de ejecución Equinox OSGI (Open Source Gateway Initiative). Abarca lenguajes dinámicos y estáticos, aplicaciones para clientes, aplicaciones del lado del servidor, entornos para el modelado de negocio y aplicaciones empotradas y móviles, entre otros [74], [75]. Para que la plataforma sea más asequible y cómoda para el desarrollador, Eclipse ofrece una arquitectura extensible llamada PDE (Plug-in Development Environment) que permite crear un entorno con las vistas, editores y perspectivas necesarias, para desarrollar aplicaciones propias. Desde septiembre de 2002, en su versión 2.0, brinda soporte para MDA, permitiendo a los desarrolladores crear aplicaciones basadas en modelos, gracias a la integración de las herramientas de modelado y de soporte al entorno [76].

4.5.13. eUML2 Studio

Es un poderoso conjunto de herramientas, desarrollado por Soyatec [77], para Eclipse y que está especialmente diseñado para emplear UML en el nivel de desarrollo, asegurando la calidad del software y reduciendo el tiempo de desarrollo. Viene en dos presentaciones, eUML2 Free Edition que cuenta con todas las características básicas necesarias para desarrolladores en Java y que es libre de usar (incluso para propósitos comerciales) y eUML2 Studio Edition que es una extensión de la versión anterior y posee características nuevas y mejoradas.

[31] Para más información sobre la arquitectura CARAT visitar la página http://dis.um.es/~jmolina/documentos/ CartuchosMDA.pdf

eUML2 Free Edition es una herramienta de software libre, madura, compatible con Ganymede, UML 2.1 y soporta el formato XMI de OMG. La version comercial, Studio Edition, alguna vez estuvo asociada con Omondo y es posible afirmar que sus primeras versiones son iguales; ofrece ingeniería inversa avanzada, herramientas análisis, UML para MDD. Está compuesta por cuatro herramientas:

- eUML2 Modeler: combina Java y UML dentro de Eclipse.
- eDepend: rastrea y analiza dependencias Java.
- eEMF Modeler: diseño visual de modelos Ecore propios.
- eDatabase: gestión de las necesidades de bases de datos.

Las principales características con las que cuenta Studio Edition son:

- Editor de diagrama de clases y paquetes.
- Editor de diagrama de secuencias.
- Editor de diagramas Ecore.
- Soporte de características de JDK 1.5.
- Sistemas en tiempo real, sincronización de modelos.
- Exportar e importar modelos.
- Soporte del formato XMI de OMG.
- Asistentes de modelado.

4.5.14. Green UML

Es un editor de diagramas de clases UML [78], soporta ingeniería directa e inversa, utilizado para crear diagramas de clases UML a partir de código existente o generar nuevo código dibujando un diagrama de clases. Es una herramienta estable, con licencia de Eclipse, para su versión 3.4, soporta XMI. Fue desarrollado persiguiendo un objetivo académico, enfocando a los estudiantes en el diseño, sin embargo, ha probado ser una herramienta robusta y flexible, que puede proporcionar soporte UML en cualquier ambiente. También soporta exploración incremental de todo el código fuente, seleccionando una clase y haciendo operaciones de exploración incrementales, pueden ser traídos a un diagrama todos los tipos con los cuales la clase está relacionada. Todas las relaciones utilizadas por Green son sus *plugins*, esto significa que en cualquier momento es posible remover una de las relaciones predefinidas, o si está interesado, desarrollar otra relación como otro *plugin*. Puede guardar diagramas como simples archivos XML y también en formato jpg (para imprimir o incluir en una página web). A la vez, soporta la funcionalidad de vista de acercamiento.

4.5.15. Visual Paradigm SDE

Es un plug-in de modelado UML integrado de forma transparente con la mayoría de los Entornos de Desarrollo Integrado o IDE's (Integration Development Environment) líderes en el mercado [79]. SDE integra UML, ERD[32] (Entity Relationship Diagram), BPD[33] (Business Process Diagram) y Mind Map[34] con Eclipse, Visual Studio .NET, Netbeans/Sun ONE, Borland JBuilder, IntelliJ IDEA, Oracle JDeveloper, BEA WebLogic Workshop. Es usado para construir

[32] ERD es un diagrama empleado en la industria de la ingeniería del software para representar modelos entidad-relación. SDE para Eclipse soporta modelado de datos físico, lógico y conceptual [79].

[33] BPD es un diagrama que representa la vista de comunicación de procesos [79].

[34] Mind Mapping es una herramienta disponible para muchos propósitos, que resulta especialmente útil en reuniones, a la hora de gestionar el concepto de "lluvia de ideas", ya que facilita la organización de ideas, conceptos, palabras, tareas, e incluso, algunas veces, también ayuda a capturar requisitos y procesos de negocios.

código a partir de modelos o modelos a partir de código (ingeniería inversa) y para generación de reportes, entre otras cosas.

SDE para Eclipse es una herramienta/extensión UML CASE. Este software de modelado UML soporta el ciclo de vida completo del desarrollo de software: análisis, diseño, implementación, pruebas y despliegue. Ayuda a la construcción rápida de aplicaciones, de forma que tengan calidad, sean mejores y más baratas; permite dibujar todos los tipos de diagramas UML en Eclipse, realizar ingeniería inversa desde código Java a diagramas de clases, generar código Java y generar documentación. Algunas de las principales características de SDE son:

- Soporte de UML versión 2.2.
- Diagramas de Procesos de Negocio.
- Interoperabilidad con modelos UML 2 a través de XMI (nueva característica).
- Editor de figuras.
- Diagramas de flujo de datos.
- Alta velocidad a la hora de cargar y guardar los proyectos.
- Editor de detalles de casos de uso.
- Diagramas EJB, visualización de sistemas EJB.
- Ingeniería inversa a bases de datos.
- Ingeniería inversa, de código a modelo y de código a diagrama.
- Generación de código de modelo a código, diagrama a código.
- Importación de proyectos Rational Rose.
- Importación y exportación de archivos XMI.
- Integración con Visio, dibujo de diagramas UML con plantillas (*stencils*) de MS Visio.
- Generador de informes en PDF/HTML.

CAPÍTULO 5.

ARQUITECTURA

Esta sección propone una arquitectura para dar soporte a la construcción de familias de sistemas telemáticos con un enfoque MDA y aclara algunos conceptos tomados en cuenta para la definición de la misma. Los estándares y las soluciones software a utilizar son seleccionados, de los mencionados en el capítulo 4.

5.1. Consideraciones iniciales

Algunos de los conceptos e ideas empleados en el capítulo anterior pueden resultar ambiguos y prestarse a interpretaciones particulares. Para evitar confusión en la definición posterior de la arquitectura, esta sección aclarará los conceptos e ideas más importantes relacionados con MDA.

5.1.1. Arquitectura Software

Hasta el momento ninguna definición sobre Arquitectura Software es respaldada unánimemente por la totalidad de la comunidad del software, incluso existen grandes compilaciones de definiciones alternativas o contrapuestas, por ejemplo la colección que se encuentra en el SEI[35]. Para la definición de la arquitectura el presente proyecto tomará como referencia la definición propuesta en las conferencias de clase de la materia ambientes de desarrollo de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca [80]; según ellas, una arquitectura software es el conjunto de decisiones significativas acerca de la organización de un sistema de software, la selección de los elementos estructurales a partir de los cuales se compone el sistema y las interfaces entre ellos, junto con su comportamiento y la composición de sus elementos estructurales.

5.1.2. Naturaleza de MDA

En la literatura existente sobre MDA están identificadas diversas posiciones respecto a este tema. Existen diversos trabajos que la definen como una aproximación al desarrollo de software, un estándar de OMG, un conjunto de estándares de OMG o un método de desarrollo [81]. En capítulos anteriores MDA ha sido definido, más la naturaleza de esta iniciativa no ha sido aclarada. Al respecto, y con base en la guía de MDA [16] y a los editores del número de la revista IEEE Software dedicado a MDD [82], es posible concluir que MDA es un conjunto de estándares que permiten seguir la aproximación que defiende MDD y pretende convertirse en un estándar para lo cual, teniendo en cuenta su estado actual, necesita definir y construir métodos precisos que proporcionen a los equipos de desarrollo de software, pautas y técnicas que puedan seguir y utilizar fácilmente.

5.1.3. Transformaciones automáticas

La guía de MDA contempla tres grados de automatización para la ejecución de transformaciones entre los diferentes modelos que propone [16]. La primera opción es que las

[35] Para consultar el listado de definiciones del SEI sobre arquitecturas de software, visitar la página http://www.sei.cmu.edu/architecture/definitions.html

transformaciones sean realizadas de forma completamente manual por parte de los usuarios del sistema; la segunda, que la transformación sea semiautomática, es decir, que requiera la participación del usuario, lo que llevaría a diferentes grados de interacción del usuario con la herramienta de transformación; y por último, que las transformaciones sean completamente automáticas y se realicen a partir de los PIM's. Estas diferentes posibilidades plantean un escenario confuso que dificulta llegar a un consenso sobre el grado de automatización que deben tener las transformaciones entre modelos. Al respecto existen diferentes posiciones, en su gran mayoría, corresponden a consideraciones personales. En el presente documento, y con base en la guía oficial de MDA [16], es concluido que no es necesario que las transformaciones sean automáticas para que sigan el enfoque MDA, pero con el fin de que las ideas que propone el MDD sean más fácilmente aceptadas, es conveniente que las transformaciones sean realizadas de la forma más automática posible [81]. El caso de estudio de este proyecto de grado realiza transformaciones de tipo semiautomáticas.

5.1.4. Concepto de plataforma

La definición del concepto de plataforma no es precisa en [16], el PIM no está claramente delimitado y la línea divisoria entre PIM y PSM es borrosa. Los siguientes tres aspectos relacionados con el concepto de plataforma son planteados, tecnología concreta, que relaciona plataforma con tecnologías tales como Enterprise JavaBeans. Tecnología abstracta, que considera plataforma a la programación orientada a objetos. Y generador de implementación, que señala que un PSM debe incluir todos los detalles necesarios para producir una implementación y que, por lo tanto, cualquier especificación del sistema que permita obtener de manera automática la implementación es un PSM. Esta es una concepción más general que la primera, no es necesario que el PSM esté expresado en términos de una tecnología específica, solo es necesario que sea posible llegar a la implementación.

En resumen, es posible afirmar que sobre el concepto de plataforma existe un intenso debate en la comunidad de Ingeniería del Software y que en este proyecto de grado se relacionará platafroma con un conjunto de artefactos (componentes o subsistemas) que forman una estructura común a partir de la cual podemos derivar (desarrollar, construir) sistemas de una forma eficiente. Los sistemas derivados de una plataforma no sólo comparten código, sino requisitos y arquitectura. Se da por tanto un proceso de reutilización natural de los artefactos de la plataforma en los diferentes sistemas, situación que se adecua perfectamente al enfoque de MDA y SPL.

5.2. Captura de requisitos

La aproximación metodológica propuesta en [38] define requisito como aquella propiedad que un sistema (hardware - software) debería tener para ser exitoso en el entorno en el cual se usará. También propone que los requisitos sean clasificados en funcionales, no funcionales y de información. De acuerdo a ella, teniendo en cuenta las propiedades deseables de los requisitos [38], y las actividades propuestas en el capítulo 3 en la sección correspondiente a la ingeniería de dominios, a continuación los requisitos del presente proyecto son listados.

5.2.1. Requisitos de información

Son los requisitos de almacenamiento de información que deberá satisfacer el sistema y que responden a la pregunta ¿qué información, relevante para los objetivos de la organización, deberá almacenar el sistema?

- **Almacenar diagramas UML**: el sistema debe almacenar la información del diagrama de la SPL que el usuario desea implementar, el formato del archivo debe cumplir con lo definido por XMI.
- **Almacenar reglas**: el sistema debe almacenar las reglas lógicas y de selección que permiten implementar el mecanismo de derivación para el manejo de la variabilidad de las SPL.
- **Almacenar decisiones**: la aplicación debe almacenar las decisiones tomadas por el usuario respecto a las características de la línea de productos que desea desarrollar.

5.2.2. Requisitos funcionales

Son los requisitos de tipo funcional que deberá satisfacer el sistema y que responderán a las preguntas ¿qué debe hacer el sistema?, o ¿qué debe permitir el sistema hacer a los usuarios con la información almacenada?

1. **Leer diagrama almacenado**: la aplicación debe ser capaz de leer el diagrama almacenado y exportar la información que contiene a un archivo en formato XMI. Dicho diagrama corresponde con el nivel de abstracción de un PIM en MDA.
2. **Capturar las decisiones del usuario**: la aplicación debe permitir capturar las decisiones tomadas por el usuario sobre el núcleo de la línea de productos y la variabilidad de la misma.
3. **Procesar la información introducida por el usuario**: de acuerdo con la información introducida por el usuario, el sistema debe seleccionar las reglas lógicas a aplicar de aquellas almacenadas en su base de datos.
4. **Procesar en conjunto la información almacenada y las decisiones tomadas por el usuario**: la aplicación debe procesar el archivo XMI, que contiene la información del diagrama seleccionado por el usuario, las reglas lógicas y de selección almacenadas en la base de datos, y la información introducida en el requisito anterior, con la finalidad de manejar la variabilidad de la SPL.
5. **Verificar y seleccionar las posibles soluciones**: el sistema debe ser capaz de encontrar la solución o soluciones adecuadas para la línea de productos en función del procesamiento de información realizado en el requisito anterior y la solución de posibles conflictos entre los diversos productos a través de la implementación del mecanismo de derivación.
6. **Crear diagrama de clases**: posterior al proceso de derivación realizado en el requisito anterior (5), la aplicación crea un archivo XMI, debe contener la información necesaria para construir diagrama de clases detallados y datos sobre la línea de productos para la cual el usuario tomó decisiones en el requisito (2). Este archivo se encuentra en el mismo nivel de abstracción del PIM, de hecho, constituye una versión detallada del PIM del primer requisito, es decir, debe implementar una transformación horizontal.

5.2.3. Requisitos no funcionales

Son los requisitos de tipo no funcional, están relacionados, normalmente, con aspectos de tipo técnico tales como: desempeño, seguridad, tiempos de respuesta, interfaz gráfica, tecnologías de implementación, entre otros.

- **Independencia de plataforma**: el sistema debe ser independiente de la tecnología de implementación, concepto fundamental a tener en cuenta en la aplicación del enfoque MDA.

- **Reutilización sistemática de activos**: el sistema debe facilitar el desarrollo de activos reutilizables de una forma sistemática, bajo el enfoque de las SPL, para no caer en una reutilización oportunista del software.
- **Posibilidad de migración**: relacionado directamente con la independencia de plataforma, el sistema debe permitir la posibilidad de ejecución sobre diferentes plataformas o arquitecturas, con un mínimo de modificaciones.
- **Escalabilidad**: el sistema debe poder manejar un crecimiento continuo de trabajo de una manera fluida o crecimiento del sistema (hasta cierto punto), sin pérdidas significativas en calidad, concepto fundamental relacionado con los dos enfoques de desarrollo de software que aborda el presente proyecto.
- **Rendimiento**: el sistema debe buscar reducir en la medida de lo posible, el consumo de memoria y el tiempo de procesamiento de información.
- **Capacidad**: el sistema debe ser capaz de procesar información correspondiente a múltiples posibles combinaciones de soluciones, de tal forma que sea posible implementar el mecanismo de derivación.

5.3. Definición de la arquitectura

Una vez realizada la captura de requisitos, esta sección define una arquitectura para dar soporte a las familias de sistemas basadas en un enfoque MDA, haciendo uso de estándares abiertos, herramientas de software libre, iniciativas nacientes en cuanto a estilos arquitectónicos y metodologías para la descripción de la arquitectura.

5.3.1. Selección del estilo arquitectónico

El estilo arquitectónico seleccionado para la definición de la arquitectura corresponde al propuesto por SPL con un enfoque MDA, es decir, sigue los lineamientos descritos en el capítulo 3 con una orientación hacia la construcción de modelos de alto nivel de abstracción definidos con el estándar de modelado UML y sometidos a transformaciones horizontales.

Los fundamentos económicos que proponen las SPL son la parte clave de este enfoque y también el factor diferenciador. Las SPL afirman que, para alcanzar el éxito con una línea de productos, es importante la integración de la técnica con la planeación orientada al mercado, esta estrategia es conocida como delimitación de la línea de productos y fue explicada en el numeral 3.3. del presente documento. De acuerdo a dicha estrategia los tres niveles descritos son aplicados a la definición de la arquitectura del presente proyecto.

5.3.1.1. Delimitación del portafolio del producto

La línea de productos seleccionada son los diagramas de clases, ella responde a la intención de trabajar siguiendo un enfoque MDA, para lo cual el modelado con UML es una de las opciones más adecuada (ver secciones 4.2, 4.4.5 y 5.4.1).

De los diferentes diagramas UML, el diagrama de clases fue seleccionado porque uno de los criterios de trabajo del presente proyecto es dejar las puertas abiertas a trabajos futuros que continúen con la labor realizada en éste y las clases cuentan con cinco características básicas que facilitan la implementación de transformaciones entre modelos y por ende la continuación en la aplicación del enfoque MDA para desarrollo de familias de productos. Ellas son: encapsulado de toda la información de un objeto (un objeto es un ejemplar de una clase),

modelado del entorno de estudio, formalización del análisis de conceptos, definición de una solución de diseño y facilitación de la construcción de activos software [83].

El rango de productos soportado por la línea abarca sistemas[36] telemáticos[37] de diversa índole, en particular, la validación de la arquitectura se realizará con videojuegos (ver capítulo 6. Caso de estudio)

5.3.1.2. Delimitación del dominio

Para identificar el área o áreas funcionales principales o el dominio o dominios relevantes a la línea de productos deben tenerse en cuenta las dos dimensiones propuestas en el capítulo 3, en la sección de fundamentos económicos. En el capítulo 6 del presente documento, en el numeral 6.3. es realizada una posible delimitación del dominio del modelo de entrada de la aplicación. Es un PIM de alto nivel de abstracción.

- **Dimensión de viabilidad**

El dominio del modelado UML es maduro y próximamente contará con 15 años de existencia; desde que OMG tomó las características principales de los lenguajes de modelado que existían para el año 1995 y estableció el estándar común que denominó Lenguaje Unificado de Modelado [15]. Está posicionado como el lenguaje de modelado estándar, es empleado en la mayoría de proyectos software; no los limita en cuanto a recursos, gracias a que sus conceptos y técnicas están a disposición del público general; tampoco introduce restricciones organizacionales debido a que es suficientemente flexible como para ser adecuado a las líneas de productos.

El dominio de las líneas de productos es maduro y ha sido explotado ampliamente por la industria (por ejemplo, por la industria de alimentos, la automotriz, la de productos de belleza, etc.), sin embargo, las líneas de productos software son un concepto que hace tan solo 5 años empezó a implementarse seriamente [25]; afortunadamente, cuenta ya con ejemplos muy exitosos [13], [33]; por ejemplo Nokia, que producía normalmente 5 productos nuevos por año, pero desde que implementaron SPL liberan al mercado de 20 a 25 productos nuevos por año; o Celsiustech[38], compañía que antes producía cerca de 55 familias de sistemas de navegación con 210 desarrolladores y que ahora, con tan solo 30 y la implementación de SPL, hace la misma cantidad y con mejor calidad. Cummins Inc. y sus sistemas de control diesel; National Reconnaissance Office/ Raytheon y su conjunto de herramientas de canales de control; Philips y sus sistemas de control hospitalario, son también ejemplos, entre otros, de prácticas exitosas de SPL.

- **Dimensión de comportamiento**

UML fue definido de forma tal que fuese capaz de especificar el diseño de sistemas generales, y no solo sistemas software, de ahí que su éxito pueda extenderse y de hecho, esta extendido en otros mercados, como el de la economía y procesos de negocios [86]. UML permite la implementación de mecanismos de variabilidad, lo cual a su vez permite la implementación y gestión de líneas de productos, la reutilización de activos y el desarrollo de nuevos activos.

---

[36] Entendiendo por sistema, una parte del mundo real que una persona o grupo de personas, durante algún intervalo de tiempo y para algún propósito, escogen para verlo como un todo, consistiendo de componentes inter-relacionados, cada componente caracterizado por propiedades que son seleccionadas como relevantes para el propósito [84].

[37] Entendiendo por sistema telemático un sistema de aplicación en el ámbito de las telecomunicaciones, que utiliza computadores con el fin de mejorar la calidad de los servicios prestados o proveer nuevos servicios [85].

[38] Celsius Technologies es una compañía que trabaja estratégicamente con Tecnologías de la Información y organización de mercados, para proporcionar estrategias de negocios y soporte a proyectos (desde el comienzo hasta el final de los mismos) con un máximo rendimiento y efectividad.

Las SPL tienen todo un campo abierto en cuanto a mercados potenciales, actualmente algunas prácticas exitosas son conocidas [13], [33]. Ellas están en los campos de la telefonía móvil, sistemas de control diesel, sistemas médicos y gestión de redes. En realidad este enfoque puede ser aplicado a cualquier sistema software y resulta especialmente útil en sistemas telemáticos, ya que permite mejorar la calidad de los servicios prestados o proveer nuevos servicios en poco tiempo.

5.3.1.3. Delimitación de los activos

Pretende definir las funcionalidades específicas que los componentes reutilizables deberían soportar. Para el presente proyecto este nivel no aplica porque los activos son definidos en función de la línea de productos a implementar y la arquitectura propuesta no hace referencia a una SPL en específico, sino a cualquier familia de productos relacionada con sistemas telemáticos (de ahí que se implementen diagramas de clases, puesto que permiten trabajar en términos generales con cualquier SPL). Una aplicación de este nivel es desarrollado en el capítulo siguiente, caso de estudio.

5.3.2. Metodología para la descripción de la arquitectura

Para facilitar el entendimiento de la arquitectura propuesta, la metodología para su descripción está basada en el modelo definido por Rational Software Corporation [86] conocido como RUP (Rational Unified Process). Está basado en el uso de múltiples vistas concurrentes para abordar los intereses de los distintos actores que participan en la definición de la arquitectura, ellos son: administradores de proyecto, ingenieros de sistemas, usuarios finales, desarrolladores, etc. y manejar por separado los requisitos funcionales y no funcionales.

El objetivo perseguido por esta metodología [86] es describir las decisiones arquitectónicas en cuatro vistas, para después ilustrar estas decisiones con un conjunto reducido de casos de uso o escenarios, ellos constituyen la quinta vista, que será desarrollada y aplicada al caso de estudio del presente proyecto. Para cada vista un conjunto de elementos (componentes, contenedores y conectores) son definidos, al igual que la forma y los patrones con que trabajan, la justificación y las restricciones, relacionando la arquitectura con algunos de sus requisitos. Cada vista es descrita en un diagrama que tiene una notación particular, sin embargo, también es posible utilizar los estilos de arquitectura que sean considerados más apropiados para cada vista y, por lo tanto, hacer que coexistan distintos estilos en un mismo sistema. El modelo de 4+1 vistas es bastante genérico y puede ser usado con otras notaciones, métodos de diseño y herramientas, diferentes a las descritas por este modelo.

Aunque las vistas reciben diferentes nombres de acuerdo al autor sobre el cual este basada su descripción, en general todas describen las mismas características de la arquitectura, así por ejemplo, la vista lógica, también se conoce bajo el nombre de vista del modelo de referencia o vista de análisis. Para la elaboración del presente documento, los nombres propuestos en el artículo de Philippe Kruchten fueron asignados a las vistas [86] (uno de los autores del modelo de 4+1 vistas) que presenta un modelo para describir la arquitectura de sistemas software.

Es importante notar que no toda arquitectura software requiere las 5 (“4+1”) vistas completas, las vistas que no son útiles pueden omitirse de la descripción de la arquitectura o aquellas que son tan similares que no requieren descripciones independientes. La única vista que si es útil en cualquier circunstancia, es la vista de la funcionalidad [86]. En el presente documento no se describió la arquitectura desde el punto de vista de la distribución física de elementos, porque

carece de sentido implementar una vista que no ofrece nuevas formas de entender la arquitectura propuesta y que por lo tanto resulta inútil en la labor de descripción.

Por último, cabe agregar que el presente trabajo de grado optó por un desarrollo iterativo, donde la arquitectura es propuesta, probada, medida, analizada y refinada en sucesivas iteraciones. Esta forma de desarrollo permite diseñar prototipos evolutivos de la arquitectura, lo cual resulta muy adecuado para su definición en líneas de productos (ver numeral 3.5.), gracias a que permite que la arquitectura crezca y evolucione, antes que convertirla en un prototipo desechable y exploratorio. Este enfoque iterativo también permite refinar los requisitos, madurarlos y comprenderlos, más profundamente [86].

5.3.2.1. Vista lógica

Apoya principalmente a los requisitos funcionales y está determinada por la arquitectura lógica, la cual es capturada por los diagramas de clases. El sistema es descompuesto en una serie de abstracciones clave, tomadas, principalmente, del dominio del problema en la forma de objetos o clases de objetos, por lo cual el estilo es de orientación a objetos. Aquí los principios de abstracción, encapsulamiento y herencia son aplicados. Esta descomposición no sólo es hecha para potenciar el análisis funcional, sino que también sirve para identificar mecanismos y elementos de diseño comunes, a diversas partes del sistema [86], [87], lo cual es adaptado perfectamente al concepto de núcleo común y variabilidad de SPL.

A continuación son mostrados los diagramas de clases de los paquetes XMI y plugin (los paquetes más representativos de la aplicación construida) en las figuras 13 y 14. Los diagramas de los otros paquetes son mostrados en el anexo A de este trabajo. La división del diagrama general de la aplicación, en varios diagramas, según los paquetes, pretende simplificar el entendimiento del mismo. El diagrama de paquetes sobre el cual está basado la división del diagrama de clases es mostrado en la vista de desarrollo.

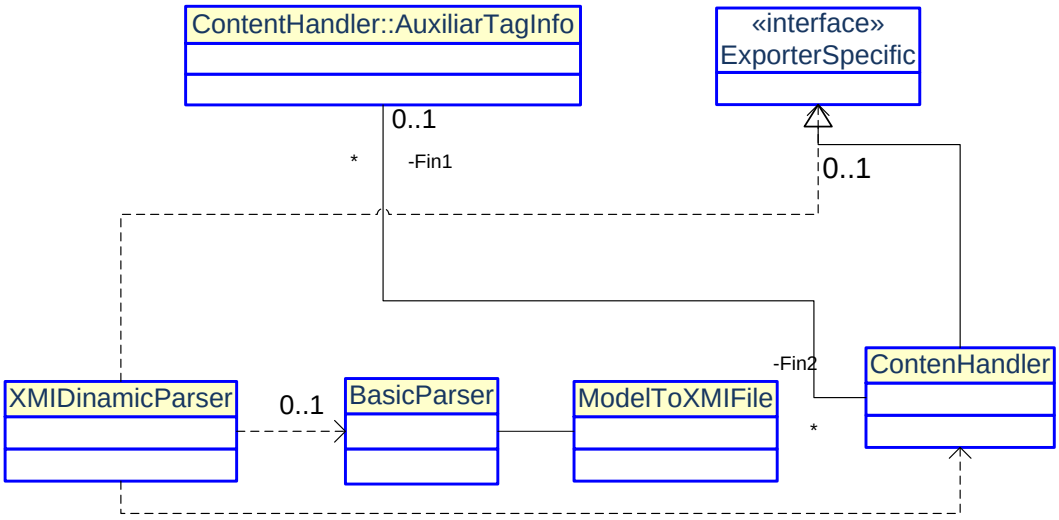

Figura 13. Diagrama de clases del paquete XMI, vista lógica

Figura 14. Diagrama de clases del paquete plugin, vista lógica

5.3.2.2. Vista de procesos

Está centrada en la estructura de los componentes y tiene en cuenta requerimientos no funcionales tales como: rendimiento, fiabilidad, factores de escala, integridad, seguridad, sincronización y administración del sistema [86], [87].

Para la descripción de esta vista son utilizados dos tipos de diagramas UML, el diagrama de análisis y el de secuencia (ver figuras 15, 16, 17 y 18). Al igual que en la vista lógica, donde los diagramas de clases fueron planteados en función de los paquetes, en esta vista los diagramas también son definidos para los paquetes XMI y plugin.

El diagrama de secuencia fue seleccionado porque es uno de los diagramas más efectivos para modelar la interacción entre objetos en un sistema, a través del tiempo y en este documento serán modelados los métodos principales de las clases de la aplicación. Al describir la vista de proceso por medio de estos diagramas los trabajadores internos de negocio y la información que ellos usan son definidos (entidades de negocio), su organización estructural en unidades independientes (sistema de negocio) y su interacción es descrita y proporciona una vista integral del comportamiento del sistema, es decir, muestra el flujo de control a través de varios objetos [14], [63].

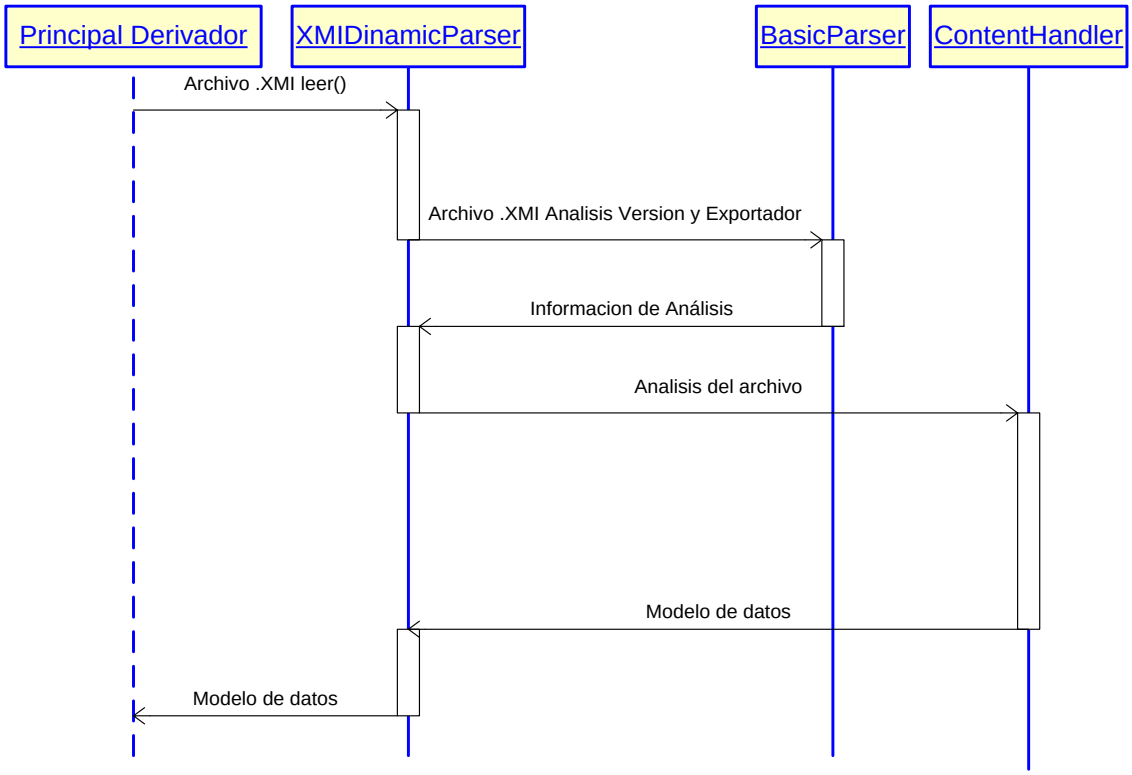

Figura 15. Diagrama de secuencia del paquete XMI, vista de procesos

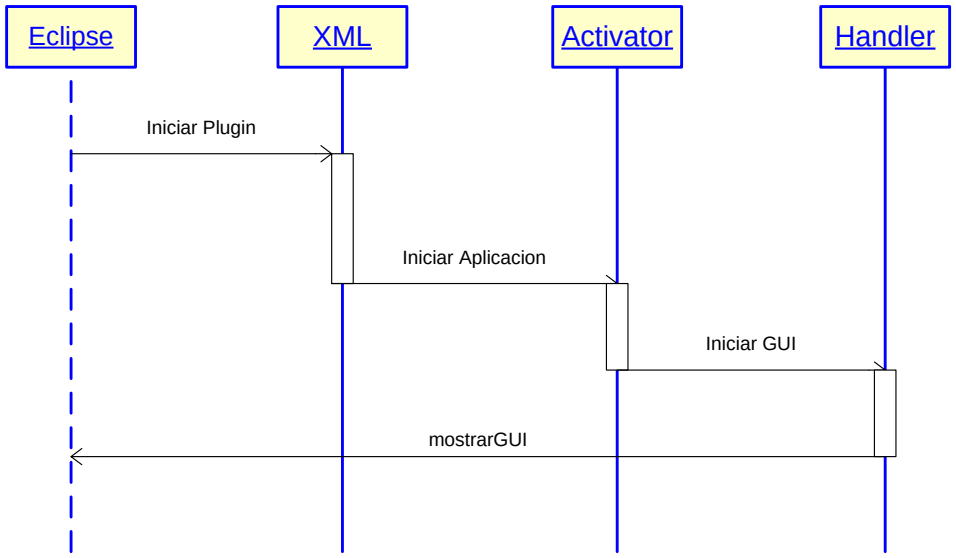

Figura 16. Diagrama de secuencia del paquete plugin, vista de procesos

El diagrama de análisis es un diagrama de actividad simplificado, que se decidió desarrollar porque permite capturar procesos del negocio de alto nivel y modelos tempranos del comportamiento y de los elementos del sistema; y aunque es menos formal que algunos otros diagramas, proporciona buenos medios para capturar las características y las necesidades esenciales del negocio [14], [63].

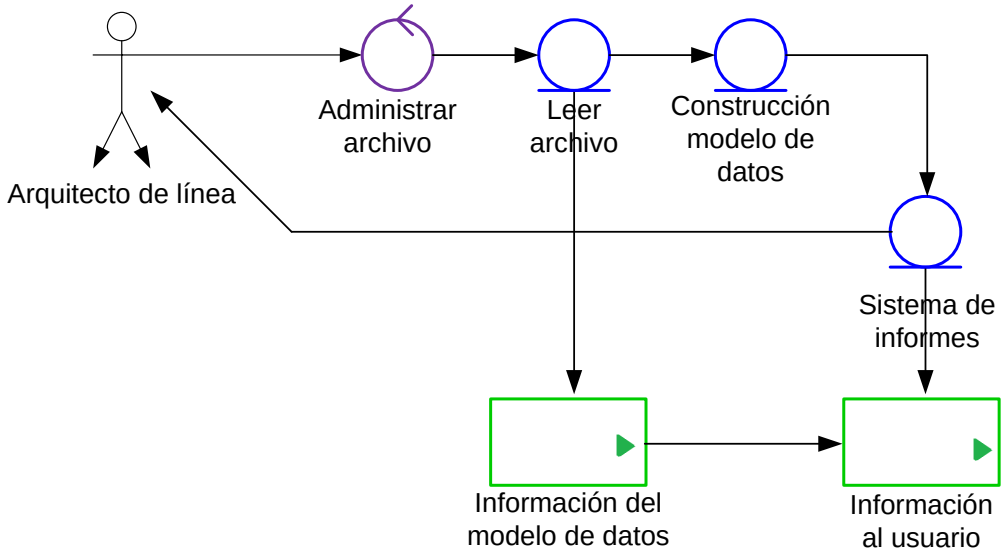

Figura 17. Diagrama de análisis del paquete XMI, vista de procesos

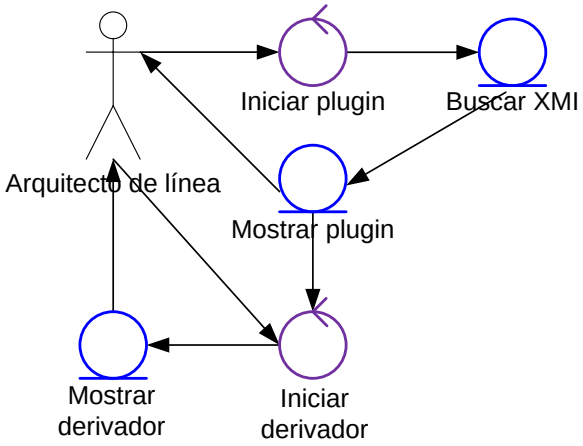

Figura 18. Diagrama de análisis del paquete plugin, vista de procesos

5.3.2.2. Vista de desarrollo

Refleja la organización de módulos de software dentro del entorno de desarrollo. Esta vista de la arquitectura toma en cuenta los requerimientos que facilitan la programación, los niveles de reutilización y las limitaciones impuestas por el entorno de desarrollo. Están a disposición dos elementos para modelar esta vista, los paquetes que representan una partición física del sistema (división del sistema en agrupaciones lógicas y las dependencias entre esas agrupaciones) y los componentes que representan la organización de módulos de código fuente [86], [87].

El diagrama de paquetes trata de organizar los paquetes de forma que maximice la coherencia interna dentro de cada paquete y minimice el acoplamiento externo entre éstos [89], situación que es adecuada al concepto de línea de productos, donde es buscado que el acoplamiento con otros dominios sea muy poco o nulo, porque de lo contrario es muy difícil desarrollar activos reutilizables. Por lo expuesto anteriormente, el presente documento describirá esta vista a través del diagrama de paquetes, en la figura 19.

Capa de presentación

- GUI: representa la interfaz gráfica en donde el usuario importa el archivo XMI e inserta las reglas lógicas y de selección a aplicar a la SPL de alto nivel de abstracción. Da inicio al proceso de búsqueda de las soluciones posibles de la SPL detallada.

Capa lógica

- XMI: se compone de todas las clases e interfaces que realizan la lectura y escritura de archivos de tipo XMI.
- Estructura de Datos: agrupa el modelo de datos representativo de la SPL de alto nivel de abstracción.
- Filtro: contiene el sistema de restricciones, filtros y el proceso de derivación que deben ser aplicados sobre la SPL de alto nivel de abstracción.
- Lógica: representa la mediación entre el plugin y la base de datos. Agrupa un conjunto de clases auxiliares que facilitan la creación de filtros de tipo lógico, haciendo el respectivo análisis de sentencias proposicionales o la construcción de tablas de verdad.

Capa de Datos

- Base de Datos: representa el servidor de base de datos en donde son almacenadas las reglas lógicas y la tabla de verdad.

- MySQL: representa la base de datos en donde se almacenan las reglas lógicas y la tabla de verdad.

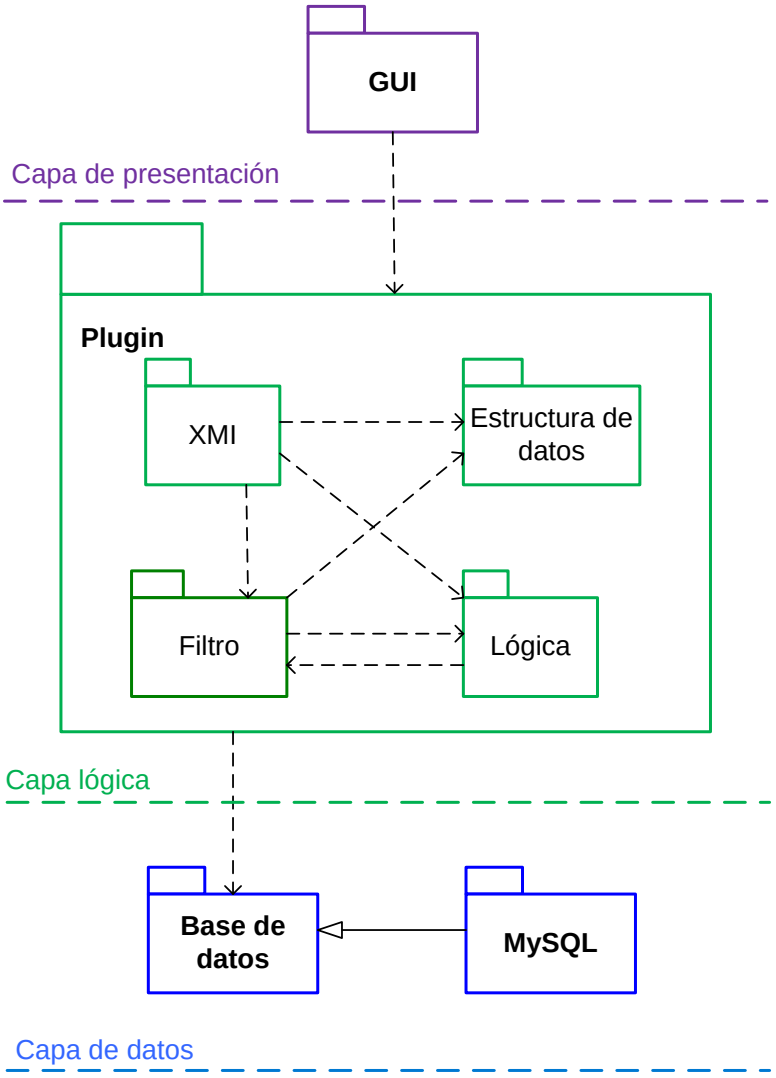

Figura 19. Diagrama de paquetes, vista de desarrollo

5.3.2.4. Vista de la funcionalidad

La vista de la funcionalidad es redundante a las otras vistas (y por lo tanto "+1") y sirve a dos propósitos principales, como guía para descubrir elementos arquitectónicos durante el diseño de la arquitectura, y como validación y verificación, después de completar el diseño de la arquitectura, al trabajar como punto de partida de las pruebas, para un prototipo de la arquitectura. Los elementos de las cuatro vistas trabajan conjuntamente en forma natural, mediante el uso de un conjunto pequeño de escenarios relevantes o instancias de casos de uso más generales, secuencias de interacciones entre objetos y entre procesos [86]. Los escenarios son de alguna manera una abstracción de los requisitos más importantes [88].

Esta vista certifica la validez de la vista lógica, de procesos y de desarrollo. Para definir la vista de la funcionalidad es posible utilizar los diagramas de casos de uso, su especificación, los diagramas de interacción (escenarios), los diagramas de actividad (flujos de trabajo) o los diagramas de estados [87].

Casos de uso

Para la descripción de esta vista fue seleccionado el diagrama de casos de uso ya que permite describir lo que el sistema debe hacer (análisis) desde el punto de vista del usuario, es decir, describe su uso y cómo interactúa con el usuario, lo cual ayuda a validar la arquitectura propuesta en el presente trabajo de grado y a verificar el caso de estudio durante el transcurso de desarrollo de éste [15], [63]. La notación gráfica (el diagrama de casos de uso) y la descripción sencilla de los casos de uso está consignada en esta sección del documento, siguiendo la plantilla para descripción de casos de uso del sistema propuesta en el capítulo 2, Estudio de Prefactibilidad, de la guía "Referencia Metodológica Integral para Desarrollo de Sistemas Telemáticos" del Grupo en Ingeniería Telemática de la Universidad del Cauca [90]. La documentación detallada de los casos de uso está en el anexo A, para su realización fue utilizada la plantilla para descripción de casos de uso propuesta, capítulo 3 "Formulación del Proyecto", en la misma guía.

Actores del sistema

- Arquitecto de línea: representa al usuario que hace uso del sistema para construir un diagrama UML con detalles específicos a una línea de productos en particular que desea implementar.
- Arquitecto de productos: representa al sistema que realiza el procesamiento de la información almacenada y la información introducida por el arquitecto de línea, para obtener productos de la SPL que se desean implementar.

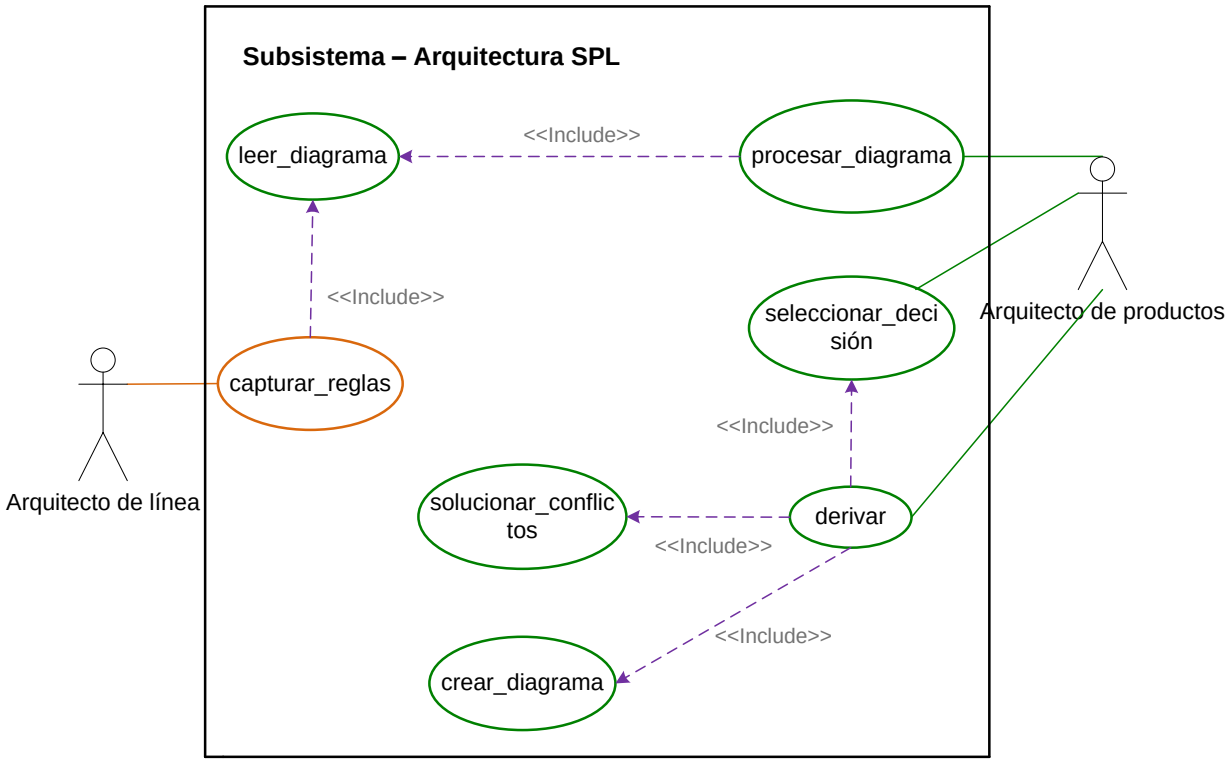

Figura 20. Diagrama casos de uso, vista de la funcionalidad

Descripción casos de uso

En las tablas 1 al 7 está la descripción inicial de los casos de uso identificados.

Caso de uso 1	Leer diagrama
Iniciador	Arquitecto de productos
Propósito	Leer la información que contiene el diagrama UML (PIM)
Resumen	El usuario ingresa al plugin, abre el archivo que contiene el diagrama UML de su preferencia, el diagrama es importado en un archivo XMI desde la herramienta en la cual fue diseñado (para el caso de estudio de este trabajo de grado, eUML2) para leer la información que contiene

Tabla 1. Descripción caso de uso 1 "leer diagrama"

Caso de uso 2	Capturar reglas
Iniciador	Arquitecto de línea
Propósito	Capturar las reglas (decisiones) que el usuario selecciona sobre la SPL que desea implementar
Resumen	El usuario selecciona las reglas que desea aplicar para la construcción de su SPL, el plugin captura las reglas seleccionadas y las almacena en la base de datos

Tabla 2. Descripción caso de uso 2 "capturar reglas"

Caso de uso 3	Procesar diagrama
Iniciador	Arquitecto de productos
Propósito	Procesar la información leída del diagrama seleccionado y de las reglas capturadas
Resumen	El usuario aplica un conjunto de reglas lógicas y de selección al diagrama seleccionado, de acuerdo a las reglas establecidas, incorporando información acerca de la variabilidad de la SPL

Tabla 3. Descripción caso de uso 3 "procesar diagrama"

Caso de uso 4	Seleccionar decisión
Iniciador	Arquitecto de productos
Propósito	Incorporar información sobre el núcleo común y variabilidad de la SPL
Resumen	El usuario toma decisiones, seleccionando algunas de las opciones posibles para incorporar información a la SPL que desea implementar, sobre el núcleo común y su variabilidad (por ejemplo, métodos y clases)

Tabla 4. Descripción caso de uso 4 "seleccionar decisión"

Caso de uso 5	Solucionar conflictos
Iniciador	Arquitecto de productos
Propósito	Obtener un conjunto reducido de soluciones posibles para la SPL
Resumen	Aplicar filtros de restricciones software para eliminar conflictos entre posibles soluciones, obteniendo un conjunto reducido o restringido de soluciones para la SPL

Tabla 5. Descripción caso de uso 5 "solucionar conflictos"

Caso de uso 6	Derivar
Iniciador	Arquitecto de productos
Propósito	Obtener un producto o ejemplar de producto de todos los posibles que permite el diseño de la SPL
Resumen	El usuario ejemplifica unos o más productos de todos los posibles que permite el diseño de la SPL, eliminando las opciones (productos) que no aplican a la línea y obteniendo un conjunto de soluciones válidas o conjunto final de soluciones, que puede estar vacío, contener tan solo un elemento (producto) o varios elementos para la SPL.

Tabla 6. Descripción caso de uso 2 "derivar"

Caso de uso 7	Crear diagrama

Iniciador	Arquitecto de productos
Propósito	Crear un diagrama que contenga información específica a la SPL a implementar (PIM detallado)
Resumen	La información resultante del caso de uso derivar es guardada en un archivo XMI y sus diagramas UML; ellos reflejan la información guardada en el mismo.

Tabla 7. Descripción caso de uso 2 “crear diagrama”

5.4. Selección de estándares y soluciones software

En el capítulo 4 fueron consignados los resultados principales obtenidos de una exploración realizada sobre los diferentes estándares y soluciones software relacionados con el enfoque MDA. Esta sección determina cuáles estándares y herramientas emplear, teniendo en cuenta los objetivos que persigue el presente trabajo de grado, la descripción de la arquitectura realizada en la sección anterior y el caso de estudio a implementar.

5.4.1. Lenguaje de modelado unificado

El lenguaje estándar de modelado seleccionado para el proyecto es UML, descrito brevemente en la sección 4.4.5. La decisión resulta evidente porque para poder emplear los conceptos de MDA en el desarrollo de software, es necesario entender en primer lugar UML [16], [52].

Su elección facilita aplicar el enfoque MDA de dos formas diferentes; primero, permite crear modelos de software de sistemas que serán construidos, partiendo del hecho que se sabe cómo y dónde aplicar el lenguaje UML, de tal forma que los modelos desarrollados son lo suficientemente precisos y consistentes como para ser utilizados dentro de MDA; y, segundo, permite definir transformaciones entre modelos de diferentes sistemas [52].

UML también facilita aplicar el enfoque de líneas de productos, porque a través de la información que contienen los diagramas de clases es posible implementar una arquitectura que permita el desarrollo de SPL para sistemas telemáticos.

UML no es una herramienta para crear sistemas software, ni un proceso descriptivo para crear sistemas software (esta tarea le correspondería a una metodología para definir arquitecturas, como 4+1, o SOA). Tampoco provee un método o proceso (como si lo hace RUP). Simplemente es un lenguaje visual para comunicar, modelar, especificar y definir sistemas, que gracias a que está diseñado para ser flexible, extensible, y comprensible, y al mismo tiempo ser lo suficientemente genérico como para servir a todas las necesidades del modelado de sistemas, es de gran utilidad para cumplir con los objetivos propuestos en el presente proyecto e implementar el caso de estudio [91]. Otra característica adicional por la cual UML resulta adecuado para este trabajo de grado es que los elementos del modelado de UML, es decir, clases, interfaces, casos de uso, diagramas de actividad, etc., pueden ser intercambiados entre herramientas diferentes utilizando XMI [92].

5.4.2. Lenguaje de marcas extensible

Es un lenguaje de marcado, similar a HTML utilizado para la definición de las páginas web, pero más moderno y evolucionado. Fue desarrollado para hacer frente a los nuevos retos que suponía la publicación de información electrónica a gran escala, fue introducido en 1998 por el World Wide Web Consortium (W3) responsable de la mayoría de las normas y estándares de Internet. Gracias a sus características de potencia y flexibilidad, en muy pocos años ha ganado

reconocimiento mundial y ha experimentado un gran auge, situándose como un elemento característicos tanto de proyectos completamente nuevos, como de las últimas versiones de las aplicaciones más clásicas [15].

Es un lenguaje estándar para estructurar información de forma completamente universal y abierta; no está enfocado a ningún objetivo particular, es flexible en el sentido de poder ser utilizado con cualquier tipo de objetivo. Está basado en etiquetas que encierran la información clasificándola. El anidamiento de etiquetas es una función fundamental del lenguaje. En cierto sentido las etiquetas son información acerca de la información, los conceptos del significado de dichas etiquetas, residen en las aplicaciones externas, no en el propio archivo XML; es decir, la lógica de comprensión y uso de la información debe ser implementada por las aplicaciones para cada forma que ésta adopte, adecuándose perfectamente al enfoque MDA. XML no es un lenguaje de etiquetas en sí mismo, como puede ser HTML, en el que las etiquetas válidas son fijas y conocidas, sino un metalenguaje, concepto que también se adecua muy bien al enfoque MDA, ya que cualquier etiqueta es válida y su significado y aplicabilidad es responsabilidad de los desarrolladores de la aplicación [15], [93].

XML es un estándar sencillo que tiene a su alrededor otros estándares que lo complementan y lo hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad y en el presente trabajo de grado, ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura y fiable, gracias a que la estructura de un documento creado en XML es fácil de entender y procesar [93]. Otra razón para trabajar con este estándar es que XML busca dar solución al problema de expresar información estructurada, de la manera más abstracta y reutilizable posible, dos características que facilitan la implementación de los dos enfoques de desarrollo de software sobre los cuales trata este proyecto; el nivel de abstracción está directamente relacionado con MDA y la reutilización con SPL [93].

5.4.3. Intercambio de Metadatos XML

Es un estándar para la definición, intercambio, manipulación e integración de objetos y datos XML, ha sufrido una rápida evolución. Su versión inicial, numerada 1.0, apareció en febrero de 1999, apenas un año después de la primera versión de XML. Como el resto de estándares de OMG ha evolucionado, basado en la publicación de borradores y RFP. Aunque en un principio la intención del estándar era la de permitir que aplicaciones heterogéneas intercambiasen objetos de forma homogénea; XMI es, también, una forma de almacenar e intercambiar diseños y diagramas arquitectónicos en UML; más aun desde la inclusión de XMI por parte de OMG dentro de la iniciativa MDA [15].

XMI resulta útil para el presente trabajo de grado porque facilita la tarea de intercambiar modelos software, gracias a que el estándar UML puede traducirse de forma casi perfecta sobre el estándar MOF. XMI es el lenguaje para describir estructuras MOF mediante XML. De esta forma es posible describir la información estructural de los diagramas en un archivo XMI, en principio compatible con todas aquellas herramientas que siguen el estándar. Lo que esto permite es tomar el modelo creado en determinada herramienta e importarlo o exportarlo a otra herramienta, situación que ayuda a UML a alcanzar su meta de ser un estándar para la comunicación entre diseñadores [15], [94]. En las figuras 21 y 22 es posible observar el efecto que tiene XMI en el intercambio de modelos.

XMI define la forma de utilización de las etiquetas XML empleadas para representar modelos MOF serializados en XML, de tal forma que pueda traducir MOF a XML (ver figura 23). XMI resuelve problemas relacionados con utilizar un lenguaje basado en etiquetas para representar

objetos y sus asociaciones, además, el hecho de que XMI esté basado en XML, significa que tanto los metadatos (etiquetas) y los ejemplares que describen (elementos), pueden ser agrupados en el mismo documento, permitiendo a las aplicaciones entenderlas rápidamente por medio de los metadatos [92], [94].

XMI no solo puede ser usado como un formato de intercambio UML, sino que puede ser utilizado para cualquier formato descrito por medio de un metamodelo MOF, por ejemplo: CWM. Las herramientas compatibles con MOF permiten definir metamodelos o importarlos, de repositorios y herramientas CASE, y editar o modificar la información del modelo. Una vez hecho esto, la información del modelo podría ser intercambiada con otra herramienta compatible con MOF por medio de XMI [92].

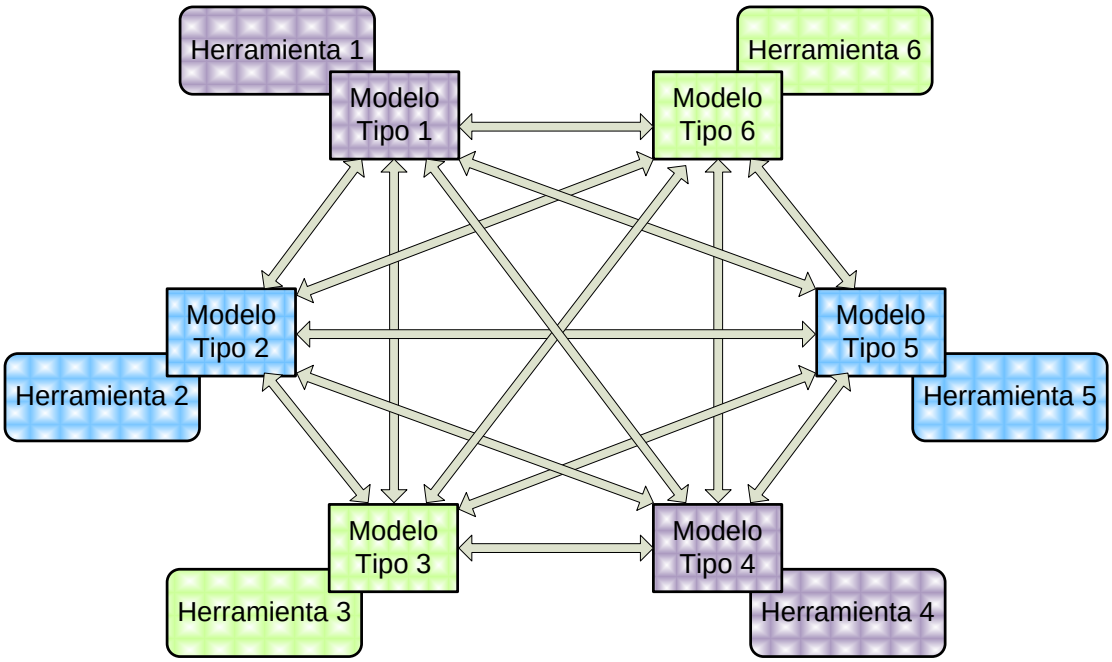

Figura 21. Intercambio de modelos entre herramientas sin XMI

XMI define muchos de los aspectos importantes involucrados en la descripción de objetos en XML [92], [94]:

- La representación de objetos en términos de elementos y atributos XML.
- Como los objetos están normalmente interconectados, XMI incluye un mecanismo estándar para enlazar objetos dentro del mismo archivo o entre archivo diferentes.
- La identidad de los objetos, permite que estos sean referenciados por otros objetos.
- El versionado de objetos y sus definiciones son gestionadas por el modelo XMI.
- La validación de documentos XMI por medio de DTD[39] (Document Type Definition) o XMI Schemas.

[39] DTD es una descripción de estructura y sintaxis de un documento XML o SGML(Standard Generalized Markup Language, es el estándar universal para la escritura de archivos de hipertexto en la Internet, basado en la separación de las imágenes del texto), cuya función básica es la descripción del formato de datos, para usar un formato común y mantener la consistencia entre todos los documentos que utilicen la misma DTD. De esta forma, dichos documentos, pueden ser validados, conocen la estructura de los elementos y la descripción de los datos que trae consigo cada documento, y pueden además compartir la misma descripción y forma de validación dentro de un grupo de trabajo que usa el mismo tipo de información [95].

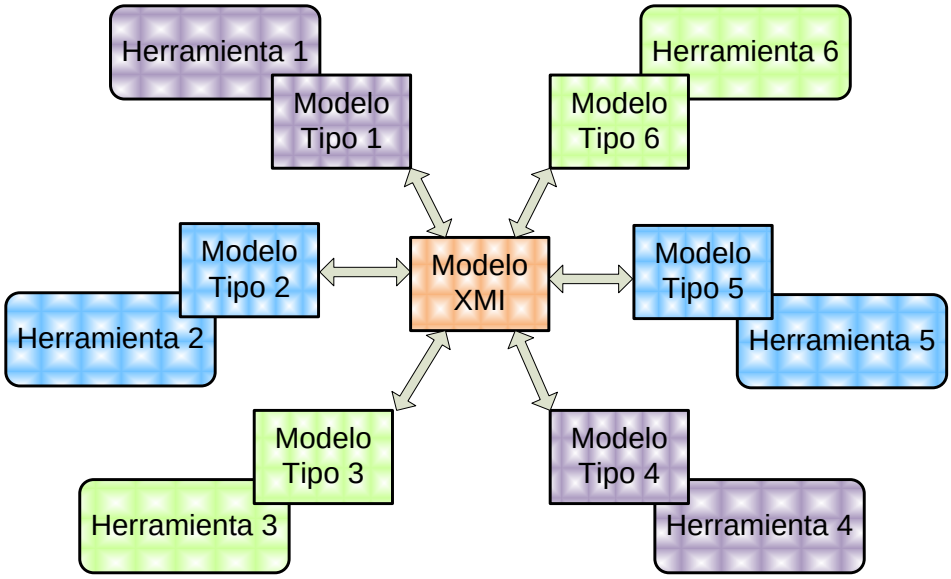

Figura 22. Intercambio de modelos entre herramientas con XMI

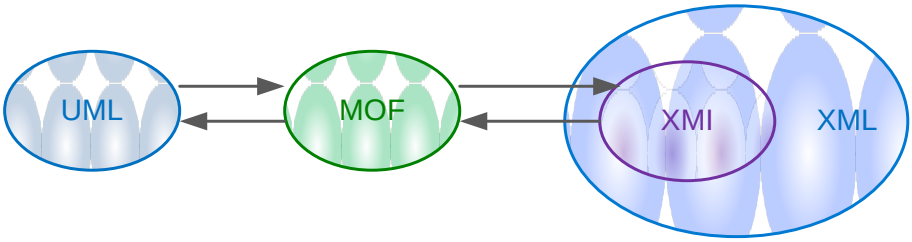

Figura 23. Relación entre UML, MOF, XML y XMI

Antes de UML 2.0 el archivo XMI no incluía información sobre la representación gráfica de los modelos, así que el diseño del diagrama no era mantenido. Bajo estas condiciones fueron desarrollados dos trabajos anteriores al presente [14], [15], es decir, para el año 2003, XMI no abordaba este aspecto tan importante del modelado UML, como es la información gráfica de los elementos, solo la información básica del diagrama era soportada en el estándar y cada implementación debía hacer uso de las extensiones del lenguaje, para proveerse de un método para guardar y recuperar la información gráfica de los modelos. Esta situación llevaba a que cada equipo de desarrolladores adoptara y modificara ligeramente el estándar, añadiendo características extra que impedían la compatibilidad cruzada, dando lugar a diferentes "dialectos" de XMI [15]. Sin embargo, actualmente esto ya no es un problema y para el desarrollo del presente trabajo de grado se cuenta con UML 2.1. y XMI 2.0.

5.4.4. Entorno de Desarrollo Integrado

El IDE de trabajo seleccionado es Eclipse, descrito brevemente en el numeral 4.5.12. Esta elección se toma porque Eclipse tiene características deseables y muy apropiadas para la implementación de los dos enfoques de desarrollo sobre los cuales está centrado el presente trabajo de grado [67]. Algunas de esas características son que proporciona un entorno abierto y multiplataforma sobre el cual construir cualquier tipo de aplicación, y en particular y con relación al presente trabajo, la construcción de herramientas para el desarrollo de software. Su naturaleza libre ha propiciado el surgimiento de muchos proyectos que abarcan múltiples

dominios tecnológicos, entre los que está MDD, directamente relacionado con MDA; además, resulta útil, para el alcance de los objetivos planteados para este proyecto, gracias a que la amplia oferta de tecnologías de libre disposición que ofrece actualmente aumenta la posibilidad de encontrar una herramienta ajustada a las necesidades particulares del proyecto, aun cuando esto signifique invertir mayor cantidad de tiempo en encontrarla, pero que cuando sea encontrada esta sea la más idónea.

A parte de lo ya mencionado, hay dos características básicas más de Eclipse resaltadas por [65] que lo hacen un IDE adecuado para el desarrollo del caso de estudio de la arquitectura propuesta en el presente proyecto de grado. Primero, facilita el modelado al permitir la integración de datos y almacenamiento de metamodelos y metadatos y segundo, su arquitectura está diseñada de forma que la mayoría de la funcionalidad proporcionada está localizada en componentes enchufables o en un conjunto de ellos relacionados (pueden crearse y adicionarse), haciendo de Eclipse una herramienta extensible.

La versión de Eclipse empleada para el desarrollo es Eclipse 3.4, llamada Ganymede, la cual corresponde a la entrega anual de proyectos de Eclipse del año 2008 y que incluye 23 proyectos. Algunos de los proyectos más destacados son la nueva plataforma p2, nuevas características de seguridad de Equinox, nuevas herramientas de modelado Ecore y soporte para Arquitectura Orientada a Servicios o SOA[40] (Service Oriented Architecture), entre otras. Ganymede busca mejorar la productividad de los desarrolladores trabajando sobre ella y proporcionando un ciclo de desarrollo más transparente y predecible.

Desde el 2006 la fundación Eclipse ha coordinado una entrega en simultáneo que incluye la plataforma Eclipse así como también otros proyectos de Eclipse. En apariencia, esto simplifica el desarrollo y mantenimiento de sistemas empresariales, además resulta conveniente. Hasta ahora, cada entrega simultánea ha tenido lugar al final del mes de junio (razón por la cual no se emplea Eclipse 3.5 o Galileo como IDE en el presente trabajo). Actualmente la meta de Ganymede es la misma meta del año 2007 con Europa, entregar N grandes proyectos de Eclipse al mismo tiempo, con el fin de eliminar la incertidumbre sobre los números de versiones de proyectos y de este modo permitir a los miembros de la comunidad empezar más pronto su propia integración entre proyectos, productos y pruebas.

5.4.5. eUML2

La herramienta de modelado escogida fue eUML para Eclipse, descrita en el numeral 4.5.13. Para el presente trabajo es empleada la versión libre de la herramienta (no puede ser utilizada con fines comerciales) que está a disposición del público general para descargas a través de la página http://www.soyatec.com/euml2/installation/. La selección de esta herramienta fue basada en las funcionalidades que ofrece, para la versión libre son las siguientes:

- Modelado UML: visualizar, diseñar, comunicar y documentar diagramas UML 2.1. con un ambiente de modelado intuitivo, visual y sofisticado.
- Interoperabilidad e integración: exportar, importar y agrupar con formato XMI 2.0. Lo que permite exportar e importar archivos XML para intercambios de datos con otras aplicaciones y permite exportar e importar archivos XMI 1.0, 1.2 y 2.0, lo cual a su vez permite almacenar modelos UML en formato XML.

[40] SOA es un concepto de arquitectura de software que define el uso de servicios para soportar los requerimientos de usuarios de software [67].

5.4.5. MySQL 5.0.

MySQL es una marca registrada de MySQL AB, compañía comercial, fundada por los desarrolladores de MySQL, pretende unir los valores y la metodología Open Source con un exitoso modelo de negocio; la compañía desarrolla, distribuye y soporta MySQL. El software MySQL tiene una doble licencia, es posible elegir utilizar el software MySQL como un producto Open Source bajo los términos de la licencia GNU (General Public License, http://www.fsf.org/licenses/), que es la versión utilizada en el presente proyecto, o adquirir una licencia comercial estándar de MySQL AB [96].

El software MySQL proporciona un servidor de base de datos SQL[41] (Structured Query Language) muy rápido, multihilos, multiusuario y robusto. El servidor MySQL está diseñado para entornos de producción críticos, con alta carga de trabajo, así como para integrarse en software para ser distribuido. Al ser un sistema de gestión de base de datos y bases de datos relacionales (MySQL Server), permite añadir, acceder, y procesar los datos almacenados en una base de datos, entendiendo por ella una colección estructurada de datos y por base de datos relacional, aquella que almacena datos en tablas separadas, para aumentar la velocidad y flexibilidad, en vez de almacenar todo en un gran almacén. Como el software de MySQL es de código abierto (software libre), su elección significa que es posible para cualquiera usarlo y modificarlo sin tener que pagar por ello, característica importante para el desarrollo del presente proyecto, de cara a trabajos futuros que implementen más niveles de abstracción de MDA [96].

Algunas de las principales características de MySQL 5.0 son: probado con un amplio rango de compiladores diferentes, existen API (Application Program Interface) disponibles para C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, y Tcl, puede usarse fácilmente sobre múltiples CPU (Central Processing Unit), es multiplataforma y funciona sobre Linux, Windows, Solaris, etc., proporciona sistemas de almacenamiento transaccionales y no transaccionales, es relativamente sencillo de añadir otro sistema de almacenamiento, cuenta con un sistema de reserva de memoria muy rápido basado en hilos, diversos tipos de columnas, registros de longitud fija y longitud variable, soporte completo para operadores y funciones en las cláusulas de consultas SELECT y WHERE, soporte para alias en tablas y columnas, puede mezclar tablas de distintas bases de datos en la misma consulta, un sistema de privilegios y contraseñas que es muy flexible y seguro, contraseñas seguras (porque todo el tráfico de contraseñas está cifrado cuando se conecta con un servidor), soporte a grandes bases de datos y a comandos SQL para chequear, optimizar y reparar tablas, además en todos los programas MySQL pueden invocarse las opciones “help o ?” para obtener asistencia en línea, [96].

Para la implementación del caso de estudio fueron buscadas soluciones de software libre, por las características que lo definen, por limitaciones de tipo económicas y porque el desarrollo de este proyecto persigue una meta académica. MySQL y PostgreSQL son las bases de datos libres más populares y más empleadas en todo el mundo; cualquiera de las dos podría haberse utilizado en este proyecto, sin embargo, considerando que para el desarrollo del mismo es una prioridad la velocidad, debido a la cantidad de información que es necesaria procesar (ver la sección de descripción del caso de estudio) y que en trabajos futuros, posiblemente aumentará, se eligió MySQL, ella está caracterizada en especial por esto y por la facilidad de uso.

[41] SQL o Lenguaje de consulta estructurado, es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en éstas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo efectuar consultas con el fin de recuperar de una forma sencilla información de interés de una base de datos, así como también hacer cambios sobre ella [98].

PostgreSQL es utilizado principalmente cuando se quiere migrar desde Oracle, Sybase, o Microsoft SQL Server (situación que no aplica al presente proyecto), mientras que MySQL es utilizado por lo general para el desarrollo Web y otro tipo de aplicaciones en las cuales la rapidez de respuesta y el bajo consumo de recursos es indispensable (situación que aplica al presente proyecto), es decir, para el proyecto de grado lo buscado es la velocidad antes que potencia.

Para consultar información sobre bases de datos propietarias, análisis comparativos entre bases de datos libres y propietarias, o entre las dos bases de datos libres mencionadas consultar el trabajo titulado "Análisis comparativo de bases de datos de código abierto vs. código cerrado (determinación de índices de comparación)" [97], documento en el cual se compara en concreto MySQL y Oracle. Consultar la página web http://www.kriptopolis.org /mysql-vs-access, el autor realiza una comparación entre MySQL y Access. En las páginas web http://www.javahispano.org/contenidos/es/mysql_vs_postgresql_cuando_emplear_cada_una_ de_ellas_11/ y http://www.guatewireless.org/mysql-vs-postgresql/ están comparaciones entre MySQL y PostgreSQL.

5.4.6. API Simple para XML

Es una interfaz simple para aplicaciones XML, fácil e intuitiva, de amplio uso entre programadores de Java. Es un analizador o "parser[42]" que sirve para estudiar el lenguaje XML, leer documentos XML, definir la estructura de un documento y comprobar si un documento es válido sintácticamente. El funcionamiento de SAX sigue una serie de pasos establecidos, primero comienza a leer el documento, luego detecta los eventos de análisis (como por ejemplo, comienzos o finales de un elemento), después la aplicación procesa esa parte leída y, por último, reutiliza la memoria y vuelve a leer hasta un nuevo evento; de tal forma que procesa el documento XML analizando la corriente de entrada XML y pasando los eventos a un método para operar con una programación definida.

En el caso de XML, como el formato siempre es el mismo, no es necesario crear un analizador cada vez con un programa, sino que existen un gran número de ellos, como SAX y el Modelo de Objetos del Documento o DOM (Document Object Model), que pueden verificar si un documento XML cumple con una determinada gramática. DOM y SAX son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc. SAX es usado en particular cuando los archivos XML ya están en una forma que es estructuralmente similar a la que se desea obtener, situación adecuada perfectamente al proceso de lectura de modelos a desarrollar en el presente proyecto, ya que busca implementar transformaciones horizontales de PIM a PIM. Otra razón por la cual SAX resulta conveniente dadas las características del trabajo de grado, para el proceso mencionado, es que cuando la información almacenada en los documentos XML, es decir, los datos, han sido generados por la máquina o son legibles para ella, SAX es la forma más directa de API para que los programas tengan acceso a dicha información. Aun cuando SAX requiere una mayor programación, su elección sobre DOM responde, a parte de lo ya dicho, a que puede ser muy útil cuando hay que tratar de rescatar un fragmento de un documento o buscar algunos elementos en particular; además de que es más versátil y más veloz que DOM, dos características fundamentales a tener en cuenta en el proceso de lectura. Aunque DOM es más potente, al ser más lento, afecta negativamente dichos procesos, mientras que SAX necesita menos código y menos memoria, lo que afecta positivamente ese proceso.

[42] Un parser XML es un módulo, biblioteca o programa que se ocupa de transformar un archivo de texto en una representación interna.

5.4.7. Modelo de Objetos del Documento

Es un modelo de objetos estandarizado para documentos HTML y XML [99]. Es un conjunto de interfaces para describir una estructura abstracta para un documento XML. Los programas que acceden a la estructura de un documento a través de la interfaz de DOM pueden insertarse arbitrariamente, borrarse y reordenar los nodos de un documento XML, es decir, con DOM es posible construir documentos, añadir, eliminar o modificar elementos y contenido, la estructura y el estilo o presentación de los documentos. Todas estas acciones son realizadas mediante llamadas a funciones y procedimientos que permiten acceder, cambiar, borrar o añadir nodos de información (datos o metadatos) de los documentos XML, lo cual son funcionalidades muy útiles para desarrollar los procesos de derivación y creación de modelos, a implementar con el presente proyecto. Además de esto, la elección de DOM también es debida a que uno de los objetivos más importantes que persigue su especificación es proporcionar una interfaz estándar de programación que pueda utilizarse en una amplia variedad de entornos y aplicaciones, situación adecuada perfectamente a la iniciativa MDA.

Al contrario de SAX, un analizador DOM opera con la corriente completa de entrada XML, es decir, lee todo el documento completo y devuelve un documento objeto (Document Object), construyendo un árbol en memoria que refleja toda su estructura. La aplicación recorre el árbol realizando su procesamiento, ya que el documento devuelto tiene un API que permite manipular el árbol (virtual) de objetos nodo. Éste representa la estructura de la entrada XML. Entonces, mientras DOM tiene acceso al documento completo y todos los elementos y atributos están disponibles a la vez, facilitando la implementación de los dos procesos mencionados en este numeral; en SAX sólo está disponible el elemento actual, facilitando la lectura de modelos.

CAPÍTULO 6.

CASO DE ESTUDIO

Este capítulo describe el entorno que da lugar al caso de estudio, el funcionamiento global de la aplicación, incluida una descripción a grandes rasgos de los conceptos tomados de [15] para el proceso de construcción de las reglas de derivación (reglas lógicas y de selección) y del proceso de derivación en sí; los criterios de evaluación considerados, las pruebas y los resultados obtenidos con el caso de estudio.

6.1. Descripción del entorno

Las líneas de productos software están inspiradas en los procesos de producción de sistemas físicos, es decir, en Líneas de Productos o PL (*Product Line*), las cuales desde la revolución industrial tiene una fuerte presencia en las economías de producción, en diferentes tipos de industria. En una línea de productos de automóviles, es posible observar como una misma cadena puede producir vehículos hasta con 100 diferencias entre sí. Al aplicar este concepto a los sistemas software, las SPL proponen un sistema de producción basado en la reutilización de componentes de un determinado dominio, que permitan construir una línea de producción capaz de generar multitud de variantes de un sistema o subsistema software [33].

Las primeras industrias en las cuales fue implantado este modelo organizativo de producción, estaban caracterizadas por desarrollar grandes proyectos de producción y de I+D, incluso hoy, en este tipo de empresas los beneficios son más inmediatos y notorios que en pequeñas empresas, en particular, porque el punto a partir del cual es posible obtener ganancias con la implementación de una SPL, es tres productos, como fue mencionado en la sección 3.2 (ver figura 4) y algunas pequeñas empresas no necesitan implementar más de un tipo de producto. En el caso de la industria de desarrollo software, la aparición de modelos de producción organizados según estos principios es relativamente reciente, sin embargo, la progresión está siguiendo la misma dinámica que en la industria tradicional; son las grandes empresas que cuentan con proyectos de desarrollo de mayor envergadura, las que más pronto y en mayor medida son beneficiadas de las ventajas brindadas por la implantación de modelos productivos basados en líneas de productos [13].

El caso de estudio a desarrollar en este proyecto toma una empresa de producción de videojuegos, debido a que cumple con las características ya mencionadas y es adaptado muy bien a los objetivos específicos de este trabajo de grado, constituyendo un buen ejemplo sobre las capacidades, ventajas y beneficios de seguir el enfoque MDA y SPL [15]. Un videojuego es un sistema completo de modelado y simulación de entornos de alguna actividad, imaginaria o real, desarrollado por lo general para entretenimiento de los usuarios. Este tipo de productos presentan una serie de características que los hacen idóneos para ejemplarizar las posibilidades que trae la implementación de una SPL bajo el enfoque MDA, ya que el desarrollo de estos productos lleva a plantear proyectos de larga duración y con costos de producción muy altos, situación que para algunas empresas de este sector, implica no poder trabajar más que en unos pocos proyectos simultáneamente (o tan solo en uno), y que el fracaso de uno solo de ellos puede ponerla en una delicada situación económica.

Algunas otras razones por las cuales una empresa grande, dedicada al desarrollo de videojuegos, resulta adecuada para la implementación del caso de estudio, son:

- Los video juegos tienen una parte hardware y otra software, situación adecuada perfectamente al concepto de PL y SPL, respectivamente. La parte hardware está encargada de proporcionar los medios de interacción físicos, visuales y sonoros; y la parte software (es la parte que compete a este trabajo de grado), encargada de simular el entorno de actividad (el comportamiento de un vehículo, el desarrollo normal de un partido de fútbol) y las reacciones del mismo y del entorno, ante las acciones del usuario.
- Cada proyecto es diferente a los anteriores, incluso en el caso de dos videojuegos del mismo tipo, tales como aviones de combate y carros de carreras; las diferencias entre proyectos son tan amplias que tradicionalmente la industria ha operado ejecutando cada proyecto de manera independiente, aprovechando la reutilización de elementos de manera tradicional (oportunista y no sistemática) y sólo en los puntos donde dicha reutilización fuese muy evidente.
- Cada proyecto comparte características comunes con todos los anteriores, debido a que todos los videojuegos tienen una arquitectura parecida y pueden compartir componentes claves de su funcionamiento general.

6.2. Descripción del caso de estudio

Como fue mencionado, la implementación del caso de estudio pretende abordar el desarrollo de una SPL bajo el enfoque MDA, centrando el trabajo en desarrollar una SPL que entregue PIM (diagramas de clases) que contengan información que facilite la construcción de videojuegos. De acuerdo con esto, de ahora en adelante este documento obviará la parte hardware de los videojuegos, la cual no concierne a los objetivos del presente proyecto y para efectos prácticos, el hablar de videojuego corresponde a la parte software del mismo.

El mayor reto enfrentado por el caso de estudio es intentar abordar el modelo de producción de las SPL, tratar con muchos productos diferentes al mismo tiempo, debido a que es difícil obtener los requerimientos correctos para todos los productos desde el principio, y por lo general el trabajo inicia con requerimientos incompletos y cambiantes. Para contrarrestar los efectos negativos de esta situación es necesario dar soporte a la variabilidad de la SPL a desarrollar [48], con tal fin, el diseño y gestión de variabilidad de una SPL es abordada mediante una aproximación a componentes funcionales, que permita diferenciar los elementos que son necesarios en todos los videojuegos, y que por tanto formarían parte de los activos comunes de la SPL; de los elementos que no son necesarios en todos los videojuegos, sino tan solo en algunos de ellos, y que por tanto serían elementos variantes de la línea, que aportan especificidad y particularización a las configuraciones derivadas que los incorporen.

La siguiente sección plantea un primer ejemplo de la representación de variabilidad de una SPL, limitada a los componentes obligatorios y opcionales descritos en el numeral 3.5.2.2, de tal forma que los componentes intercambiables son ignorados, con el único propósito de facilitar el entendimiento de este primer acercamiento hacia el desarrollo de una SPL, más adelante, los numerales 6.2.4 y 6.4 explican cómo implementar mecanismos de variabilidad que den lugar también a los componentes intercambiables..

6.2.1. Delimitación de los activos

En la sección delimitación de los activos, en el capítulo acerca de la arquitectura, fue mencionado que los activos son definidos en función de la SPL a implementar, ahora que ha sido definido que el desarrollo del caso de estudio está orientado hacia la construcción de PIM's para videojuegos es posible decir que los principales activos (componentes) que forman

parte de un videojuego son los listados en la tabla 8 [15]. Es evidente que la complejidad del videojuego afectará la complejidad de cada uno de estos activos, los cuales pueden estar formados por un solo elemento o por varios interconectados entre sí, cada uno con características de variabilidad internas, con elementos comunes a todas las posibles configuraciones y elementos que están presentes solo en un conjunto limitado de estas configuraciones.

Activo	Representación
Núcleo central de control	Necesario
Componentes de la lógica de simulación	Necesario
Sistema de grabación y reproducción	Opcional
Interfaces y protocolos de comunicación	Opcional
Sistema de representación visual	Necesario
Sistema de representación sonora	Opcional
Sistema de entradas y salidas	Necesario

Tabla 8. Componentes arquitectónicos de un videojuego

- **Núcleo central de control:** es el componente encargado de la coordinación de los demás componentes. Dirige el videojuego y es el encargado de hacer cumplir los requisitos de tiempo real del mismo.
- **Componentes de la lógica de simulación:** son los componentes encargados de modelar el comportamiento del entorno, las leyes físicas que rigen las relaciones de los elementos que lo componen, la "inteligencia artificial" de los mismos en los casos en que sea necesaria y el comportamiento del elemento simulado (un jugador de fútbol, un avión de combate, etc.)
- **Sistema de grabación y reproducción:** es el componente encargado de almacenar la información relativa a las ejecuciones del videojuego, de tal forma que pueda ser retomado más tarde, a partir del punto donde fue guardado por última vez y no desde el inicio.
- **Interfaces y protocolos de comunicación:** en un videojuego pueden haber varios jugadores participando al mismo tiempo, accediendo desde diversos dispositivos (celular, computador, consola, etc.) y diferentes medios (Internet, WiFi, GPRS, etc.) que tal vez no comparten el mismo software o arquitectura, y que constituyen, un sistema distribuido. Es necesario, entonces, un componente que permita una comunicación efectiva entre los diferentes jugadores, y por tanto, un conjunto de protocolos e interfaces que definan las posibilidades de dicha comunicación.
- **Sistema de representación visual:** es el componente encargado de la comunicación visual con el jugador, lo que generalmente abarca la creación y proyección de un modelo bidimensional que representa su entorno de ejecución.
- **Sistema de representación sonora:** es el componente encargado de la comunicación con el usuario mediante sonidos, lo que corresponde a la reproducción de un entorno sonoro acorde con las acciones del videojuego.
- **Sistema de entradas y salidas:** es el componente encargado de establecer la comunicación desde y hacia el usuario del videojuego. Recoge las acciones del jugador frente al entorno software del juego y modifica dicho entorno en función de los comandos recibidos, desde la lógica del videojuego.

6.2.2. Gestión de variabilidad de la arquitectura

La aplicación a desarrollar consistirá en una aplicación construida siguiendo el enfoque MDA y que deberá realizar una transformación horizontal de un PIM de alto nivel de abstracción a un

PIM más detallado, permitiendo la gestión de la variabilidad de diagramas de clases de una SPL de videojuegos de entrada, para obtener a la salida un diagrama de clases de un videojuego en específico. Dado que es muy importante que el diagrama de clases de entrada esté bien definido, puesto que si no se cuenta con un modelo completo, consistente y preciso, es inevitable la propagación de errores por parte de la aplicación a etapas posteriores; es sugerido que los usuarios de la aplicación que deseen crear su propio diagrama de clases inicial, sigan las recomendaciones propuestas en el artículo "Architectural Modelling in Product Family Context" [100]. En el anexo B están ilustrados mediante dos diagramas de estados, los procesos y pasos necesarios propuestos en el artículo sugerido, para realizar el modelado de una arquitectura de referencia, ambos diagramas están basadas en las mejores experiencias identificadas por los autores, en compañías pertenecientes al proyecto CAFÉ, que desarrollaron el concepto de SPL y que lograron resultados exitosos con su implementación.

La sección 5.3.1.1 delimitó el portafolio del producto a los diagramas de clases y explicó, también, el porqué de esta elección. En función de esto y de la descripción realizada de la aplicación fue definido, para el caso de estudio, el diagrama de clases que representa la aplicación. Consta de 6 clases y una interfaz (ver tabla 9), representativas de los componentes de la tabla 8.

Clase(C) / Interfaz(I)	Componente	Variable
Núcleo (C)	Núcleo central de control	False
Componentes (C)	Componentes de la lógica de simulación	False
GrabaRepro (C)	Sistema de grabación y reproducción	True
Comunica (I)	Interfaces y protocolos de comunicación	True
Visual (C)	Sistema de representación visual	False
Sonido (C)	Sistema de representación sonora	True
EntradaSalida (C)	Sistema de entradas y salidas	False

Tabla 9. Elementos del diagrama de la SPL para videojuegos

Una vez creados los diferentes modelos (diagramas de clases), la información de variabilidad debe especificarse para los componentes variables mediante el valor etiquetado "true" (verdadero), que indica a la aplicación que el elemento al que acompaña, puede no aparecer en las derivaciones; para el caso contrario, el valor de la etiqueta será "false" (falso), lo que indica que el elemento aparecerá en todas las derivaciones (es común). Para el caso de estudio del presente trabajo de grado, la información de variabilidad esta codificada en los modelos UML (diagramas de clases) mediante el uso de valores etiquetados que representan la información de variabilidad en la forma etiqueta-valor, de modo que la identificación de un punto de variabilidad (elemento variante o componente variable) es realizada asignando el valor "true".

Para la descripción del caso de estudio fueron seleccionados los diagramas de actividades basados en la notación DTD (ver figura 24), los cuales son similares a los diagramas de actividades UML. La elección de este tipo de diagramas responde a que están caracterizados por mostrar la secuencia de estados por los que pasa un caso de uso, un objeto a lo largo de su vida, o bien, todo el subsistema que constituye el prototipo, facilitando el entendimiento de la aplicación; además, amplían la descripción de la vista funcional de la arquitectura, al introducir un ejemplo práctico, ya que, como fue mencionado en la sección 5.3.2.4, esta vista está directamente relacionada con el caso de estudio y el diagrama de actividades es una de las opciones que la metodología "4+1" sugiere emplear para su descripción [15], [63], [86]. Algunos de los símbolos del diagrama que pueden resultar confusos al momento de interpretarlos, son explicados en la tabla 10.

Símbolo	Significado
Sistema externo	Cualquier sistema, usuario o equipo que, sin formar parte de la aplicación, puede establecer una comunicación de datos o acciones con la misma
Almacén de datos	Identifica un sistema de almacenamiento de información, como una base de datos o una carpeta
◌	Identifica un punto de entrada o salida del diagrama. Representa una conexión entre diferentes diagramas o una abstracción de un elemento no significativo, como puede ser la interfaz de usuario

Tabla 10. Aclaración de símbolos del diagrama

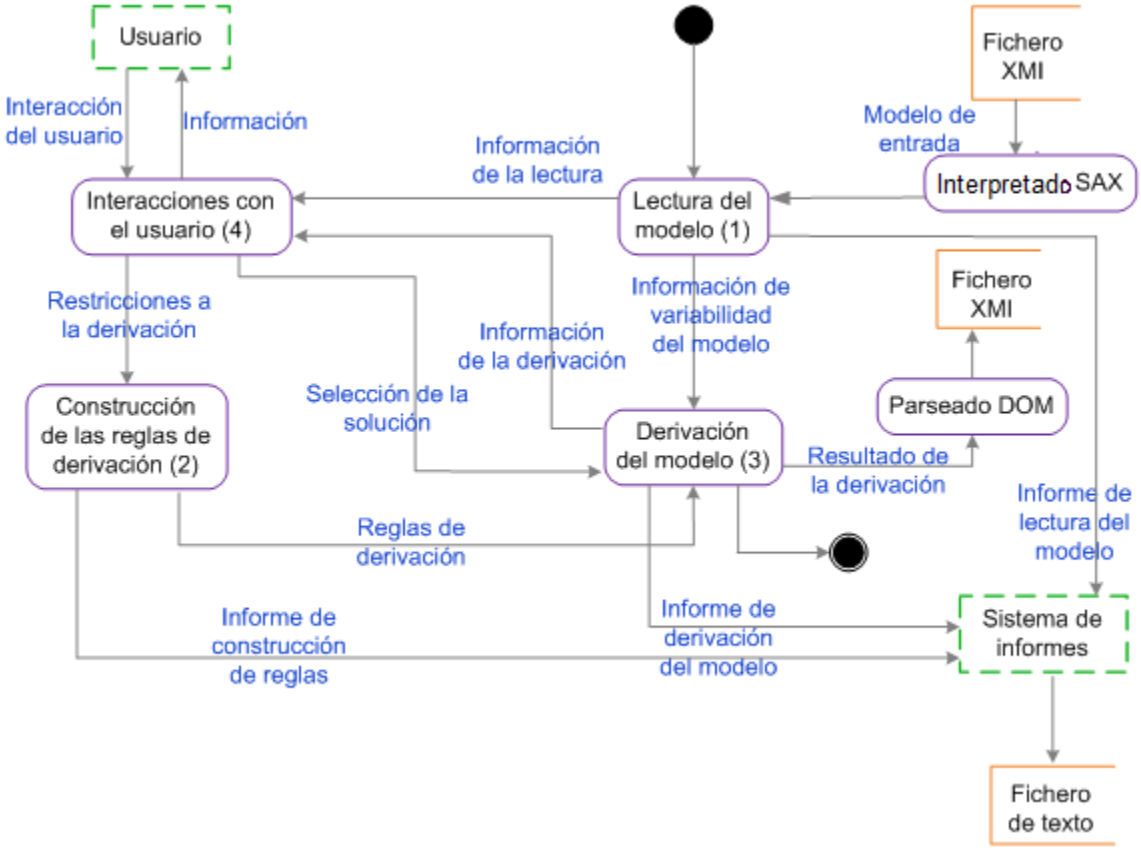

Figura 24. Diagrama general de la aplicación

En las siguientes secciones están descritas cada uno de los procesos que conforman el diagrama general de la aplicación (figura 24) y por lo tanto el caso de estudio.

6.2.3. Lectura del modelo

Este primer proceso es encargado de acceder y leer el archivo que contiene el diagrama de clases utilizado como entrada de la aplicación, es decir, el diagrama seleccionado por el usuario; toma toda la información relevante para construir un modelo de datos de memoria y exportar dicha información a la aplicación en un archivo de tipo XMI. La información que tendrá el modelo de datos en memoria es el punto de salida principal de este proceso y servirá de

entrada para los procesos de creación de reglas y de derivación. En el anexo A, en la figura 1 se encuentra un diagrama que ilustra este proceso.

El archivo XMI que contiene el diagrama será cargado en la aplicación mediante el uso de la clase FileReader que ofrece java para cargar archivos desde cualquier ubicación en disco y leído a través del analizador sintáctico SAX a través de la implementación de métodos heredados de las clases empaquetadas ofrecidas en org.xml.sax. Permiten, en conjunto con las características del analizador, leer todo el archivo y cargar tanto los elementos del mismo como sus atributos, de forma dinámica y rápida, mejorando de esta manera la velocidad de la aplicación y el uso de memoria en el momento de ejecución de la lectura, teniendo en cuenta que el archivo XMI viene bien definido y estructurado desde eUML2, a diferencia del proyecto [15] en donde fue necesario primero adecuar el archivo a la versión de ese entonces del estándar XMI para que éste pudiese ser analizado dentro de la aplicación.

Cabe agregar que la construcción del modelo de datos está ligada al proceso de lectura del archivo XMI, debido a que dicho modelo es una estructura que representa la información del diagrama de clases, necesaria para realizar la derivación y cuya construcción avanza conforme es leído el archivo XMI que contiene la información.

6.2.4. Construcción de las reglas de derivación

La intención con este proceso[43] es analizar las restricciones a la derivación introducidas por el usuario y junto con la información disponible del modelo, obtenida como resultado del proceso anterior, producir un conjunto de reglas utilizables por la aplicación y que representen dichas restricciones. Básicamente existen dos tipos de reglas, las reglas lógicas y las reglas de selección múltiple (que dan lugar a los componentes intercambiables mencionados con anterioridad), cada una sigue un proceso de creación diferente según su tipo (ver figura 2 del anexo A), y corresponde con algunos de los mecanismos de variación de la técnica de derivación descritos en la sección 3.5.2.5, a saber: elección de componentes, selección de componentes y selección lógica.

Las reglas de restricción representan características impuestas o excluyentes en las soluciones válidas de la derivación, son organizadas en forma de cadena (ver anexo A, figura 3), de modo que las soluciones finales deben cumplir todo el conjunto de reglas introducidas por el usuario, dando lugar a las reglas de derivación.

Las reglas de selección basan su funcionamiento en la selección de un valor entre un conjunto de posibilidades disponible, así como en indicar si se desea que dicho valor seleccionado esté presente o ausente en las soluciones de la derivación (el valor “true” y “false” ya mencionados). Sin embargo, esta estructura de etiqueta-valor permite un abanico de posibilidades mucho más amplio. El diseño de la aplicación puede permitir que cualquier pareja de cadenas etiqueta-valor sea válida, así como también que para una misma etiqueta pueda generarse un rango de posibilidades de selección que controlen la derivación de cualquier manera que el usuario/arquitecto del modelo necesite. Ni el tipo, ni los valores de las etiquetas están predeterminados, lo que implica que este tipo de restricciones puede particularizarse para cada caso. Así, por ejemplo, si el interés es discriminar la derivación con información acerca de la plataforma objetivo (PSM, para MDA), se podría caracterizar los elementos del modelo con la etiqueta “Plataforma” y el rango de valores “Windows”, “Linux”, “MacOS…”. Algunos otros

[43] El proceso de construcción de reglas ha sido tomado de [15].

posibles campos de derivación podrían ser el lenguaje de implementación, el tipo de aplicación objetivo, o cualquiera con una única restricción.

Las reglas lógicas operan de igual forma que las reglas de selección, estableciendo características que deben estar presentes en las soluciones consideradas válidas. Sin embargo, al contrario que las reglas de selección, las reglas lógicas basan su comportamiento en el significado de expresiones de lógica proposicional. Una restricción de lógica proposicional[44] utiliza una proposición lógica para definir relaciones lógicas entre los elementos del modelo. Este tipo de restricciones no hace uso de los valores etiquetados (más allá de la identificación de puntos de variabilidad), sino que referencia directamente los elementos del diagrama. La resolución de expresiones proposicionales recibe el nombre de evaluación de expresiones, suele caracterizarse por la tabla de verdad de la proposición, en ella para todas las posibles combinaciones de valores lógicos de entrada un valor lógico de salida es identificado. El proceso de construcción de reglas lógicas es explicado e ilustrado en el anexo A (ver figura 4), al igual que la explicación sobre la incorporación de las reglas a la cadena. En este proyecto las expresiones proposicionales pueden utilizar conectores lógicos de los siguientes tipos:

Conector	Significado	Símbolo
AND	Conjunción: la proposición es cierta si lo son todos sus componentes	&
OR	Disyunción: la proposición es falsa si lo son todos o alguno de sus componentes	v
XOR	Disyunción exclusiva: la proposición es cierta sólo si sus componentes tienen valores opuestos	+
NOT	Negación: invierte el valor de verdad de una proposición	-
Paréntesis	Agrupa proposiciones simples en otras más complejas y modifica la precedencia de los operadores	()

Tabla 11. Conectores válidos en las restricciones de lógica proposicional

6.2.5. Derivación del modelo

Después de caracterizada la variabilidad del diagrama de la arquitectura de la SPL, por medio de etiquetas, es posible derivar el modelo para obtener el diagrama de clases de un videojuego en específico. El proceso de derivación[45] del modelo consiste en imponer las reglas creadas en el proceso anterior, como restricciones al conjunto de aplicaciones que son soluciones válidas del proceso.

El modelo de entrada para la derivación proviene de los dos procesos ya descritos. Dicho modelo contiene información sobre la variabilidad de la SPL, pero no sobre las restricciones a la misma, por lo que en un principio el conjunto de posibles soluciones de derivación, solo está limitado por la cantidad de puntos de variabilidad que presenta el modelo, que pueden ser de tipo variable o no; de tal forma que un modelo de entrada que tenga x elementos (paquetes, clases o interfaces) de los cuales n son variantes y (x-n) son invariantes, puede tener 2^n posibles derivaciones de salida, tantas como combinaciones de los diferentes elementos variantes sea posible formar (ver figura 5, anexo A). El proceso de derivación opera construyendo una representación de esas 2^n posibles derivaciones de salida, y ejecutando la

[44] Las restricciones proposicionales son expresiones que forman parte del álgebra fundamental y que relacionan elementos a través de un conjunto de palabras clave denominadas conectores, aportando a cada conector un significado lógico distinto.
[45] Algunos de los conceptos empleados para la derivación del modelo han sido tomados de [15].

cadena de reglas, es decir, para cada filtro presente en la cadena, el proceso de derivación comprueba si cada una de las posibles derivaciones de salida cumple las restricciones que el filtro representa. En caso contrario, esa posible derivación no es una solución válida y es descartada. Una vez recorrida toda la cadena de filtros, el resultado es el conjunto de posibles soluciones que son válidas, es decir, que cumplen todas las restricciones introducidas por el usuario. Para el presente caso de estudio, la derivación es un proceso automático, donde el usuario solo debe seleccionar que combinación o combinaciones de entre las válidas desea guardar. En el anexo A (ver figura 6), el proceso de derivación es mostrado.

Inicialmente el proceso de derivación contempla todas las posibles soluciones como válidas, por lo que genera una representación de todas las posibles combinaciones que la presencia o ausencia de los elementos variantes del modelo puede producir. Para modelos medianos o grandes, el número total de posibles variaciones puede llegar a ser muy elevado, por lo que esta representación debe tener en cuenta restricciones en cuanto a memoria y optar por un esquema de implementación con el mínimo consumo de memoria posible. Para tal propósito, la clase llamada DerivationVector es definida, representa las posibles combinaciones que un modelo puede producir, implementando un vector de BitSets donde los elementos variantes del modelo son ordenados y les es asignado un valor y una posición en el vector, representando una posible combinación de elementos variantes a través del valor de los bits que componen el vector, el 1 indica la presencia de ese elemento en la combinación y el 0 su ausencia. La utilización de BitSets es debida a que éstos representan un único bit por elemento y combinación, característica que facilita alcanzar el objetivo de disminuir el consumo de memoria y mejorar el alcance en cuanto a rendimiento y capacidad del caso de estudio. Sin embargo, a pesar de esta medida preventiva y ahorrativa, es fácil comprobar (ver tabla 12) cómo el consumo de memoria necesario para representar todas las combinaciones crece exponencialmente con el número de elementos variantes del modelo.

Número de elementos variantes	Número de combinaciones posibles	Memoria necesaria para las combinaciones
4	2^4=16	2^4.4 bits = 8 Bytes
12	2^{12}=4096	2^{12}.12 bits = 6 KBytes
16	2^{16}=65536	2^{16}.16 bits = 128 KBytes
20	2^{20}=1048576	2^{20}.20 bits = 2,5 MBytes
32	2^{32}=4294967296	2^{32}.32 bits = 16 GBytes

Tabla 12. Memoria requerida en función de la cantidad de elementos variantes

Como es posible observar en la tabla 12, después de 20 elementos el consumo de memoria es considerablemente alto y dificulta el funcionamiento de la aplicación, por lo tanto, es necesario encontrar una solución que reduzca el consumo de memoria e incremente la velocidad de ejecución, sin limitar el número de elementos variantes a 20, lo cual sería dar una ineficiente solución al problema que impondría serías limitaciones a los beneficios (el alcance) que pueden lograrse con la implementación de una SPL.

Este trabajo de grado implementará la solución propuesta en [15] que plantea derivar por dominios, entendiendo por dominio un elemento variante y contenedor, es decir, el modelo general o uno de los paquetes contenido. Donde los dominios son excluyentes, es decir, el dominio de modelo abarca todos los elementos que contiene el modelo, incluido los paquetes; pero los elementos contenidos en cualquiera de esos paquetes ya no forman parte del dominio del modelo, sino del dominio del paquete al que pertenecen. Del mismo modo, un dominio de paquete abarca todos los elementos que dicho paquete contiene, y no los contenidos en el interior de otros paquetes, independientemente de que estén también en su árbol jerárquico.

De esta forma los elementos del modelo quedan clasificados de forma natural en función del elemento que los contiene y cada paquete, al igual que el dominio del modelo, es derivado de forma individual y el resultado de la derivación es el producto cartesiano de los resultados de cada una de las derivaciones de dominio. En la sección de resultados es explicado, mediante un ejemplo, las ventajas de implementar esta solución.

Cabe agregar que este conjunto de soluciones de salida puede contener un elemento; puede estar vacio, si no hay ninguna posible derivación que cumpla todas las restricciones introducidas; o puede contener varios elementos, si más de una posible derivación cumple todas las restricciones introducidas, como fue dicho en el resumen del caso de uso 6 “derivar”. Debido a esto, es necesario realizar después de la derivación un proceso de selección de soluciones, en el que el usuario indica que modelo solución, de entre todas las soluciones válidas, desea que se almacenen en archivos XMI.

6.2.4. Interacciones con el usuario

Este proceso agrupa todo el conjunto de la lógica de operación de la aplicación y de la interfaz de usuario. El objetivo principal de este proceso es el intercambio de datos con el usuario, entendiendo por datos, los comandos de operación como la apertura de un modelo, cierre de un modelo, visualización de informes, creación de reglas de derivación, cierre de un modelo, cierre de la aplicación; la información necesaria para la ejecución como restricciones a la derivación y selección de una solución final; y la información producida por la aplicación durante los procesos principales y su presentación al usuario en forma de informes.

6.3. Criterios de evaluación y pruebas

Los componentes hardware empleados para la construcción del caso de estudio, fueron:

- Procesador: AMD Turion(tm) Dual-Core Mobile RM-70, 2GHz
- Memoria RAM: 2GB
- Disco duro: 250GB

Los componentes software empleados para la construcción del caso de estudio, fueron:

- Sistema operativo Windows Vista Home Premium, Service Pack 2.
- Máquina virtual java: jdk1.6.0_07.
- Eclipse 3.4, Ganymede.
- eUML2 para Eclipse.

Debido a que el objetivo principal que persigue el presente trabajo de grado es definir la arquitectura que debe utilizar un entorno de desarrollo para soportar el trabajo con familias de sistemas bajo el enfoque propuesto por MDA, los modelos de dicha iniciativa que conciernen al caso de estudio son los PIM y por lo tanto las pruebas ejecutadas sobre la aplicación están enfocadas en la comprobación del funcionamiento correcto de la lectura y procesamiento de modelos, así como de la derivación y escritura de componentes. Para trabajos futuros que implementen los otros niveles de abstracción de MDA y que lleguen a realizar transformaciones verticales que entreguen código de un producto en específico, sería importante y conveniente realizar algunas pruebas sobre dichos productos, no sin antes tener en cuenta que como ya se dijo en el capítulo 3 en las secciones de ingeniería de dominios y de aplicaciones, y en la descripción del proceso de derivación, la variabilidad dificulta realizar una evaluación minuciosa de la arquitectura, ya que incluso un número pequeño de puntos de variación, rápidamente llevarán a un número inmenso de configuraciones posibles, haciendo muy difícil evaluar todos

los productos potenciales en una línea, lo que implica que la arquitectura no pueda ser evaluada para cada producto, porque no se pueden predecir con precisión, cuáles serán los nuevos productos y mucho menos cuántos [33].

En este caso de estudio son aplicados algunos de los fundamentos económicos de las SPL relacionados con la gestión y delimitación del producto a entregar. Para la delimitación del portafolio de producto y con el fin de probar la capacidad de la aplicación para manejar modelos de tamaño medio y grande, así como para realizar derivaciones complejas, es desarrollada una SPL que entregue PIM con información específica a una familia de videojuegos de simuladores de vuelo, es decir, ellos determinan el rango de productos soportado. El modelado de dicha SPL es realizado empleando la información recogida y consignada en [15] sobre simuladores reales, adaptada al concepto de videojuegos. La complejidad definida para su implementación sirve al propósito de mostrar las capacidades de la herramienta desarrollada. Para la delimitación del dominio es definido como modelo de entrada un diagrama de clases que representa un núcleo central de control (activo definido en la tabla 8) de un videojuego, compuesto por el dominio principal o dominio del modelo (núcleo central del videojuego), cuatro paquetes de organización, cada uno de los cuales representa un dominio de derivación independiente, 35 clases y ocho interfaces. Finalmente, la delimitación de los activos está consignada en el anexo A, en la sección de descripción de clases del modelo de entrada.

Bajo el dominio del modelo están agrupadas aparte de los paquetes de organización, descritos más adelante, 4 clases del núcleo de la SPL de videojuegos, estos últimos descritos en el anexo A (ver tabla 8).

En el paquete Comunicaciones están agrupadas las clases encargadas del sistema de comunicación del simulador. Los elementos variables de este paquete están caracterizados, además del par, variable: *true* – *false*, que especifica su variabilidad, por los pares, componente: visual – sonido – movimiento, que permiten discriminar las tareas de comunicaciones en función del componente al que van dirigidas y el par integración: real – simulada, que permite indicar si se desea que el sistema de comunicaciones de los componentes sea el final, o si se prefiere una simulación de las comunicaciones para poder integrar el simulador, a falta del despliegue final de algunos de sus componentes, situación que aplica al presente proyecto debido a que no trabajará con la parte hardware de los videojuegos. Este paquete contiene 10 clases, cinco interfaces, descritas en el anexo A (ver tabla 9).

En el paquete ModelosMatemáticos están agrupados todos aquellos elementos encargados de la representación activa del entorno del videojuego, como la representación del comportamiento de la gravedad, de la trayectoria de un proyectil, de las fuerzas de movimiento y de fricción, de las leyes de aerodinámica, etc., así como del comportamiento de elementos o eventos que existen o sucederían de forma real en el entorno que el videojuego simula, pero que no pueden modelarse de forma física en éste; tal es el caso del comportamiento de los instrumentos de cabina, los eventos de colisión o de los blancos, las armas y la munición si los videojuegos son de tipo combate, etc. Los componentes variables de este paquete están caracterizados según dos parámetros, entorno: aéreo – terrestre, que permite discriminar los modelos de entorno en función del ámbito general de simulación y tipo: misión – entrenamiento, que permite diferenciar si los módulos de entorno están enfocados hacia la competencia o el combate (de tipo militar), o a adquirir habilidad y destreza en el vuelo. Este paquete contiene nueve clases y una interfaz, descritas en el anexo A (ver tabla10).

En el paquete Elementos están agrupados los elementos móviles y que interactúan con el usuario durante el juego, es decir, aquellos encargados de rellenar el entorno, dotándolo del

realismo necesario para conseguir que el videojuego simule la acción de volar, como otros aviones encontrados en el espacio aéreo o en la base de despegue, helicópteros, carros que transitan por una vía terrestre que es sobrevolada, personas en un parque, etc. A diferencia del entorno, que está basado en una serie de reglas globales y predefinidas, los elementos del entorno táctico son inherentemente individuales y su comportamiento está basado más en reglas de inteligencia artificial y respuesta ante estímulos, en lugar de reglas globales. En este paquete los elementos variantes están caracterizados mediante el mismo tipo de información que el paquete anterior. Al emplear las mismas etiquetas en diferentes paquetes o dominios del diseño es posible realizar derivaciones que proporcionen arquitecturas para videojuegos específicos como simuladores de vuelo, para entrenamiento o combate, en entornos aéreos, acuáticos o terrestres, tan sólo indicando una o dos reglas de derivación. Según esto, es posible deducir que las reglas de selección están enfocadas en dirigir globalmente la derivación, mientras que las reglas lógicas establecen relaciones entre elementos, y solo actúan dentro de un dominio, por lo que su uso como reglas de derivación está más enfocado a particularizar la forma de la derivación entre un número limitado de elementos y dominio. Este paquete contiene nueve clases, descritas en el anexo A (ver tabla 11).

El paquete Interacciones incluye el módulo de control del simulador de vuelo, desde él pueden modificarse el comportamiento y las características que gobiernan el núcleo central de la simulación, así como las clases e interfaces del modelo de interfaz de usuario. Las clases variantes están caracterizadas mediante un único parámetro, control: GUI – RV, permite diferenciar entre un sistema de control del núcleo basado en un entorno gráfico (GUI) y uno basado en reconocimiento de voz (RV). Este paquete contiene cuatro clases, una interfaz, descritas en el anexo A (ver tabla 12).

La derivación del caso de estudio parte de la suposición de que el simulador va a prescindir de la plataforma de movimiento, pero que tendrá tanto sistema de representación visual como de comunicaciones, ello puede expresarse mediante la regla de selección ELIMINAR componente de movimiento, indica al proceso de derivación que debe eliminar todos los elementos caracterizados por la pareja de metadatos componente – movimiento, respetando el resto de elementos del diseño. También es supuesto que simulará un avión para mejorar la destreza en el vuelo, por lo que el tipo indicado será entrenamiento, mediante la regla SELECCIÓN tipo - entrenamiento.

En un principio podría parecer que el entorno de simulación debe ser aéreo e indicarlo mediante la regla SELECCIÓN entorno aéreo, sin embargo, si es deseado que el avión esté diseñado para combatir objetivos terrestres sería necesario simular también, móviles y objetivos en tierra, más para este caso, como parte del supuesto que el fin que persigue el usuario del videojuego es solo de entrenamiento, esta situación es expresada mediante la regla SELECIÓN entorno - terrestre. Otra suposición para el videojuego es que el control va a estar basado en un entorno gráfico, mediante la regla de selección SELECCIÓN control GUI. Por último, es posible suponer que no es necesario el sistema de generación de sonidos o que su construcción será ordenada a terceros y que el resto de los activos del simulador serán de producción propia de la empresa y son definidos, por tanto, un conjunto de reglas lógicas que permitan integrar los componentes (activos) disponibles y simular las comunicaciones con el subsistema de generación de sonido. El conjunto de reglas que permitirá derivar el modelo arquitectónico con esas característica es, SistemaVisualReal AND (NOT SistemaVisualSimulado); SistemaControlReal AND (NOT SistemaControlSimulado); SistemaSonidoSimulado AND (NOT SistemaSonidoReal). No es necesario especificar información acerca de las comunicaciones con el subsistema de movimiento, puesto que la

regla de selección introducida en primer lugar ya afecta a los componentes relacionados con este subsistema, eliminándolos de los diseños derivados.

6.4. Resultados

Los resultados obtenidos con el caso de estudio son descritos para los tres procesos principales implementados en las siguientes secciones.

6.4.1. Lectura del modelo

El resultado de la lectura del modelo de entrada efectuada por el plugin es mostrado en la figura 25.

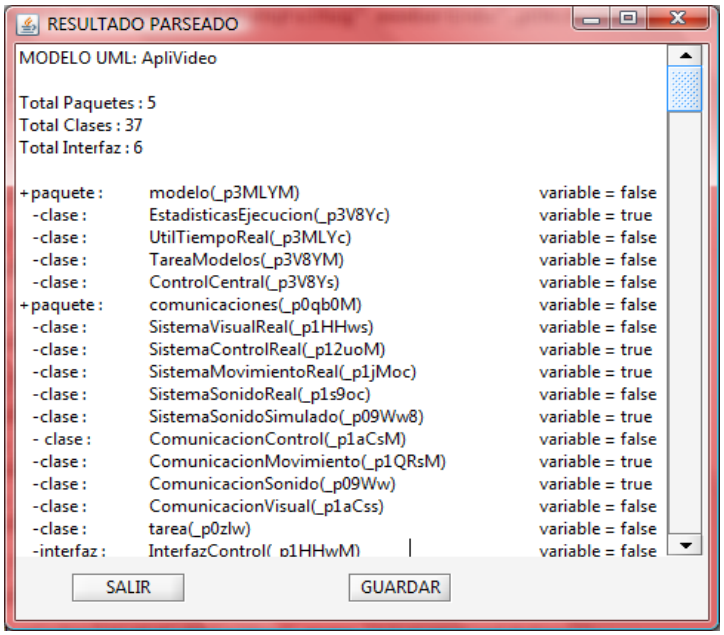

Figura 25. Lectura del modelo de entrada.

6.4.2. Derivación del modelo

El modelo de entrada definido para las pruebas consta de cuatro paquetes variantes y una clase variante correspondiente al dominio del modelo. El paquete Comunicaciones tiene ocho elementos variantes, el de ModelosMatematicos seis, el de Elementos siete y el de Interacciones tres; para un total de 29 elementos variantes, resultado de la suma de cuatro paquetes, una clase del dominio del modelo y 24 elementos de todos los paquetes. El número de elementos no variantes es irrelevante a este caso de estudio, ya que estos elementos no influyen en el proceso de derivación, son constantes y por lo tanto no son tenidos en cuenta.

Si la derivación no fuera realizada por dominios, este caso de estudio tendría 2^{29}=536'870.912 posibles combinaciones, cada una de las cuales consumiría 29 bits de memoria, para ocupar un total de 1,81 GBytes. Además de la gran cantidad de memoria necesaria bajo esta forma de derivación, también es significativa la enorme carga de procesamiento y el tiempo que implicaría analizar cuáles combinaciones son válidas y cuáles no lo son, de las 536'870.912 posibles, con el sistema de reglas y derivación de soluciones. Sin embargo, al aplicar el concepto de derivación por dominios, el trabajo esta presente en cinco dominios disjuntos (ver sección 3.3.6.), ellos son:

- El dominio del modelo, con cinco elementos variantes, una clase y cuatro paquetes.
- El dominio del paquete Comunicaciones, con ocho elementos variantes.
- El dominio del paquete ModelosMatematicos, con seis elementos variantes.
- El dominio del paquete Elementos, con siete elementos variantes.
- El dominio del paquete Interacciones, con tres elementos variantes.

Al derivar de forma independiente y disjunta cada uno de estos dominios los siguientes resultados son obtenidos:

- El dominio del modelo tiene 2^5=32 posibles combinaciones, cada una de las cuales consumiría cinco 5 bits de memoria, ocupando un total de 20 bytes.
- El dominio del paquete Comunicaciones tiene 2^8=256 posibles combinaciones, cada una de las cuales consumiría ocho bits de memoria, ocupando un total de 256 bytes.
- El dominio del paquete ModelosMatematicos tiene 2^6=64 posibles combinaciones, cada una de las cuales consumiría seis bits de memoria, ocupando un total de 48 bytes.
- El dominio del paquete Elementos tiene 2^7=128 posibles combinaciones, cada una de las cuales consumiría siete bits de memoria, ocupando un total de 112 bytes.
- El dominio del paquete Interacciones tiene 2^3=8 posibles combinaciones, cada una de las cuales consumiría tres bits de memoria, ocupando un total de 3 bytes.

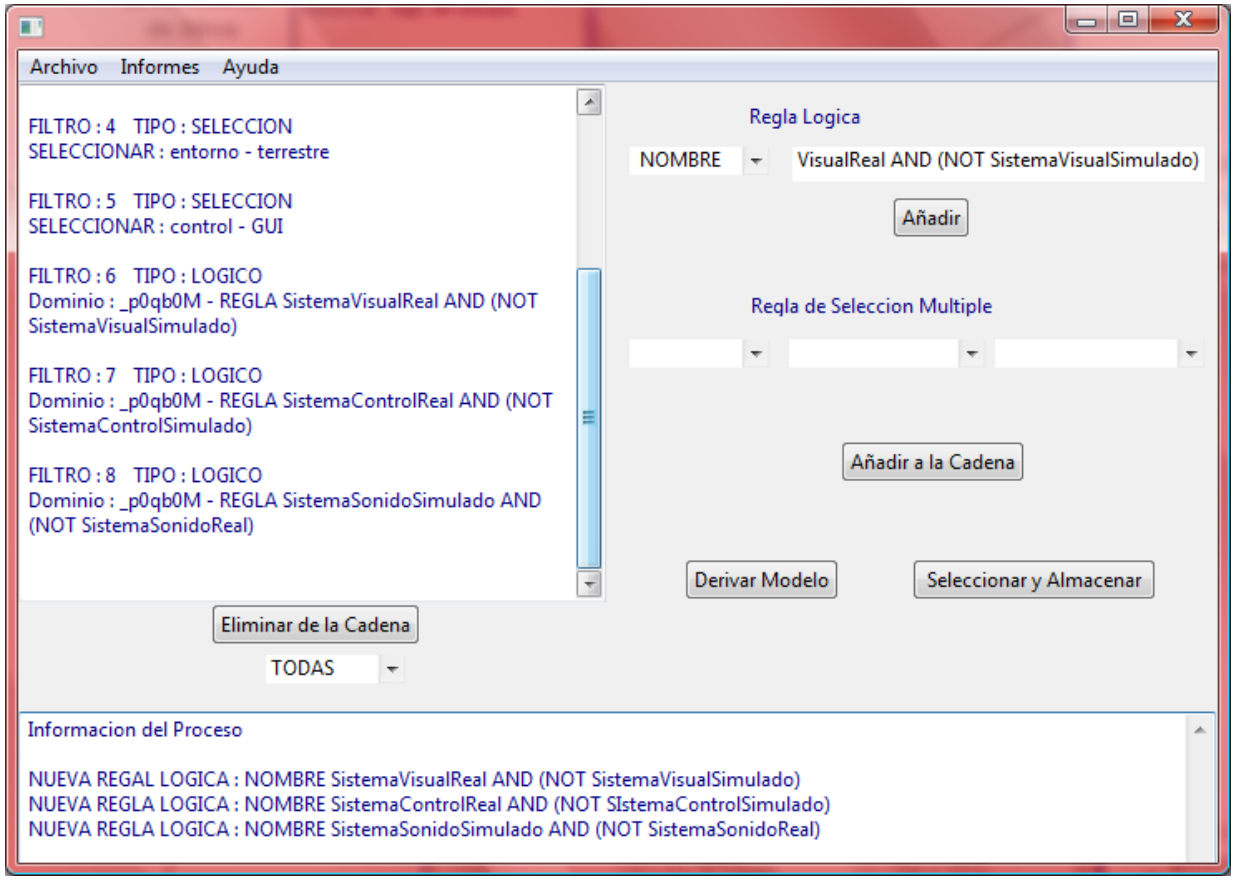

Figura 26. Derivación del modelo.

La derivación del modelo por dominios produce cinco rangos de resultados, uno para cada uno de los dominios que han sido derivados de manera independiente y el resultado final de la derivación es la combinación conjunta de una solución cualquiera de cada uno de estos cinco dominios. A diferencia de la primera forma de derivación planteada que consume 1,81 GBytes

de memoria, esta segunda forma consume un total 429 bytes, resultado de la suma de 20 bytes del dominio del modelo, 256 bytes del dominio del paquete Comunicaciones, 48 bytes del dominio del paquete ModelosMatematicos, 112 bytes del dominio del paquete Elementos y tres bytes del dominio del paquete Interacciones. El proceso de derivación de soluciones por dominios debe analizar 488 posibles combinaciones, en lugar de 536'870.912 , resultado de la suma de 32 combinaciones del dominio del modelo, 256 combinaciones del dominio del paquete Comunicaciones, 64 combinaciones del dominio del paquete ModelosMatematicos, 128 combinaciones del dominio del paquete elementos y ocho 8 combinaciones del dominio del paquete Interacciones; obteniendo como resultado final, no solo una mejora significativa en el consumo de memoria, sino también en la velocidad de derivación. El proceso de derivación e inserción de reglas efectuada por el usuario es mostrado en la figura 26.

6.4.3. Creación del modelo

El resultado de la selección del modelo final y la creación del mismo efectuada por el plugin es mostrado en la figura 27.

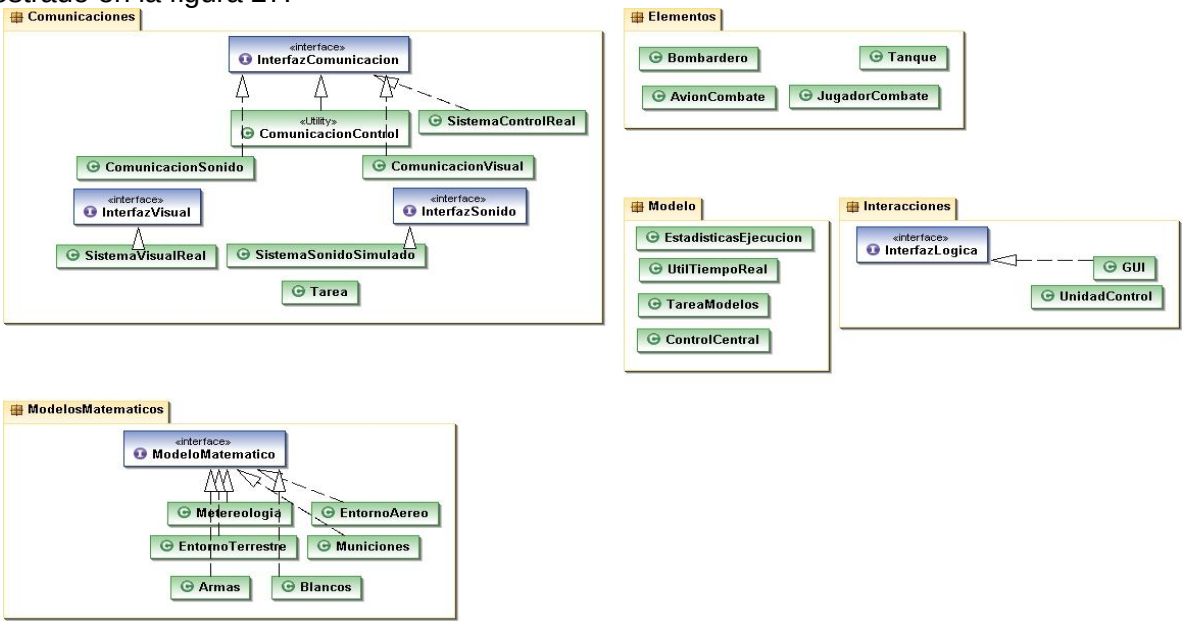

Figura 27. Creación del modelo

6.5. Problemas e inconvenientes

En el proceso de desarrollo de la herramienta mda.sql.architecture fueron encontrados los siguientes problemas para su ejecución:

- Software libre: a pesar de que el estudio realizado sobre las herramientas de modelado UML 2.1 de acceso libre para desarrollos de tipo académico compatibles con Eclipse Ganymede (Eclipse v. 3.4) nos mostro una amplia gama de las mismas para ofrecer al usuario un modelado completo de su línea de producto para ser derivado, ninguna de las licencias presentó total compatibilidad con la exportación de los modelos a XMI o el funcionamiento de su etiqueta valor. Después de realizar diferentes pruebas de modelado y mantener la idea de utilizar software libre, fue decidido utilizar la herramienta eUML2 que soporta la exportación de archivos XMI y tenía etiquetas de valor predefinidas que fueron modificadas a través de la misma herramienta para poder realizar la lectura del archivo XMI con éxito en el plugin.

- Etiqueta Valor: las etiquetas de valor necesarias para la identificación de propiedades de las clases no podían ser modificadas por las etiquetas de valor de la clase predefinida por su falta de particularidad, por ello fue decidido trabajar con la parte de documentación de la clase para poder adicionarlas al archivo XMI.

- Licencia: para el *plugin* eUML2 mostrada en la página como freeEdition es una licencia de evaluación de 15 días que no cuenta con las características mencionadas en el punto anterior, por esta razón el plugin debió descargarse e instalarse en varias ocasiones en el Eclipse Ganymede.

- Pruebas: Al realizar las pruebas de ejecución del *plugin,* tomaba mucho tiempo en mostrar el resultado debido a la carga de aplicaciones en el Eclipse Ganymede, lo que demoraba el proceso de desarrollo de la herramienta, después de llevado a cabo el análisis del problema fue decidido trabajar el eclipse ejecutado sin ninguna aplicación o plugin descargado sobre él para agilizar el proceso.

- Código: El archivo XMI que exporta la herramienta eUML2 maneja códigos de identificación demasiado extensos para todos los componentes UML que trae el plugin, por lo que fue decidido trabajar solo con la parte identificadora del componente específico ya que el id es generalizado para cada paquete del modelo.

- Almacenamiento y búsqueda: debido a la simplificación del código el proceso de almacenamiento de los componentes de la aplicación en vectores fue realizado de forma directa, pero la búsqueda fue complicada un poco, este proceso fue mejorado en el proceso de desarrollo del software adquiriendo habilidad en el manejo de índices y métodos que contienen las clases abstractas predefinidas de eclipse.

- http://dsc.ufcg.edu.br/~ledo/project_management.jpg

CAPÍTULO 7.

CONCLUSIONES

7.1. Conclusiones

Una vez finalizado el trabajo y estudios realizados para el desarrollo del presente proyecto, así como para la elaboración de este documento, son listadas las principales conclusiones obtenidas con de él, es importante resaltar que esta inmerso en un campo innovador para el desarrollo de software como son las Líneas de Productos Software y la Arquitectura Dirigida por Modelos.

- Este trabajo de grado estudia y analiza los conceptos relacionados con las SPL, entre ellas la gestión de la variabilidad, los aspectos económicos y técnicos del enfoque, las ventajas y desventajas que su aplicación puede traer a los interesados en implementar una línea de productos y los principales proyectos desarrollados o que actualmente están en ejecución y siguen los principios propuestos por SPL. De la iniciativa MDA estudia y analiza los modelos y estándares propuestos por el OMG, las herramientas que implementan los conceptos del Desarrollo Dirigido por Modelos, las transformaciones entre modelos y al igual que las SPL, los principales proyectos desarrollados o que actualmente están en ejecución y que siguen los principios propuestos por este enfoque. Esta contribución es desarrollada en los capítulos 2, 3 y 4 del presente documento.

  A diferencia de los dos trabajos antecesores a este proyecto, [14] y [15], el presente trabajo de grado no centra todo el proceso de desarrollo de software en el enfoque SPL, por el contrario, plantea una solución híbrida entre SPL y MDA que saca provecho de las ventajas que cada enfoque ofrece (ver capítulo 3 y 4, respectivamente). Para ello define una arquitectura para una línea de productos a partir de la cual puedan derivarse modelos de alto nivel de abstracción e independientes de plataforma (PIM), mediante transformaciones horizontales que entregan PIM más detallados, que contienen información un poco más específica introducida por un usuario, sobre los productos que la línea implementará, es decir, diagramas UML de clases. Esta contribución es desarrollada en los capítulos 3, 4 y 5, y en el anexo A (modelado) del presente documento.

- Para los años en que los proyectos antecesores [14] y [15] finalizaron, 2002 y 2003 respectivamente, las herramientas de modelado y sus estándares, o relacionados con ellos, estaban en estado inmaduro y no brindaban un soporte adecuado, ni conceptual ni tecnológico, al desarrollo de software. En la actualidad la situación es diferente, es por esto que en este proyecto para verificar la arquitectura es posible contar con un estándar XMI maduro y con herramientas de modelado UML como la empleada, eUML2, que soportan archivos XMI, tanto lectura como escritura y todas las características del lenguaje UML; un entorno de desarrollo portable, robusto, completo, libre y extensible, como Eclipse, que hacen de la solución propuesta mejor en cuanto a costos, interoperabilidad, posibilidad de migración y calidad, entre otros aspectos.
  Esta contribución es desarrollada en el capítulo 6 y en los anexos A (modelado), C (manual de usuario) del presente documento y en la aplicación construida.

- Siguiendo el enfoque de las Líneas de Productos Software es posible obtener de forma inmediata y sencilla la arquitectura de los productos posibles de una SPL dada, ahorrando tiempo y recursos en el proceso de desarrollo software, al automatizar uno de los

mecanismos de implementación de variabilidad que existen, como la derivación desde la arquitectura de referencia hacia la de un producto específico, siguiendo el enfoque MDA.

- El *plugin* de Eclipse eUML2 es una de las herramientas de software libre que mejor pueden ser adaptadas a los objetivos del proyecto debido a las funcionalidades que proporciona. Es de destacar que implementa todas las características y diagramas definidos en UML 2.2. con un ambiente de modelado intuitivo, visual y sofisticado. Es compatible con Eclipse 3.5. y todas las versiones anteriores, incluyendo Ganymede, que es la versión de Eclipse 3.4. (2008-2009) utilizada en este proyecto.

- Los modelos de variabilidad obtenidos al implementar una SPL a través del mecanismo de derivación son candidatos naturales a ser considerados como PIM; para obtener dichos modelos es posible seguir el enfoque MDA. Debido a que esta iniciativa centra el proceso de desarrollo de software en la creación y transformación de modelos es factible obtener PIM's más detallados a través de transformaciones de otros modelos PIM, es decir, a través de transformaciones horizontales, o en el mismo nivel de abstracción.

- Debido a que tanto MDA como SPL son enfoques promovidos por actores importantes de la industria del desarrollo de software [25], el éxito de uno sobre el otro puede depender de factores externos, ajenos a sus características y bondades, por ejemplo: a estrategias de mercadeo, a la posición en el mercado de sus respectivos promotores o a intereses económicos de las partes implicadas.

- En comparación con MDA, SPL es un enfoque más joven si se toma en consideración lo que hace falta por trabajar (no cuando fueron iniciados los trabajos en MDA y SPL). Este hecho se ve reflejado en la pequeña cantidad y poca madurez y difusión de las herramientas existentes en el mercado para desarrollo de software, que siga las directrices de las SPL. Por tal razón es necesario trabajar más al respecto, para lograr llevar a la práctica los conceptos que este enfoque propone, con aplicaciones como la realizada en este proyecto.

- Los niveles de abstracción en la historia del desarrollo de software se han incrementado sucesivamente, el siguiente paso en esta evolución parece dirigirse hacia los lenguajes de modelado; al respecto la iniciativa MDA sienta las bases al definir de manera clara la tecnología que recomienda utilizar para su enfoque (UML, MOF, QVT, etc.) y las SPL proporcionan una guía metodológica a los desarrolladores al recomendar la creación de arquitecturas para las SPL, entornos de implementación, patrones de diseño y mecanismos de variabilidad.

- Es adecuado seguir el enfoque SPL cuando existe la intención de construir una familia de sistemas y/o se va a trabajar en un dominio específico. Es adecuado seguir el enfoque MDA cuando la interoperabilidad con otras herramientas o el uso de herramientas existentes (en especial las que siguen los estándares de OMG) sea un factor decisivo [25]. Sin embargo, en muchos casos prácticos los dos escenarios confluyen en uno solo, por lo que una estrategia mixta como la propuesta en este trabajo, saca provecho de las ventajas que presenta cada uno de estos enfoques y satisface las necesidades de los dos escenarios planteados.

- La arquitectura de una SPL, por si sola, constituye un activo muy valioso que puede ser reutilizado, debido a que representa conocimiento y experiencia en el dominio del problema, en cuanto a requerimientos, conceptos, estructura, textura, etc.

- El metamodelado es un mecanismo para definir lenguajes de modelado, resulta muy útil en el contexto de MDA ya que permite definir modelos no ambiguos, de forma tal que una herramienta pueda leer, escribir y entender los modelos construidos, para definir una aplicación de manera conceptual y, también, para especificar las reglas de transformación de un modelo de cierto nivel de abstracción a otro de diferente nivel.

- Una alternativa para el desarrollo de software bajo el enfoque MDA, es crear modelos más claros y de más fácil mantenimiento usando de forma conjunta UML y XMI, debido a que estos estándares definen como almacenar e intercambiar modelos, dos procesos fundamentales para la implementación de este enfoque.

- Mda.spl.architecture.project es una herramienta que genera modelos PIM detallados a partir de modelos PIM de alto nivel de abstracción a través de la edición de archivos tipo XMI, permitiendo al usuario administrar una SPL siguiendo el lineamiento MDA según las características de desarrollo de la línea requeridas por él mismo, lo que permite concluir que la viabilidad de la construcción de extensiones a herramientas que permitan construir software base en el desarrollo de familias de productos es buena.

- Las características incluidas sobre el PIM de alto nivel de abstracción en la herramienta eUML2 son propias del modelo de la línea de producto y dependen específicamente del arquitecto de línea, las reglas de derivación introducidas en la herramienta mda.spl.architecture.project para generar el modelo PIM detallado, estan totalmente relacionadas con las características mencionadas anteriormente dependiendo de esta manera del arquitecto de la SPL o de un buen conocedor de la misma, dado que mda.spl.architecture.project no genera reglas de derivación.

- El proceso de derivación implementado en mda.spl.architecure.project es totalmente automático (sigue un lineamiento matemático y lógico), basado en el modelo de PIM de alto nivel de abstracción leído y en las reglas lógicas y de selección introducidas por el usuario, el resultado de la derivación esta directamente relacionado con el conocimiento de la SPL y el producto que el usuario desea desarrollar, por lo tanto el nivel de aceptación de el PIM detallado debe ser definido por el usuario desarrollador del mismo.

7.2. Trabajos futuros

En esta sección están propuestos posibles trabajos futuros en el campo de las Líneas de Productos Software y la Arquitectura Dirigida por Modelos o aquellos que podrían ampliar o extender el presente proyecto, la descripción de los mismos es general debido a las limitaciones de espacio impuestas al presente documento final.

- **Incorporación de lenguaje de almacenamiento y gestión de modelos**
  En el presente proyecto se implementó una transformación horizontal de PIM a PIM, para dar continuidad a trabajos futuros que amplíen el realizado, es necesario definir un lenguaje que permita el almacenamiento y la gestión de los modelos que propone MDA (CIM, PIM, PSM y PSI) de tal forma que éstos sean intercambiados entre los diferentes niveles de abstracción y puedan ser realizadas las otras transformaciones, incluidas, las verticales.

- **Incorporación de un sistema de verificación y validación**
  Con la aparición del Desarrollo Dirigido por Modelos y de las Líneas de Productos Software, ha cobrado mayor importancia los procesos de verificación y validación de los desarrollos

software, debido a que al no disponer de modelos completos, consistentes y precisos, es inevitable la propagación de errores a etapas posteriores. Lo mismo puede ser aplicado a las arquitecturas de SPL. Un trabajo futuro podría incorporar, al presente proyecto, un sistema de verificación y validación que identifique y corrija errores en etapas tempranas del desarrollo, de tal forma que también pueda ser realizado una selección automática de una solución óptima entre el conjunto de soluciones válidas que producen las derivaciones. Además, a través de un sistema de políticas, pueden ser agregadas a la aplicación características adicionales deseadas para una solución óptima.

- **Incorporación de nuevo tipos de diagramas**
  Si bien los diagramas de clases son los más importantes y utilizados a la hora de realizar diseños UML, el formato XMI no está limitado a ellos, contempla todos los posibles diagramas de UML. Un trabajo futuro puede incluir la adaptación de la herramienta para extender la derivación a otro tipo de diagramas, por ejemplo: los diagramas de secuencia, de componentes o de casos de uso, ello permitiría a la herramienta cubrir de forma más completa las necesidades de sus usuarios [15].

- **Implementación de transformaciones verticales**
  La mayoría de las herramientas disponibles actualmente en el mercado y que promulgan dar soporte a MDA, realizan transformaciones verticales entre los diferentes niveles de abstracción que propone el enfoque, en especial de PIM a PSM y de PSM a PSI. Un trabajo futuro sería extender el presente proyecto, implementando transformaciones verticales entre modelos de diferentes niveles de abstracción, por ejemplo: de CIM a PIM, de PIM a PSM o de PSM a PSI.

- **Implementación de transformaciones horizontales de PSM a PSM**
  El presente proyecto implementó la transformación horizontal de PIM a PIM, un trabajo futuro sería implementar transformaciones horizontales entre los otros modelos que propone MDA y que están en un mismo nivel de abstracción, es decir, implementar transformaciones de PSM a PSM.

- **Implementación de la aplicación en proyectos desarrollados previamente**
  El caso de estudio del presente proyecto fue implementado para una empresa hipotética dedicada a la producción de videojuegos; sería interesante implementar en trabajos futuros la aplicación desarrollada, usando como modelos de entrada diagramas de clases definidos previamente en proyectos académicos y empresariales, que puedan ser fácilmente adaptados al concepto de SPL, de tal forma que sea posible comparar los resultados alcanzados sin la implementación de una SPL y con su implementación.

REFERENCIAS BIBLIOGRÁFICAS

[1] Introducción sistemas de información, en diccionario de internet, Monografías.com [En línea]. Disponible: http://www.monografias.com/trabajos7/sisinf/sisinf.shtml [Fecha de acceso: febrero de 2009]

[2] R. Heradio, "Metodología de desarrollo de software basada en el paradigma generativo. Realización mediante la transformación de ejemplares", Tesis Doctoral, Universidad Nacional de Educación a distancia, Escuela Técnica Superior de Ingeniería Informática. [En línea]. Disponible: http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio.html [Fecha de acceso: febrero de 2009]

[3] T. Granollers, J. Lóres, F. Perdrix. "Modelo de Proceso de la Ingeniería de la Usabilidad. Integración de la Ingeniería del Software y la de la Usabilidad", en Workshop de Investigación sobre nuevos paradigmas de interacción en entornos colaborativos aplicados a la gestión y difusión del Patrimonio cultural. Granada, España, 2002. [En línea], Disponible: http://lsi.ugr.es/~mgea/workshops/coline02/Articulos/toni.pdf [Fecha de acceso: febrero de 2009].

[4] A. M. Reina, J. Torres, M. Toro and J. A. Álvarez. "Separación de conceptos y MDA: Arquitectura de un framework*". I Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Málaga, España. 2004.

[5] J. Greenfield, K. Short, S. Cook, S. Kent. "Asembling Applications with Patterns, Models Frameworks and tools". Software Factories. Wiley Publishing Inc, 2004.

[6] P. Wegner; "Research directions in software technology", en la tercera conferencia internacional de Ingeniería de Software, 1978.

[7] S. J. Mellor, K. Scott, A. Uhl, D. Weise. "MDA Distilled: Principles of Model-Driven Architecture". Adisson Wesley. 2004.

[8] A. Brown. "An introduction to Model Driven Architecture", Personal IBM, 2004.

[9] IT Information Technology, IT. [En línea] Disponible: http://en.wikipedia.org/wiki/Information_ technology [Fecha de acceso: enero de 2009]

[10] Familias de Productos. Tema II. Master Oficial en Sistemas Telemáticos e Informáticos. Universidad Rey Juan Carlos [En línea] Disponible: http://triana.escet.urjc.es/aspf/MasterURJC-SAPF-Tema2-08-09.pdf [Fecha de acceso: Marzo 2009]

[11] J. Grieendfield, "Un caso para la fábrica de software", Enterprise Frameworks and Tools. Microsoft Corporation, 2004.

[12] Soluciones software para pequeña empresa, en diccionario de internet, Intel [En línea] Disponible: http://www.intel.com/espanol/solutions/providers/smallbusiness.htm [Fecha de acceso: Noviembre 2008]

[13] L. M. Northrop, P. C. Clements con F. Bachmann, J. Bergey, G. Chastek, S. Cohen, P. Donohoe, L. Jones, R. Krut, R. Little, J. McGregor, L. O'Brien. "A Framework for Software Product Line Practice, Version 5.0". Carnegie Mellon University. 2007. [En línea]. Disponible: http://www.sei.cmu.edu/productlines/framework.html [Fecha de acceso: febrero 2009].

[14] J. Izquierdo. "Automatización en el manejo de modelos UML de arquitectura del software". Tesis de pregrado. Universidad politécnica de Madrid. Escuela técnica superior de ingenieros de telecomunicación. 2002.

[15] G. Monte. "Derivación de modelos en UML para líneas de productos". Tesis de pregrado. Universidad politécnica de Madrid. Escuela técnica superior de ingenieros de telecomunicación. 2003.

[16] OMG Object Management Group. "MDA Guide Version1.0" [En línea]. Disponible:http://www.modelware-ist.org [Fecha de acceso: febrero de 2009].

[17] Engineering Software Architectures, Processes and Platforms for System-Families. ESAPS [En línea]. Disponible: http://www.esi.es/esaps/index.html [Fecha de acceso: febrero de 2009]
[18] Eureka a network for market oriented I+D. Eureka [En línea]. Disponible:http://www.eureka.be/about.do [Fecha de acceso: enero de 2009]
[19] Information Technology for European Advancement . ITEA [En línea]. Disponible:http://www.itea2.org/about_itea_2 [Fecha de acceso: enero de 2009]
[20] From Concepts to Application in System-Family Engineering. CAFÉ [En línea]. Disponible: http://www.esi.es/Cafe [Fecha de acceso: febrero de 2009]
[21] European Software Institute - Tecnalia. ESI - Tecnalia [En línea]. Disponible:http://www.esi.es/index.php?op=7.1 [Fecha de acceso: enero de 2009]
[22] FAct-based Maturity through Institutionalisation Lessons-learned and Involved Explartion of System-family engineering. FAMILIES [En línea]. Disponible: http://www.esi.es/Families [Fecha de acceso: enero de 2009]
[23] Software Engineering Institute Carnegie Mellon. SEI Carnegie Mellon [En línea].Disponible: http://www.sei.cmu.edu/about/. [Fecha de acceso: enero de 2009]
[24] B. González, M. A. Laguna. “MDA e Ingeniería de Requisitos para líneas de productos”. *II Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Granada, España. 2005.
[25] J. Muñoz, V. Pelechano. “MDA vs. Factorías de Software”. *II Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones.* Granada, España. 2005.
[26] Software Product Lines. Framework [En línea]. Disponible: http://www.sei.cmu.edu/productlines/framework.html. [Fecha de acceso: enero de 2009]
[27] P. Clements, L. Northrop. “Sofware product lines: practices and patterns”. Adisson Wesley. 2001. [Fecha de acceso: enero de 2009]
[28] J. Bergey, L. O'Brien, D. Smith. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. “Software Product Lines: Proceedings”. Second Software Product Line Conference. San Diego. 2002.
[29] J. P. Lorenzo, J.D. García, E. V. Sánchez, A. Estévez. “Implementación de un motor de transformaciones con soporte MOF 2.0 QVT” *II Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Granada, España. 13 de septiembre de 2005
[30] ModelWare. ModelWare [En línea]. Disponible: http://www.modelware-ist.org [Fecha de acceso: febrero de 2009]
[31] Information Society Technologies. IST [En línea]. Disponible: http://cordis.europa.eu/ist/about/about.htm [Fecha de acceso: febrero de 2009]
[32] European Conference on Model Driven Architecture - Fondations and Applications. ECMDA-FA [En línea]. Disponible: http://www.fokus.fraunhofer.de/en/motion/news _events/veranstaltungsreihen/ecmda/index.html
[33] F. J. van der Linden, K. Schmid, E. Rommes. “Software Product Lines in Action. The Best Industrial Practice in Product Line Engineering”. Springer-Verlag Berlin Heidelberg. 2007.
[34] K. Schmid and M. Verlage. “The economic impact of product line adoption and evolution”. *IEEE Software*. 2002.
[35] A. Birk, G. Heller, I. John, T. von der Maßen, K. Muller, K. Schmid. “Product line engineering: The state of the practice”. *IEEE Software*. 2003.
[36] M. E. Porter. Strategy and Society: The Link Between Competitive Advantage and Corporate Social Responsibility. *Harvard Business Review*, Diciembre de 2006, pp. 78-92.
[37] K. Pohl, G. Bockle, F. van der Linden. “Software Product Line Engineering: Foundations, Principles, and Techniques”. Springer-Verlag Berlin Heidelberg. 2005.
[38] M. F. Solarte. “AMIR-ST: propuesta de una Aproximación Metodológica para la Ingeniería de Requisitos de Sistemas Telemáticos”. *III Jornada de investigación y desarrollo en informática, TECNOCOM .* Medellín, Colombia. 2003.

[39] I. Somerville, P. Sawyer. “Requeriments Engineering: A Good Practices”. Wiley.1997.
[40] C. Mason, G. Milne. An approach for identifying cannibalization within product line extensions and multi-brand strategies. *Journal of Business Research*. 1994.
[41] K. Schmid. The product line mapping method. Technical Report 028.00/E, Fraunhofer IESE, 2000.
[42] S. Bandinelli, G. Mendieta. Domain potential analysis: Getting serious about product-lines. *In Third International Workshop on Software Architectures for Product Families*. 2000.
[43] K. Schmid. An assessment approach to analyzing benefits and risks of product lines. *In The Twenty-Fifth Annual International Computer Software and Applications Conference.* 2001.
[44] K. Schmid. A comprehensive product line scoping approach and its validation. *In Proceedings of the 24th International Conference on Software Engineering.* 2002.
[45] M. Jazayeri, A. Ran, F. van der Linden. “Software Architecture for Product Families”. Addison–Wesley, 2000.
[46] M. Fowler. “Analysis Patterns: Reusable Object Models”. Addison–Wesley. 1997.
[47] D. Bredemeyer. Software architecture workshop, course handouts. [En línea]. Disponible: http://www.bredemeyer.com/, 2002. [Fecha de acceso: abril de 2009]
[48] F. Bachmann,M. Goedicke, J. Leite, R. Nord,K. Pohl, B. Ramesh, A. Vilbig. A meta-model for representing variability in product family development. *In Proceedings in the 5th International Workshop on Product Family Engineering*. 2003.
[49] F. Bachmann and L. Bass. Managing variability in software architectures. *In ACM SIGSOFT Symposium on Software Reusabilit.* 2001.
[50] J. Bosch. “Design and Use of Software Architectures”. Addison–Wesley, 2000.
[51] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. “Refactoring: Improving the Design of Existing Code”. Addison–Wesley, 2000.
[52] A. Kleppe, J. Warmer, W. Bast. “MDA Explained: The Model Driven Architecture: Practice and Promise”. Adisson Wesley. 2003.
[53] Wikipedia Modeling Driven Architecture. [En línea]. Disponible: http://es.wikipedia.org/wiki/ Model_Driven_Architecture. [Fecha de acceso: marzo de 2009].
[54] K. Jordan; “Software reuse”, MJY Team's Software Risk Management, 1997 [En línea]. Disponible: http://www.baz.com/kjordan/swse625/htm/tp-kj.htm
[55] I. Jacobson; M. Griss; P. Johnson*;" Software reuse. Architecture, Process and Organization for Business Success"*, ACM Press. Addison Wesley. 1997.
[56] Mª E. Manso; F. J. García; M. A. Laguna “Métricas en la reutilización orientada al objeto”, 1999. [En línea]. Disponible: http://www.sc.ehu.es/jiwdocoj/remis/docs/ reutiloo.html. [Fecha de acceso: marzo de 2009].
[57] S. J. Mellor, K. Scott, A. Uhl, D. Weise. “MDA Distilled: Principles of Model Driven Architecture”. Adisson Wesley. 2004.
[58] OMG Document Number: formal/2006-01-01.Meta Object Facility (MOF) OMG Core Specification, *version 2.0*. [En línea]. Disponible: http://www.omg.org/spec/ MOF/ 2.0/PDF/ [Fecha de acceso: abril de 2009].
[59] OMG Document Number: formal/2007-12-01. MOF 2.0/XMI Mapping, *Version 2.1.1.* [En línea]. Disponible: http://www.omg.org/docs/formal/07-12-01.pdf. [Fecha de acceso: abril de 2009].
[60] OMG Document Number: formal/2008-04-03. Meta Object Facility (MOF) 2.0 Query/ View/ Transformation Specification, *Version 1.0.* [En línea]. Disponible: http://www.omg.org/docs/ formal/08-04-03.pdf. [Fecha de acceso: abril de 2009].
[61] OMG Document Number: formal/2006-05-01. Object Constraint Language OMG Specification *Version 2.0.* [En línea]. Disponible: http://www.omg.org/spec/OCL/ 2.0/PDF/. [Fecha de acceso: abril de 2009].

[62] OMG Document Number: formal/2003-03-02. Common Warehouse Metamodel (CWM) Specification. *Version 1.1.* [En línea]. Disponible: http://www.omg.org/docs/ formal/03-03-02.pdf. [Fecha de acceso: abril de 2009].
[63] OMG Document Number: formal/2009-02-04. OMG Unified Modeling Language (OMG UML), Infrastructure *Version 2.2.* [En línea]. Disponible: http://www.omg.org/ spec/UML/2.2/ Infrastructure/PDF/. [Fecha de acceso: abril de 2009].
[64] AndroMDA Cutting Edge MDSD/MDA Toolkit. AndroMDA [En línea]. Disponible: http://www.andromda.org [Fecha de acceso: abril de 2009].
[65] V. Bollati, J. M. Vara, B. Vela, E. Marcos. "Una revisión de herramientas MDA". *IV Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Granada, España. 2007.
[66] Xcarecrows Cogenit Opens Source Software Engineering Tools. Xcarecrows [En línea]. Disponible: http://www.xcarecrows.com/xcarecrows/ [Fecha de acceso: marzo de 2009].
[67] E. A. Sánchez, G. Merin, J. Muñoz. "Hacia un marco práctico de evaluación de tecnologías de transformación de modelos en la plataforma Eclipse". *IV Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Granada, España. 2007.
[68] Acceleo, Acelerate your developments Effective MDA. Acceleo [En línea]. Disponible: http://www.acceleo.org/pages/home/ [Fecha de acceso: abril de 2009].
[69] Blu age Build, Blu Age Agile Model Transformation. Blu age Build OptimalJ http://www.bluage.com/index.php?cID=blu_age_build [Fecha de acceso: marzo de 2009]
[70] MotionModeling. MotionModeling [En línea]. Disponible: http://motionmodeling.source forge.net/ [Fecha de acceso: marzo de 2009]
[71] OptimalJ, Manuales de OptimalJ Compuware. OptimalJ [En línea]. Disponible: http://www.compuware.com/products/optimalj/ [Fecha de acceso: marzo de 2009]
[72] J. García, J. Rodríguez, M. Menárguez, M. J. Ortín, J. Sánchez. "Un estudio comparativo de dos herramientas MDA: Optimal J y ArcStyler" *I Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Málaga, España. 2004.
[73] ArcStyler, Manuales de ArcStyler interactive objects. ArcStyler [En línea]. Disponible: http:// www.arcstyler.com/[Fecha de acceso: febrero de 2009]
[74] Eclipse. Eclipse [En línea]. Disponible: http://www.eclipse.org [Fecha de acceso: febrero de 2009]
[75] Aplication Development. IBM Gears Up for Modelling. [En línea]. Disponible: http://www.eweek.com/c/a/Application-Development/IBM-Gears-Up-for-Modeling/ [Fecha de acceso: marzo de 2009]
[76] Eclipse Proyect. Eclipse Proyect [En línea]. Disponible: http://www.eclipse.org/projects/project_summary.php?projectid=eclipse [Fecha de acceso: febrero de 2009]
[77] eUML2 Studio Soyatec. eUML2. [En línea]. Disponible: http://www.soyatec.com/euml2/ [Fecha de acceso: febrero de 2009]
[78] Green UML. Green UML. [En línea]. Disponible: http://green.sourceforge.net/index.html [Fecha de acceso: febrero de 2009]
[79] Visual Paradigm Smart Development Environment. Visual Paradigm SDE. [En línea]. Disponible: http://www.visual-paradigm.com/product/sde/ec/provides/ [Fecha de acceso: febrero de 2009]
[80] M. Solarte. "Arquitecturas de sistemas telemáticos". Conferencias de clase de la materia ambientes de desarrollo. Énfasis en telemática, Ingeniería Electrónica y de Telcecomuncaciones, Facultad de Ingeniería Electrónica y de Telecomunicaciones, Universidad del Cauca. Popayán, Colombia. 2007.
[81] J. Muñoz, V. Pelechano. "MDA a debate". *". I Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*. Málaga, España. 2004.
[82] S.J. Mellor, A.N. Clark, T. Futagami. Model Driven Development - Guest editor's introduction. *IEEE Software*. Septiembre - Octubre. 2003.

[83] J. Vilalta. “Artefactos UML”. Conferencias del taller de modelado de la organización Vico. 2006. [En línea]. Disponible: http://www.vico.org/aRecursosPrivats/UML_TRAD/talleres/mapas/UMLTRAD_101A/LinkedDocuments/UML_diagClases.pdf [Fecha de acceso: abril de 2009]
[84] R. Braek, O. Haugen. Engineering Real Time Systems, Prentice Hall, 1993.
[85] M. Solarte. “Conceptos básicos sobre ingeniería de sistemas telemáticos”. Conferencias de clase de la materia ambientes de desarrollo. Énfasis en telemática, Ingeniería Electrónica y de Telecomunicaciones, Facultad de Ingeniería Electrónica y de Telecomunicaciones, Universidad del Cauca. Popayán, Colombia. 2007.
[86] P. Kruchten. “Planos arquitectónicos: el Modelo de “4+1” Vistas de la Arquitectura del Software”. [En línea]. Disponible: https://www.u-cursos.cl/ingenieria/2008/1/CC61H/1/material_docente/objeto/160601 [Fecha de acceso: marzo de 2009].
[87] M. Solarte. “Arquitectura del sistema”. Conferencias de clase de la materia ambientes de desarrollo. Énfasis en telemática, Ingeniería Electrónica y de Telecomunicaciones, Facultad de Ingeniería Electrónica y de Telecomunicaciones, Universidad del Cauca. Popayán, Colombia. 2007.
[88] G. Booch. Object-Oriented Analysis and Design with Applications. Benjamin-Cummings Pub. Co., Redwood City, California, 2nd edition, 1993.
[89] Diagrama de paquetes, Lenguaje de modelado Unificado. Diagrama de paquetes. [En línea]. Disponible: http://es.wikipedia.org/wiki/Diagrama_de_paquetes [Fecha de acceso: marzo de 2009].
[90] C.Serrano. “Referencia metodológica integral para desarrollo de sistemas telemáticos”. Conferencias de clase de la materia laboratorio III de electrónica. Grupo de investigación: GIT-Grupo en Ingeniería Telemática, de la Universidad del Cauca. Popayán, Colombia. 2006.
[91] UML 2.1. Enterprise Architect versión 7.5. Enterprise Architect. . [En línea]. Disponible: http://www.sparxsystems.com/products/ea/index.html. [Fecha de acceso: marzo de 2009].
[92] A. Caramazana. “Tecnologías MDA (Model Driven Architecture) para el desarrollo el software”. *I Jornada Académica de Investigación en Ingeniería Informática*. Madrid, España. 2004.
[93] OMG Document. ISO Especification ISO/IEC/2004-03-03. XML Metadata Interchange Specification, *Version 2.0* [En línea]. Disponible: http://www.omg.org/cgi-bin/apps/doclist.pl [Fecha de acceso: mayo de 2009].
[94] OMG Document Number: formal/2007-12-01. MOF 2.0/XMI Mapping, *Version 2.1.1* [En línea]. Disponible: http://www.omg.org/docs/formal/07-12-01.pdf [Fecha de acceso: mayo de 2009].
[95] DTD, Document Type Definition. DTD. [En línea]. Disponible: http://es.wikipedia.org/wiki/DTD. [Fecha de acceso: mayo de 2009].
[96] Manual de referencia, MySQL 5.0. MySQL 5.0. En línea]. Disponible: http://dev.mysql.com/doc/refman/5.1/en/ [Fecha de acceso: mayo de 2009].
[97] D. J. Burbano. “Análisis comparativo de bases de dato de código abierto vs. código cerrado (determinación de índices de comparación)”. Quito, Ecuador. 2006
[98] SQL, Structured Query Language. SQL. En línea]. Disponible: http://es.wikipedia.org/wiki/SQL [Fecha de acceso: mayo de 2009].
[99].SAX, Simple API for XML. SAX y DOM. [En línea]. Disponible: http://www.hipertexto.info/documentos/dom.htm
[100] R. Cerón, J. L. Arciniegas, J. L. Ruiz, J. C. Dueñas, J. Bermejo y R. Capilla. “Architectural Modelling in Product Family Context”. *The First European Workshop on Software Architecture – EWSA.* St Andrews, Reino Unido. 2004.