Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

Master:
Ingeniería del Software, Métodos Formales y Sistemas de Información

ANÁLISIS Y DISEÑO DE UN GENERADOR AUTOMÁTICO DE SISTEMAS DE DIAGNÓSTICO BASADO EN LÍNEAS DE PRODUCTO

Presenta: María Eugenia Cabello Espinosa

Director: Dr. Isidro Ramos Salavert

Valencia-España, Julio de 2007

RESUMEN

Este trabajo de investigación presenta el análisis y diseño de una solución genérica para la construcción de sistemas expertos que realizan tareas de diagnóstico usando arquitecturas software orientadas a aspectos y técnicas de líneas de producto software. Esta aproximación ha seguido la iniciativa del Model Driven Architecture-MDA del Object Management Group-OMG para construir modelos de dominio (basados en Modelos Independientes de Computación-), que serán transformados automáticamente en modelos arquitectónicos PRISMA (como Modelos Independientes de Plataforma-PIM) y compilados en una aplicación ejecutable en .NET. Las técnicas de Líneas de Producto Software son utilizadas para capturar la variabilidad de esta clase de sistemas. Un Lenguaje Específico de Dominio-LSD ha sido utilizado para que la interfaz del usuario sea amigable (clara y concisa).

Palabras clave: arquitecturas software, reutilización del software, lenguaje de definición arquitecturas, aspectos, sistemas expertos, diagnóstico médico, líneas de producto, variabilidad, modelos conceptuales, ingeniería del dominio, ingeniería de la aplicación.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1

INTRODUCCIÓN

El trabajo presentado en esta tesis es una aproximación al desarrollo de sistemas de diagnóstico basado en líneas de producto. Para el desarrollo de este trabajo, se ha considerado la iniciativa MDA (Model Driven Architecture) del OMG (Object Management Group) para la construcción de modelos de dominio y su transformación automática en una aplicación ejecutable, así como SPL (Software Product Line) para captar la variabilidad de dichos sistemas.

Dicha aproximación es denominada BOM (Base-Line Oriented Model Diagnosis): un "generador automático de sistemas de diagnóstico basado en líneas de producto", el cual obtiene, de forma automática, un producto específico en una línea de productos software a partir de una serie de modelos previos.

BOM es un "framework" que captura en modelos conceptuales la información del diagnóstico y del dominio específico, utilizando

técnicas de MDA, para generar automáticamente sistemas de diagnóstico con arquitecturas PRISMA, basados en líneas de producto.

La estructura de este capítulo es la siguiente: en la sección 1 se presenta el planteamiento del problema y la justificación de este trabajo de investigación. La sección 2 contiene la hipótesis y los objetivos de esta tesis. En la sección 3, se comenta brevemente el contenido de los capítulos restantes, así como los apéndices que conforman esta tesis.

1.1 Planteamiento del problema y justificación del trabajo

Es notable el interés que han cobrado los sistemas que realizan tareas de diagnóstico en los últimos años. El principal objetivo de este tipo de sistemas es el de identificar el estado (puede ser un problema o disfunción) de una entidad a partir de una serie de datos (variables observables). Han sido muchos los dominios de interés que han servido de escenario para el desarrollo y la aplicación de los sistemas de diagnóstico, entre los que se encuentran: agricultura, derecho, electrónica, mecánica, educación y medicina. Al respecto, es conveniente resaltar que este tipo de aplicaciones han ido adquiriendo una mayor importancia a la vez que una mayor complejidad, y por consiguiente, existe la necesidad de mejorarlas.

Por otro lado, para captar los complejos requisitos software surge el entorno PRISMA [Pérez, 2006], que aporta una expresividad de

Arquitecturas Software con Aspectos a un alto nivel de abstracción y que presenta propiedades y ventajas en la construcción de modelos arquitectónicos, a través de su potencia expresiva, facilidad de uso y novedad. De esta manera se cubre el hueco existente en el modelado de sistemas software altamente reconfigurables y reutilizables dentro de un marco de calidad controlada. El modelo arquitectónico PRISMA integra dos aproximaciones: el Desarrollo de Software Basado en Componentes [Szyperski, 1998] y el Desarrollo de Software Orientado a Aspectos [AOSD]. Esta integración se consigue definiendo los elementos arquitectónicos mediante aspectos. De esta forma, el modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las propiedades necesarias de cada uno de ellos. PRISMA presenta un enfoque integrado y flexible para describir modelos arquitectónicos complejos, distribuidos y reutilizables, que puede ser aplicado al dominio del diagnóstico.

Además el desarrollo de sistemas complejos se está complicando cada vez más, debido a una serie de factores como la aparición de nuevas tecnologías (Internet e intranet), interconexión de múltiples sistemas y plataformas, la necesidad de integrar viejos sistemas aun válidos (“Legacy Systems”), la adaptación personalizada del software a cada tipo de usuario, las necesidades específicas de un sistema y las diferentes plataformas de implementación. Esta situación nos lleva a necesitar múltiples versiones de la misma o parecida aplicación cada poco tiempo. En consecuencia, la Ingeniería del Software debe proporcionar herramientas y métodos que permitan poner en marcha métodos de desarrollo no para un

único producto sino para una familia de productos con distintas capacidades y adaptables a situaciones variables.

Por esto, las líneas de producto de software (LPS) [Clements et al., 2002] surgen como un esfuerzo para controlar y minimizar los costos sorprendentes de desarrollo de software. Esta aproximación se basa en la creación de un diseño que comparten todos los miembros de una familia de programas (líneas de producto). De esta manera, un diseño hecho explícitamente se beneficia del software común y puede ser usado en diferentes productos mediante la adición de distintas características ("*features*"). Con ello se reduce sustancialmente los gastos generales y el tiempo para construir nuevos productos.

Cabe mencionar que el desarrollo de las LPS es, desde el punto de vista práctico, el enfoque que mayores éxitos ha alcanzado en el campo de la reutilización del software, gracias a la combinación de un desarrollo sistemático y a la utilización de componentes reutilizables de grano grueso. Las líneas de productos software permiten la reutilización sistemática en los casos en los que se tienen familias de productos, es decir productos similares, diferenciados por algunas características. Promoviendo de este modo la industrialización del desarrollo software.

Uno de los elementos clave para una LPS es la representación y gestión de la variabilidad. En este contexto, se encuentra la iniciativa de la MDA [MDA] en la construcción de modelos de dominio (expresados como grafos de características o "*features*") y su posible transformación en modelos arquitectónicos. La novedad que propone MDA es la posibilidad de automatizar la transformación que especifica cómo se convierten las instancias

del modelo de características [Batory et al., 2006] en una aplicación ejecutable.

Tomando en cuenta lo anteriormente expuesto, y ante la necesidad de:

- crear sistemas de diagnóstico en diferentes dominios,
- minimizar los costos de producción reutilizando "paquetes de software",
- generar código automáticamente con la finalidad de incrementar la productividad y la calidad, y disminuir el "time to market",
- construir un sistema de una forma sencilla utilizando las ontologías de diagnóstico y del dominio que faciliten la comunicación hombre-máquina, y
- desarrollar sistemas independiente de plataforma y que sean abordados desde la perspectiva del problema y no de la solución, permitiendo generalidad a la aproximación desarrollada y aplicabilidad a diversos dominios,

se ha decidido construir el "Generador automático de sistemas de diagnóstico basado en líneas de producto".

Con ello se pretende mejorar el desarrollo de los sistemas de diagnóstico de varias formas:

- Utilizando las ventajas de los sistemas expertos, al incorporar múltiples estrategias de razonamiento, para que ante la solución de un problema, se apliquen las adecuadas que permitan resolver el problema de forma más eficiente.
- Aplicando técnicas de las líneas de producto de software, al crear un diseño que comparten todos los miembros de

una familia de programas; de esta manera, con un diseño hecho explícitamente se construye al software común y puede ser usado en diferentes productos, reduciendo costos, tiempo, esfuerzos y complejidad en su construcción.

- Construyendo las arquitecturas de la línea de productos en el marco del modelo PRISMA, proporcionando las ventajas de los sistemas distribuidos, facilitando la reutilización y gestionando la complejidad.
- Aplicando técnicas de MDA, al implementar sistemas que pueden ser integrados en diferentes plataformas, y al automatizar las transformaciones de las instancias del modelo de características del diagnóstico en una aplicación ejecutable, generando su código automáticamente.
- Utilizando transformaciones XSLT en documentos XML, considerando que es una buena alternativa en el campo de las transformaciones, ya que al estar basada en estándares aprovecha la reutilización de tecnología.

1.2 Hipótesis y objetivos de la tesis

1.2.1 Hipótesis

Con base en lo anteriormente descrito, la hipótesis planteada en esta tesis expresa que:

"Es factible generar de forma automática sistemas que realizan tareas de diagnóstico basados en líneas de producto, utilizando técnicas del MDA (niveles de abstracción de modelado y transformación de modelos) y de las líneas de producto".

1.2.2 Objetivos

1.2.2.1 Objetivo general

El objetivo general de esta tesis, derivada de la hipótesis, es:
"Analizar y diseñar un software que permita la generación automática de sistemas de diagnóstico basado en líneas de producto".

1.2.2.2 Objetivos específicos

Para la realización del objetivo general se plantearon los siguientes objetivos específicos:

1.- Modelar los requisitos de los sistemas de diagnóstico, mediante Modelos Independientes de Computación-CIM, capturando en modelos conceptuales la información del diagnóstico y de los dominios específicos, a través de técnicas de líneas de producto de software (en particular el paradigma de la Programación Orientada a Características-FOP) y técnicas de transformación, utilizando BOM. De esta manera seleccionar los esqueletos o plantillas de los elementos arquitectónicos que conforman la base de los productos de la línea de productos software de diagnóstico (LPSD), añadiendo las características adecuadas para crear los tipos PRISMA correspondientes. Se hará uso de la herramienta XAK para realizar transformaciones XSLT en documentos XML (esqueletos y tipos PRISMA especificados en el ADL de PRISMA).

2.- Representar la funcionalidad y estructura de los sistemas de diagnóstico abstrayéndose de los detalles tecnológicos de la plataforma de implementación, mediante Modelos Independientes de Plataforma-PIM, utilizando la herramienta PRISMA-CASE para configurar la arquitectura específica del producto de la LPSD.

3.- Combinar las especificaciones contenidas en el PIM con los detalles de la plataforma de implementación, mediante Modelos Específicos de Plataforma-PSM, para generar el código automáticamente (sobre .NET) con la herramienta PRISMA-MODEL-COMPILER y finalmente ejecutar el sistema con el middleware PRISMA-NET.

1.3 Estructura de la tesis

Los capítulos restantes de esta tesis y sus anexos, están organizados de la siguiente manera:

Capítulo 2.- Estado del arte
Este capítulo consta de dos partes. En la primera parte se presenta un panorama del estado del arte relacionado con la temática de esta tesis: se comenta el papel de los sistemas expertos que realizan tareas de diagnóstico, las técnicas del MDA y de las líneas de producto de software aplicadas en esta tesis, así como del modelo arquitectónico PRISMA. En la segunda parte se mencionan algunos trabajos relacionados con la temática de esta tesis.

Capítulo 3.- Análisis de la aproximación MDA para líneas de producto software orientadas al diagnóstico.

Este capítulo presenta un análisis de la propuesta de este trabajo de investigación: las LPS en el diagnóstico, la aproximación para los procesos de ingeniería de la LPS, la variabilidad, una metodología para construir arquitecturas software en sistemas de diagnóstico a partir de los requisitos software, y el análisis del dominio del diagnóstico y del dominio de aplicación.

Capítulo 4.- Desarrollo de la aproximación MDA para líneas de producto software orientadas al diagnóstico.
Este capítulo presenta el desarrollo de la propuesta de este trabajo de investigación, describiendo la metodología realizada para el desarrollo BOM: un *framework* generador automático de sistemas de diagnóstico basado en líneas de producto.

Capítulo 5.- Conclusiones y trabajos futuros
Las principales aportaciones de este trabajo de investigación son presentadas en este capítulo. Así mismo son propuestos los trabajos de investigación de futuro en el marco de esta tesis.

Apéndice A.- Conceptos utilizados en el modelado de la LPSD
En este apéndice se definen los conceptos que conforman la ontología del diagnóstico.

Apéndice B.-Estudio de campo
Este apéndice presenta el estudio de campo realizado en la presente tesis, abarcando una breve descripción de cinco casos de estudio: un diagnóstico educativo, un diagnóstico televisivo, un diagnóstico para otorgar becas, un diagnóstico de desastres y un diagnóstico médico.

Apéndice C.-Un caso de estudio: el diagnóstico televisivo

Este apéndice contiene la descripción en detalle de uno de los casos de estudio abordados en esta tesis. En él se presenta el desarrollo de un sistema de nuestra línea de productos que realiza un diagnóstico televisivo para diagnosticar el estado de un video y con ello tomar la decisión de si debe o no debe ser transmitido al aire.

Capítulo 2

ESTADO DEL ARTE

En este capítulo se presenta el estado del arte de las tendencias actuales más importantes en investigación sobre los sistemas expertos en tareas de diagnóstico, la arquitectura dirigida por modelos, las líneas de producto software, la transformación de modelos y las arquitecturas PRISMA. Enmarcadas en esta presentación, se incluyen referencias al trabajo realizado en estas temáticas.

Asimismo, en este capítulo se presentan algunos trabajos relacionados con la temática de esta tesis.

2.1 ESTADO GENERAL DEL ARTE

2.1.1 Sistemas de diagnóstico

2.1.1.1 Sistemas expertos

Una vez que se ha detectado un problema, se tiene que tomar una decisión (y aunque suene ilógico, inclusive la de no hacer nada). Se elige una alternativa que parezca suficientemente racional, y que permita incrementar o no, el valor esperado después de que se haya resuelto el problema. Posteriormente se emite un plan de control que servirá de guía en la toma de decisiones, inclusive en decisiones con respecto a modificar dicho plan.

A veces se considera la toma de decisiones como la parte que se realiza desde que se tienen las conductas alternativas generadas hasta que se realizan la elección de la acción a llevar a cabo. Pero otras veces se considera que todo el proceso está incluido en la toma de decisiones.

[Hastie, 2001] plantea una serie de definiciones que sirven para aclarar el proceso de toma de decisiones, que es una parte de la resolución de problemas, entre éstas se encuentran:

Decisiones.- son combinaciones de situaciones y conductas que pueden ser descritas en términos de tres componentes esenciales: acciones alternativas, consecuencias y sucesos inciertos.

Tomar una decisión.- se refiere al proceso entero de elegir un curso de acción.

Según estas definiciones el proceso de toma de decisiones sería encontrar una conducta adecuada para una situación en la que hay una serie de sucesos inciertos. La elección de la situación ya es un elemento que puede entrar en el proceso. Hay que elegir los elementos que son relevantes y obviar los que no lo son y analizar las relaciones entre ellos. Una vez determinada cual es la situación, para tomar decisiones es necesario elaborar acciones alternativas, extrapolarlas para imaginar la situación final y evaluar los resultados teniendo en cuenta la incertidumbre de cada resultado y su valor. Así se obtiene una imagen de las consecuencias que tendría cada una de las acciones alternativas que se han definido. De acuerdo con las consecuencias se asocia a la situación la conducta más idónea eligiéndola como curso de acción.

Existen diferentes tipos de Sistemas de Apoyo a las Decisiones:

1 Sistema de Soporte a la Toma de Decisiones (DSS)
2 Sistemas de Información para Ejecutivos (EIS)
3 Sistemas para la Toma de Decisiones en Grupo (GDSS)
4 Sistemas Expertos para la Toma de Decisiones (EDSS)

Los sistemas de soporte a la toma de decisiones (del término inglés Decision Support Systems-DSS) han ayudado a los ejecutivos a tomar decisiones dentro de las organizaciones desde los años 60`s. Los DSS son un bloque de toma de decisiones sustentado en base de datos, que puede ser usado por quienes toman las decisiones para apoyar el proceso de decidir. [Little, 1970] define los DSS como un modelo basado en un conjunto de

procedimientos para procesar datos y juicios que ayudan a los administradores a tomar decisiones. [Bonczek et al., 1980] definen los DSS como un sistema basado en computadoras con tres componentes clave: lenguaje de sistema (un mecanismo para proveer comunicación entre el usuario y otros componentes del DSS), un sistema de conocimiento (repositorio de conocimiento sobre problemas, datos o procesos) y el sistema procesador de problemas (la unión entre los otros dos componentes). En nuestros días la toma de decisiones es uno de los procesos más valorados en cualquier empresa.

De los diferentes tipos de sistemas de apoyo a las decisiones, los sistemas basados en el conocimiento o sistemas expertos darán mayor soporte en el proceso de toma de decisiones, permitiendo tener el conocimiento del experto almacenado en una base de conocimiento y utilizarlo cuando se requiera sin que esté él presente. Los sistemas expertos que están basados en reglas, contienen conocimientos predefinidos que se utilizan para tomar todas las decisiones.

Por ello, los sistemas expertos empiezan a tener cada vez mayor auge, hasta el punto de suponer un punto de referencia importante en la toma de decisiones. En realidad, incluso se podría decir que el límite de las aplicaciones, objeto de los sistemas expertos, está en la imaginación humana, siendo siempre de utilidad allí donde se necesite un experto.

Han sido propuestas hasta la fecha, diferentes definiciones de sistemas expertos. Sin embargo un fundamento común puede ser extraído de todas ellas: un sistema experto es un sistema con pericia en la solución de problemas; esto es, un sistema que posee

razonamientos, habilidades y conocimientos acerca de un dominio particular, para resolver los problemas de forma similar a la de un experto humano.

Los primeros éxitos en los sistemas expertos surgieron a mediados de las décadas de los 60`s. Joshua Lederberg, profesor de genética de la Universidad de Stanford, realizó un programa para enumerar todas las posibles configuraciones permitidas de un conjunto de átomos. Él llamó a su programa DENDRAL. Posteriormente a principios de los 80`s los sistemas expertos salen del ámbito de las universidades y de los laboratorios de investigación para infiltrarse con una tecnología comercial muy prometedora y solucionar gran cantidad y variedad de problemas. A principios de la década de los 90`s los sistemas expertos se emplean con bastante éxito en problemas de diagnóstico de fallos, interpretación de datos, predicción de comportamientos, planeación de producción, monitoreo de sistemas, el diseño, la depuración, la reparación, la instrucción y control de procesos, entre otros.

Cualquier dominio de aplicación que requiera de la pericia humana para la solución de problemas se convierte, de hecho, en un escenario probable para la aplicación de los sistemas expertos. Por ello, han sido muchas las áreas o dominios de interés que han servido de escenario para el desarrollo y la aplicación de los sistemas expertos. Dentro de estas áreas de aplicación destacan las siguientes: agricultura, ciencias militares, control de procesos, derecho, electrónica, física, geología, ingeniería, matemáticas, medicina, meteorología, química, sistemas software, finanzas y gestión, educación, administración, transportes, aeronáutica, agricultura, arqueología, derecho, geología, industria electrónica y telecomunicaciones.

Las metodologías y aplicaciones desarrolladas de los sistemas expertos forman una amplia categoría de productos de investigación, ofreciendo sugerencias y soluciones a dichos sistemas en dominios específicos. Dichas metodologías y aplicaciones están diversificadas según los antecedentes de los autores, su experiencia, sus intereses de investigación, sus habilidades en la metodología utilizada y el dominio del problema. Un estudio realizado por [Liao, 2005] examina y contempla las metodologías de los sistemas expertos, clasificándolos en 11 categorías.

Asimismo, los sistemas expertos han sido implementados en diversos paradigmas de programación, como la estructurada, la lógica y la orientada a objetos, entre otros, existiendo una tendencia en su desarrollo hacia lenguajes de cuarta generación y métodos de programación visual para dar un ambiente y una estructura amigable de comunicación con el usuario [Liao, 2003].

Sin embargo, conforme a lo consultado en la literatura, la arquitectura de estos sistemas se ha realizado con una estructura basada en componentes [Giarratano et al., 2004] sin considerar adicionalmente la orientación a aspectos, como lo contempla el modelo propuesto en este trabajo, que ha integrado estos dos enfoques.

La arquitectura de estos sistemas se refiere a los componentes o módulos que constituyen parte del mismo, independiente del dominio. Específicamente cuando se habla de sistemas basados en el conocimiento se definen tres tipos de arquitecturas en general:

1 Arquitecturas de primera generación, en donde el control y el conocimiento están centralizados. Estas arquitecturas propiciaron el origen del término Sistema Basado en el Conocimiento.

La arquitectura básica de los sistemas basados en el conocimiento o sistemas expertos es la presentada en la figura 1.

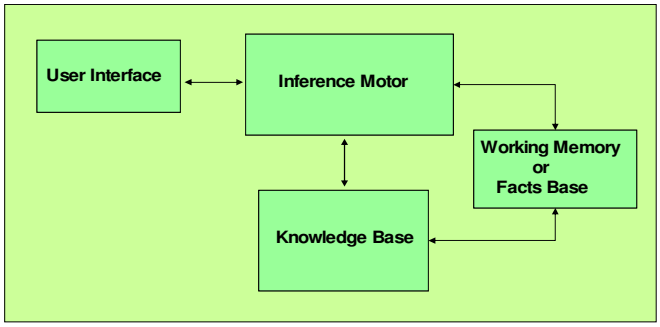


Figura 1. Arquitectura básica de un Sistema Experto

La base de conocimientos, el motor de inferencia, la memoria de trabajo o base de hechos, y la interfaz del usuario, son los componentes principales de la arquitectura de todo sistema experto basado en reglas.

Estos elementos son independientes unos de otros y forman unidades separadas. Los datos están agrupados en la memoria de trabajo (almacén temporal de información dinámica). Se utiliza la representación del conocimiento de modo deductivo mediante reglas de la forma IF <premisa > THEN <conclusión>, que forma la base de conocimientos. El control es independiente y es realizado por el motor de inferencia, al realizar procesos de

inferencia a través de estrategias de razonamiento. La entrada y salida de información del sistema se realiza a través de la interfaz del usuario.

2 Arquitecturas de segunda generación, guiada principalmente por la filosofía de sistemas distribuidos. Se habla de agentes inteligentes, en donde cada uno de ellos presenta un comportamiento inteligente. Surge el término Sistema de Conocimiento y Sistemas de Agentes Inteligentes.

3 Arquitecturas de última generación, en donde la idea básica es la reutilización de muchos de los componentes del sistema. Una arquitectura de este tipo, es la que se propone en esta tesis, en la que se dice que un sistema está formado por cinco elementos básicos [Fresnel et al, 1999; Calad, 2001; Studer et al, 2001]:
   - Una tarea que define el problema que debería ser solucionando por el SBC
   - Un método de solución de problemas que define su proceso de razonamiento
   - Un modelo de dominio que describe el conocimiento de su dominio
   - Una ontología que provee la terminología usada en las tareas, los métodos de solución de problemas y el dominio.
   - Adaptadores para establecer la relación apropiada entre la tarea, el método de solución del problema y del dominio. Se define un adaptador para la relación entre dos componentes de la arquitectura.

Un sistema basado en el conocimiento o sistema experto es un programa software que permite simular el comportamiento de un especialista humano frente a un problema de su competencia en un determinado campo. Estos sistemas intentan codificar los conocimientos y reglas de decisión de los especialistas, de manera que se pueden aprovechar estas pericias al tomar las decisiones. Estos sistemas están orientados esencialmente a ciertos tipos de trabajos limitados conceptualmente a una serie de acciones o decisiones.

El sistema experto es la forma más demostrativa de la informática; por concepto es un programa de computación basado en el conocimiento, que emula a un experto humano en la resolución de un problema significativo de un dominio específico. En los sistemas expertos se permite la interacción del usuario con la computadora. Un sistema experto puede ser considerado como un programa inteligente.

La necesidad del uso de estos sistemas se debe por un lado al aumento constante de la cantidad de conocimientos y datos a considerar en una decisión, y por otro lado a la capacidad de estos sistemas en manejar una gran cantidad de conocimientos y emular el razonamiento humano para tomar una decisión o llegar a una conclusión.

En la estructura de un sistema experto se pueden distinguir dos partes fundamentales: la base de conocimientos y el mecanismo de inferencias. La base de conocimientos está formada por los conocimientos que se usan para resolver los problemas que se plantean. El mecanismo de inferencia se refiere a la estrategia que usa el sistema para el manejo de los conocimientos que le

permitan llegar a resultados satisfactorios. Los expertos en el dominio del diagnóstico, deben tener bien claro la definición de las variables y las reglas utilizadas en la simulación del proceso del diagnóstico.

Una de las formas más utilizadas para representar el conocimiento es a través de un conjunto de reglas del tipo “si-entonces”. En la parte del “si” están expresadas las premisas de alguna situación, en tanto que en la parte del “entonces” se encuentra la conclusión.

La estructuración del conocimiento como sistemas de si-entonces, permite ordenar el conocimiento como árboles virtuales, en los que la base se encuentra formada por las conclusiones terminales y las premisas son hojas de diferentes ramas. La complejidad del árbol resulta de la interconexión dinámica de las diferentes ramas y del número de conclusiones posibles. Los árboles de decisión, son una secuencia de pasos en los cuales se selecciona un camino a través de una cadena de eventos y acciones.

En la actualidad existen una gran cantidad de sistemas expertos utilizados en diversos ámbitos. Algunos ejemplos de sistemas expertos que actualmente existen o que se encontraban en desarrollo desde el año 2004 son:

1 Desarrollo de un sistema experto para diferenciar los dolores de cabeza de la migraña [Kopec et al., 2004],
2 Sistema experto para la evaluación de campos visuales automáticos del proceso de descomprensión neuronal óptico [Feldon, 2004],
3 Sistema de soporte a la decisión clínica en psiquiatría en la información de la edad [Kot04],

4 Desarrollo de biomarcadores basados en estereotipos metabólicos dependientes de la dieta: producto resultante en el desarrollo del sistema experto basado en la clasificación de modelos en estudios metabólicos [Shi, 2004],
5 Manejo de la interacción en el razonamiento de reglas de producción fuzzy [Yeung, 2004],
6 Sistema de soporte a la decisión médica y el concepto de contexto [Karlsson et al., 2004],
7 Sistema experto médico desarrollado en J. MD, un shell de sistema experto basado en Java: aplicación en laboratorios clínicos [Van Hoof et al., 2004],
8 Detección asistida por computadora de nódulos pulmonares [Marten et al., 2005],
9 Sistemas expertos para guiar una población de pacientes de cuidado primario para dejar de fumar, comer saludable, prevenir cáncer de piel y recibir mamografías regulares [Prochaska et al., 2005],
10 Soporte a la decisión asistido por computadora para el diagnóstico y tratamiento de enfermedades infecciosas en unidades de cuidado intensivo [Schurink et al., 2005],
11 Inteligencia artificial- la base de conocimiento aplicada a la nefrotología [San05],
12 La enfermedad de Alzheimer es sustancialmente prevenible en los Estados Unidos: reseña de factores de riesgo, terapia y los prospectos para un sistema experto [Jansson, 2005],
13 Desarrollo del modelo EASE [Tickner et al., 2005],
14 Valuación de los riesgos de la salud en gente adulta usando un sistema experto: un estudio piloto [Iliffe et al., 2005],
15 SEA: Sistema Experto de Algodón [CORPOICA, 2006],

16 MAS LECHE: software de apoyo al ganadero en la toma de decisiones de tipo técnico y administrativo en empresas ganaderas de leche especializada [CORPOICA, 2006].

17 Manejo experto en praderas.- software utilizado como herramienta para la toma de decisiones en el manejo y administración de praderas con ganado bovino [CORPOICA, 2006]

18 y muchos más que pueden ser consultados en: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?CMD=search&DB=pubmed.

2.1.1.2 Los sistemas expertos en el dominio del diagnóstico

El diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a partir de una serie de datos (variables observables o propiedades). Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una cuestión a partir de datos observables y razonamientos.

La integración de los dominios donde se realiza un diagnóstico con la informática, ha permitido extender la aplicación de los sistemas software. Vista la informática como una herramienta, puede ofrecer ayuda a las personas que realizan un diagnóstico. Existe una emulación entre el pensamiento de las personas que realizan un diagnóstico y el simulado por la informática, que al ser más rápido y objetivo hace que la intuición del pensamiento trate de encontrar

formalizaciones del pensamiento, que logren encontrar soluciones mucho más objetivas y evidentes para la solución de los problema.

La característica principal de la información que se suministra al ordenador para los procesos de la informática, es la exactitud en la descripción de la realidad objetiva. Esto requiere un esfuerzo para desarrollar el pensamiento de los expertos en el diagnóstico en aras de una mejor descripción de la realidad objetiva, donde se trata de sustituir la intuición y las inferencias por procesos lógicos o matemáticos. La cantidad y la calidad de la información suministrada al ordenador, determinan la calidad y el desarrollo del producto informático.

En particular, las personas que realizan diagnósticos, por lo general se basan en ciertos tipos de razonamientos para llegar a un diagnóstico Simulando estas estrategias, los sistemas basados en reglas utilizan uno o varios tipos de razonamientos, entre los que se encuentran: el razonamiento progresivo o deductivo (donde se parte de la premisa de las variables observables para concluir el diagnóstico), el razonamiento regresivo o inductivo (que parte del hecho de que una entidad tiene cierto diagnóstico para inferir las variables observables), el razonamiento diferencial (para discernir entre dos o más posibilidades diagnósticas e inferir el resultado del diagnostico), el razonamiento oportunístico (que es una combinación de los razonamientos deductivo e inductivo), el razonamiento basado en casos (que guarda una bitácora de la forma en que se realizaron diagnósticos anteriores para ser aplicado al caso actual), y el razonamiento probabilística (que utiliza la teoría bayesiana para llegar a un diagnóstico). Por otra parte, los sistemas basados en redes neuronales resultan muy útiles para problemas de diagnóstico, sobre todo cuando el

diagnóstico es fuertemente dependiente del reconocimiento de patrones.

La aplicación de los sistemas expertos ha sido extensa en muchos ámbitos de la vida humana, sin embargo se ha mostrado un notable interés en la ciencia biomédica y la práctica clínica. Por ello, la construcción de los sistemas expertos para resolver tareas de diagnóstico en el campo de la medicina es muy amplia, como puede observarse en [Liebewitz, 1998].

En lo referente al diagnóstico médico, existe una serie de aplicaciones extensa en número, pero quizá la más conocida, a la vez que la más antigua, podría ser MYCIN [Mycin]. El cual es el primer sistema experto que llegó a funcionar con la misma calidad que un experto humano, dando a su vez explicaciones a los usuarios sobre su razonamiento. Antes del desarrollo de MYCIN (mediados de los 70), se criticaba a la Inteligencia Artificial por resolver únicamente problemas "de juguete", sin embargo, el éxito de MYCIN demostró que la tecnología de los Sistemas Expertos estaba suficientemente madura como para salir de los laboratorios y entrar en el mundo comercial. MYCIN es, en definitiva, un sistema de diagnóstico y prescripción en medicina, altamente especializado, diseñado para ayudar a los médicos a tratar con infecciones de meningitis.

Por otra parte, el campo del diagnóstico abarca otras aplicaciones además de las médicas (si bien pueden ser estas últimas las más conocidas). En este caso se trata de fallos, averías o anomalías que se producen generalmente en un ente.

2.1.2 Arquitectura Dirigida por Modelos

2.1.2.1 Desarrollo de Software Dirigido por Modelos

En la actualidad existen dos iniciativas principales y diferentes del Desarrollo de Software Dirigido por Modelos (DSDM) (Model Driven Development-MDD). Una de ellas es el enfoque de la OMG, a través de la estrategia MDA [OMG, 2003], y la otra iniciativa es la de Microsoft, a través de Software Factories-SF y Domain Specific Languages-DSL [Greenfield et al., 2004].

El DSDM constituye una aproximación para el desarrollo de sistemas software basada en la separación de la especificación de la funcionalidad del sistema de su implementación en plataformas específicas. DSDM persigue elevar el nivel de abstracción dándole una mayor importancia al modelado conceptual y al papel de los modelos en el desarrollo de software.

En el paradigma DSDM, los modelos constituyen los elementos centrales en el desarrollo de software, al contrario que en el enfoque tradicional que está más centrado en la implementación. Los modelos se expresan usando conceptos menos limitados que los utilizados en los lenguajes de programación, y la implementación del sistema es independiente de la tecnología utilizada. Con ello los modelos son fácilmente especificados y su mantenimiento es más sencillo, sirviendo de guía para el desarrollo del sistema mediante la tecnología aplicada. Uno de los objetivos de DSDM es automatizar todo ese proceso de desarrollo. Los modelos permiten capturar el conocimiento y la lógica del negocio en modelos independientes de la tecnología utilizada para

implementar los sistemas, con el fin de proteger la inversión frente a cambios y evoluciones en la tecnología.

La investigación relacionada con los estándares de la propuesta MDA, en el contexto del DSDM, supone uno de los retos actuales más importantes en el ámbito de la Ingeniería del Software. La importancia de la propuesta MDA, se manifiesta en importantes consultoras, como Butler, que en un informe del año 2003 [Butler Group, 2003] declara que MDA se consolidará a lo largo de esta década como la principal tecnología para el desarrollo de software. Es razonable pensar que las empresas de desarrollo de software tendrán que adaptar sus procesos a las tendencias DSDM en un futuro inmediato, y para mejorar su competitividad deberán de aprovechar los beneficios que aportan MDA o las Software Factories.

En el paradigma del Desarrollo de Software Dirigido por Modelos, los modelos de alto nivel de abstracción son transformados en artefactos de implementación. Tales transformaciones son denominadas exógenas, que relacionan modelos expresados en diferentes lenguajes. De esta manera, el desarrollo del software cambia de la implementación del código al modelado.

Existen otro tipo de transformaciones: las endógenas, que son usadas para sintetizar programas de LPS a partir de las características. Una alternativa es usar transformaciones exógenas con el fin de generar transformaciones endógenas, incrementando el nivel de abstracción en las características [Trujillo, 2007].

2.1.2.2 MDA

A finales del año 2000, Object Management Group-OMG define un marco de trabajo nuevo denominado: arquitectura dirigida por modelos (Model Driven Architecture-MDA), i.e. una arquitectura para el desarrollo de software dirigido por modelos [Miller et al. 2001].

OMG es un consorcio internacional, creado en 1989, dedicado al mantenimiento y establecimiento de diversos estándares para un amplio rango de tecnologías, con el fin de potenciar el desarrollo de aplicaciones distribuidas orientadas a objetos Esta organización ha definido estándares importantes como:
- UML: empleado para la definición de los modelos independientes de la plataforma y los modelos específicos de las plataformas de destino. Es un estándar para el modelado introducido por OMG.
- MOF: establece un marco común de trabajo para las especificaciones de OMG, a la vez que provee de un repositorio de modelos y metamodelos.
- XMI: permite transformar modelos UML en XML para poder ser tratados automáticamente por otras aplicaciones.
- CWM: define la transformación de los modelos de datos en el modelo de negocio a los esquemas de base de datos.

OMG define "*object management*" como el desarrollo software que modela el mundo real mediante su representación como objetos. Estos objetos no son más que la encapsulación de atributos, relaciones y métodos de componentes software reutilizables.

La nueva iniciativa de OMG provee un nuevo enfoque en el desarrollo de software, basado en el uso de modelos. El interés y la

importancia del modelado en el desarrollo de cualquier tipo de software, es debido a la facilidad que ofrece un buen diseño tanto a la hora de desarrollar software, como al hacer la integración y mantenimiento de los sistemas de software.

OMG integra diferentes especificaciones y estándares definidos por la misma organización con la finalidad de ofrecer una solución a los problemas relacionados con los cambios en los modelos de negocio, la tecnología y la adaptación de los sistemas de información a los mismos.

MDA nos permite el despliegue de aplicaciones empresariales, diseñadas sin dependencias de plataforma y expresado su diseño mediante el uso de UML y otros estándares, potencialmente en cualquier plataforma existente, abierta o propietaria, como servicios Web, .Net, Corba, J2EE, entre otras.

La clave del MDA es la importancia de los modelos en el proceso de desarrollo de software. MDA propone la definición y uso de modelos a diferente nivel de abstracción, así como la posibilidad de la generación automática de código a partir de los modelos definidos y de las reglas de transformación entre dichos modelos.

La característica fundamental de MDA es la definición de modelos formales como ciudadanos de primera clase para el diseño e implementación del sistema y la definición de transformaciones de un modelo a otro, de tal forma que estas transformaciones puedan ser automatizadas. MDA propone la especificación del sistema separando la especificación de la lógica del mismo de los detalles de su implementación en una plataforma concreta. Así, los estándares de MDA permiten que la lógica de los sistemas,

expresada en modelos, sea realizada en diversas plataformas específicas a través de la definición de transformaciones o aplicaciones “mappings”. De este modo, MDA proporciona una manera de salvar los problemas de integración de los sistemas actuales con las nuevas plataformas que surgen para su implementación, a la vez que le permite preservar la lógica de sus aplicaciones.

2.1.2.3 Modelos en MDA

En MDA la especificación de las aplicaciones y la funcionalidad de las mismas se expresan en un modelo independiente de la plataforma que permite una abstracción de las características técnicas específicas de las plataformas de despliegue. Mediante transformaciones y trazas aplicadas sobre un modelo independiente de la plataforma, se consigue la generación automática de código específico para la plataforma elegida, lo que proporciona una independencia entre la capa de negocio, y la tecnología empleada. De esta manera, es mucho más simple la incorporación de nuevas funcionalidades o cambios en los procedimientos de negocio, sin tener que llevar a cabo los cambios en todos los niveles del proyecto. Simplemente se desarrollan los cambios en el modelo independiente de la plataforma, y éstos se propagarán a la aplicación, consiguiendo reducir considerable del esfuerzo en el desarrollo, minimizar errores que tienden a producirse en los cambios introducidos en las aplicaciones mediante otros métodos de desarrollo, reducir los costes, y aumentar la productividad.

Es por ello que MDA ha definido tres niveles conceptuales de modelado, en diferentes niveles de abstracción. Estos pueden ser modelos formales, y por lo tanto, entendibles por el ordenador. Cabe señalar que el sistema implementado se corresponde directamente con estas ideas.

En el primer nivel de abstracción, se modelan los requisitos del sistema mediante Modelos Independientes de Computación (Computer Independent Model-CIM) que sirven de puente entre los expertos del dominio y los desarrolladores que afrontan la realización del sistema. Los CIM no representan detalles del sistema, sino del dominio de aplicación del mismo.

En segundo nivel de abstracción, se encuentran los Modelos Independientes de Plataforma (Platform Independent Model-PIM) que sirven para la representación de la funcionalidad y la estructura del sistema, aislándose de los detalles tecnológicos de la plataforma de implementación. Con los PIM se desarrollan los sistemas con una tecnología neutral, que proporciona servicios que se implementarán posteriormente de forma diferente en cada plataforma concreta. Los modelos del nivel PIM pueden ser refinados tantas veces como se quiera hasta obtener una descripción del sistema con el nivel de claridad y abstracción deseado. Un PIM es una especificación de un sistema en términos de conceptos del dominio y con independencia de plataformas tecnológicas. MDA ha generado un amplio espectro de áreas de investigación: meta-modelos, perfiles UML, construcción y transformación de modelos, definición de lenguajes de transformación, construcción de herramientas de soporte, aplicación en dominios específicos, etc.

En el tercer nivel de abstracción, se encuentran los Modelos Específicos de Plataforma (Platform Specific Model-PSM), que combinan las especificaciones contenidas en un PIM, con los detalles de la plataforma elegida. Un PIM puede transformarse automáticamente en un PSM, y también a partir de distintos PSM se pueden generar automáticamente distintas implementaciones del mismo sistema.

Adicionalmente, en un nivel de abstracción más bajo, se puede considerar al código que implementa el sistema como otro modelo, con una notación distinta de la del resto de los modelos.

2.1.2.4 Transformaciones de modelos en MDA

Las transformaciones de modelos en MDA se definen como el proceso de convertir un modelo del sistema en otro modelo del mismo sistema.

MDA propone algunas restricciones en las transformaciones entre estos niveles de abstracción:

1. definir reglas de transformación solamente ente los modelos PIM y PSM, y
2. mantener una relación de trazabilidad entre los requisitos del sistema (expresados en el modelo CIM) y los artefactos representados en los modelos PIM y PSM que permiten llevar a cabo tales requisitos.

Por ello, las transformaciones contempladas en MDA pueden ser agrupadas en dos tipos [France et al., 2001]: verticales y horizontales.

1 Una transformación vertical es aquella en la que los niveles de abstracción del modelo origen y el modelo destino son diferentes. MDA contempla varias transformaciones verticales; por ejemplo:
   - cuando un PIM está suficientemente refinado se transforma en un modelo dependiente de la infraestructura final de ejecución, de modo que un PIM se transforma en uno o varios PSM,
   - cuando partiendo de una implementación en una plataforma concreta, se desea abstraerse de los detalles concretos de dicha plataforma, se está aplicando una transformación vertical para pasar del PSM al PIM.

   Ejemplos de estas transformaciones son el refinar un modelo o implementarlo en un lenguaje de programación concreto.

2 Una transformación horizontal es aquella en la que el modelo origen y el modelo destino corresponden al mismo nivel de abstracción. La principal aplicación de este tipo de transformaciones es la evolución de modelos; dicha evolución puede ser:
   - perfectiva, para mejorar un diseño,
   - correctiva, para corregir errores en el diseño, o
   - adaptativa, para introducir en un diseño nuevos requisitos o restricciones.

Otra aplicación de estas transformaciones permite obtener modelos de diferentes vistas del sistema (por ejemplo, de contenido, de hipertexto o de comportamiento) pasando de un PIM a otro, o bien obtener también modelos específicos para distintas plataformas pasando de un PSM a otro.

De acuerdo con OMG [MDA Guide 2003], una transformación de modelos " es el proceso de convertir un modelo en otro modelo del mismo sistema". En general, una transformación de modelos es el proceso de generación automática de un modelo origen a un modelo destino, de acuerdo a una definición de transformación, la cual es expresada en un lenguaje de transformación de modelos [Kurtev, 2005].

Durante los últimos años, se ha realizado una gran cantidad de investigaciones en el campo de las transformaciones. De estas investigaciones han resultado numerosas aproximaciones: basadas en teoría de grafos, lenguajes formales, lenguajes específicos de dominio (Domain Specific Language-DSL), etc. para definir transformaciones entre modelos, surgiendo asi propuestas concretas tales como Atlas Transfromation Language-ATL [Jouault et al., 2005] y Attributed Graph Grammar System-AGG [Taenatzer, 2000], entre otros. Así mismo, la OMG ha apromovido la especificación de transformaciones entre modelos, a través del estándar denominado Query View Transformations-QVT [OMG, 2005]. Cabe señalar que en la actualidad no existe una implementación de referencia que abarque la propuesta completa de QVT, aunque si han surgido propuestas que lo soportan en parte, tales como la herramienta MOMENT [Queralt et al., 2006] y SmartQVT [Model Ware Project, 2006].

2.1.2.5 MDA en la Línea de Productos Software

Uno de los elementos clave para una Línea de Producto Software o LPS es la representación y gestión de la variabilidad. En este contexto, se encuentra la iniciativa de MDA en la construcción de modelos de dominio (expresados como grafos de características o "features") y su posible transformación en modelos arquitectónicos. Dado que el enfoque de esta tesis tiene como punto de partida un modelo del dominio del diagnóstico, seguido de un modelo del dominio de aplicación, la vista general es una secuencia de transformaciones automáticas de dichos modelos de características a una arquitectura para la línea de productos software de diagnóstico.

MDA se concentra sobre un aspecto concreto de variabilidad. Actualmente, la variación de las características puede ser implícitamente definida en MDA a través del uso de perfiles de dominios particulares; por lo tanto, la variabilidad debe ser modelada explícitamente, con el fin de obtener productos óptimos.

La gestión de la variabilidad consiste en la habilidad para manejar eficientemente las diferencias o cambios en el desarrollo del software, considerando todas las fases de desarrollo, desde el análisis de requisitos hasta la implementación. El manejo de la variabilidad puede involucrar varios aspectos como los requisitos software, características, interfaz del usuario, o la plataforma de implementación.

La variabilidad es particularmente relevante en el campo de las líneas de producto software. La variabilidad entre productos de una línea de productos puede ser expresada en términos de

características [Kang et al., 1998], [Bosch, 2000]. Una característica es una unidad lógica de comportamiento que es especificada por un conjunto de requisitos funcionales o de calidad [Bosch, 2000].

El desarrollo de las LPS es, desde el punto de vista práctico, el enfoque que mayores éxitos ha alcanzado en el campo de la reutilización del software, gracias a la combinación de un desarrollo sistemático y a la utilización de componentes reutilizables de grano grueso [González-Baixauli et al., 2005].

En el proceso de desarrollo de una aplicación concreta (miembro de la línea de producto), ésta debe derivarse a partir del "framework" del dominio. En este proceso se deben seleccionar aquellas variantes que resultan apropiadas para los requisitos funcionales y no funcionales expresados por los usuarios finales del sistema. Las variantes seleccionadas deben ser tenidas en cuenta por el ingeniero de la aplicación del dominio (o ingeniero del conocimiento) en función de los requisitos particulares de la aplicación. Esta selección consiste en una transformación dirigida por la selección de características, efectuada por el ingeniero originando un modelo conceptual de características y que a su vez (por las relaciones de trazabilidad) generará el modelo arquitectónico de la aplicación.

La novedad que propone MDA es la posibilidad de automatizar (al menos parcialmente) la transformación que especifica cómo se convierten las instancias del modelo de características en una aplicación ejecutable. La transformación sería equivalente a la compilación de un modelo descrito por un lenguaje específico de dominio. El requisito previo antes de evaluar esta posibilidad es

disponer de un metamodelo que represente uniformemente los conceptos de cada una de las técnicas utilizadas.

[Deelstra, et al., 2003] aplican MDA como un medio para derivar productos específicos sobre una plataforma a partir de un PIM para su línea de productos (familias de productos configurables). En su trabajo, la variabilidad se instancia en la plataforma de destino del sistema. En este contexto, la ventaja de aplicar MDA es que la variabilidad de plataformas puede separarse del modelo del dominio e integrarse en la transformación. Sin embargo, aunque la plataforma es un punto de variabilidad, en el modelo del dominio existe más variabilidad que debe de tomarse en cuenta.

2.1.3 Líneas de Producto Software

Ante la complejidad creciente de los programas software, surge el concepto de líneas de producto de software (LPS) con la finalidad de controlar y minimizar los altos costos del desarrollo de software. Esta situación provocó la creación de un diseño que puede ser compartido por todos los miembros de una familia de programas (líneas de producto). De esta manera, un diseño hecho explícitamente para un producto, beneficia al software común y puede ser usado en diferentes productos, reduciendo los gastos generales y el tiempo para construir nuevos productos.

Además, el desarrollo de sistemas complejos se está complicando cada vez más debido a una serie de factores, como la aparición de nuevas tecnologías (Internet e intranet), la interconexión de varios

sistemas y plataformas, la necesidad de integrar viejos sistemas aun válidos ("Legacy Systems"), la adaptación personalizada del software a cada tipo de usuario, las necesidades específicas de un sistema y las diferentes plataformas de implementación. Esta situación lleva a necesitar, en cortos tiempos, múltiples versiones de la misma o parecida aplicación. Por ello, la Ingeniería del Software debe proporcionar herramientas y métodos que permitan desarrollar una familia de productos con distintas capacidades y adaptables a situaciones variables, y no sólo un único producto.

El objetivo principal de una LPS es reducir el tiempo, el esfuerzo, el costo y la complejidad de crear y mantener los productos de la LPS mediante:

1 la capacitación de los aspectos comunes de la línea de productos, a través de la consolidación y reutilización de los activos de entrada a la línea de productos, y
2 el manejo de los aspectos variables de los productos de la línea, a través de los puntos de variación de los activos y los modelos de decisión (Krueger, 2006).

Los productos de una LPS son diferenciados por sus características, donde una característica es un incremento en la funcionalidad del producto. Es evidente que las características pueden ser usadas para distinguir miembros de una LPS.

En este contexto han sido definidas las LPS por varios investigadores del área. Se ha recurrido a la bibliografía especializada, donde existen diversas definiciones, para recoger algunas de ellas y que a continuación se presentan:

Las líneas de producto son “un grupo o familia de productos relacionados realizados por el mismo proceso y para el mismo propósito, diferenciándose sólo en el estilo, modelo o tamaño. Una línea de producto agrupa aplicaciones relacionadas en familias de aplicaciones aprovechando los elementos comunes de la familia. Dado que los productos están relacionados (tienen funcionalidad o requisitos de usuario similares) hay un alto grado de aspectos comunes en una línea de producto” [Sonnemann, 1995].

Una línea de productos es “una colección de productos software que recogen un conjunto de requisitos de sistema comunes, y están organizadas alrededor de una actividad específica de negocio” [Cohen et al., 1995].

Las líneas de producto software son “un grupo de productos que comparten un conjunto de características comunes gestionadas, que satisfacen las necesidades específicas de un sector concreto del mercado” [Bass et al., 1997].

“Una línea de productos consiste en una arquitectura de línea de productos y en un conjunto de elementos software reutilizables que han sido diseñados para su incorporación en la arquitectura de línea de productos. Adicionalmente, la línea de productos incluye los productos que han sido desarrollados utilizando los assets mencionados” [Bosch, 2000].

Una línea de productos software es “un conjunto de productos que comparten un conjunto común de requisitos, pero que exhiben una variabilidad significativa en sus requisitos” [Griss, 2000].

Una línea de productos software es “un conjunto de productos que comparten una arquitectura común y un conjunto de componentes reutilizables” [Svahnherg et al., 2000b].

Una línea de productos software “consiste de una familia de sistemas de software que tienen un funcionalidad común y alguna funcionalidad variable” (Gomma, 2004).

Las líneas de producto software “se refieren a técnicas de ingeniería para crear un portafolio de sistemas software similares, a partir de un conjunto compartido de activos de software usando un medio común de producción” (Krueger, 2006).

Las líneas de producto software son definidas como “un conjunto de sistemas software que comparten un conjunto de características común y gestionado, que satisface las necesidades específicas de un segmento de mercado o misión y que son desarrolladas a partir de un conjunto central de “assets” de una manera preestablecida” [Clements et al., 2002].

Esta última definición se puede descomponer en 5 partes:

Productos: “*Un conjunto de sistemas software…”.* Las LPS cambian el enfoque del desarrollo de software al desarrollo de las LPS. Los procesos de desarrollo no intentan construir una aplicación, sino un número de ellas. [Clements et al., 2002].

Características: *“… que comparten un conjunto de características común y gestionado …”.* Las características son “unidades por medio de las cuales diferentes productos pueden ser distinguidos y definidos en una LPS” [Batory et al., 2004].

Dominio: *"…que satisface las necesidades específicas de un segmento de mercado o misión…".* Una LPS es creada en el ámbito de un dominio. Un dominio es un área de aplicación de productos software Un dominio es definido como "un bloque de conocimiento especializado, un área de experticia o un conjunto de funcionalidades relacionadas" [Northrop, 2002].

Activos de software reutilizables: *"…y que son desarrolladas a partir de un conjunto central de assets…".* Un activo de software reutilizable es "un artefacto o recurso que es usado en la producción de una línea de productos software mas que en un producto" [Clements et al., 2002].

Plan de producción: "…*de una manera preestablecida".* Se establece cómo es producido cada producto y como se unen todos los activos de software reutilizables para permitir la producción del producto final. Una línea de productos está formada por un conjunto de aplicaciones muy parecidas que pertenecen a un dominio determinado, como por ejemplo, el dominio del diagnóstico. El plan de producción es "una descripción de cómo son usados los activos de software reutilizables para desarrollar un producto de una línea de productos y especificar como usar el plan de producción para construir el producto final" [Chastek, 2002].

Las LPS tienen su origen en la década de los ochenta, en las escuelas de negocio, con un claro objetivo económico mediante el desarrollo sinérgico de productos [Knauber et al.,2001]. La reducción de costes, el descenso del tiempo de mercado, y la mejora en la calidad, son beneficios que se derivan de una estrategia basada en las líneas de productos. Sin embargo, para

cada posible beneficio derivado de la reutilización existe un coste asociado. En [Clements et al., 1998] se presentan los elementos que afectan tanto a la línea de productos como a los nuevos productos, junto a los beneficios y los costes asociados.

El desarrollo del software ha cambiado, se ha pasado de los programas individuales a artefactos reutilizables que pueden se usados en diferentes programas. Estos programas se distinguen por la existencia de distintos requisitos, que involucran incrementos en la funcionalidad. Por lo tanto, la LPS necesita técnicas para realizar esos incrementos en la funcionalidad. La clave de ello es cómo realizar esa variabilidad. En general, el código de un artefacto reutilizable es transformado durante el desarrollo de un programa. Esta transformación no cambia el tipo del artefacto software (p.e. código), sino solamente incrementa su funcionalidad. Este tipo de transformación es conocida como endógena y es fundamental para realizar la variabilidad de las LPS.

Es importante considerar técnicas de análisis de requisitos funcionales y no funcionales de la nueva línea de producto, así como técnicas de clasificación y estudio de las relaciones entre los requisitos del sistema, que a la vez sugieren una utilización de las bibliotecas de reutilización como gestores de este tipo de componentes software reutilizables o "*assets"*. Existen algunas propuestas muy útiles, que pueden ser consideradas como puntos de partida y que deben complementarse con las propuestas de las nuevas técnicas. En particular, Feature-Oriented Domain Analysis-FODA [Kang et al., 1990] y Organization Domain Modeling- ODM [Simos et al., 1996] que sistematizan las etapas iniciales del desarrollo de las LPS. Sin embargo, se necesitan herramientas más potentes para obtener modelos de requisitos más precisos

que los propuestos en estos métodos, así como cambiar su enfoque al espacio de la solución en lugar de estarlo al espacio del problema que se pretende solucionar.

Un concepto que está muy íntimamente relacionado con las líneas de productos es el de familia de productos, noción que aparece directa o indirectamente en las definiciones anteriores.

Una familia de productos de software es un conjunto de productos de software asociados a un dominio determinado. Una familia de productos de software tiene:

1 aspectos comunes que son compartidos por todos sus productos, tales como: un diseño arquitectónico común, un conjunto de componentes reutilizables, capacidades y servicios comunes, tecnologías comunes.
2 aspectos variables que establecen diferencias entre los productos.

[García et al., 2002] definen una familia de productos como: “el conjunto de productos diferentes que pueden ser producidos desde un diseño común, “assets” compartidos y mediante un proceso de ingeniería de aplicaciones. La pertenencia a este conjunto depende de la abstracción que unifica los “assets” en un sistema que funciona: una arquitectura, las reglas físicas o de negocio, o la plataforma hardware” o como “el conjunto de productos que comparten una plataforma común, pero tiene características y funcionalidad requeridas por el cliente”.

Las LPS permiten la reutilización sistemática de las familias de productos, es decir productos similares diferenciados por algunas características, promoviendo de este modo la industrialización del

desarrollo software. En [Deelstra et al, 2003] se describe una aproximación usando MDA para derivar productos de una clase particular de familias de productos software.

Algunas veces es confundido el término de líneas de producto con los términos familia de productos y dominio, o bien se realiza una correspondencia biunívoca entre línea de productos y unidad de negocio.

Una LPS no necesita construirse como una familia de productos, aunque es la forma de obtener los mayores beneficios. Una familia de productos no necesita constituir una línea de productos si los productos resultantes tienen poco en común en términos de sus características.

Una LPS no es un grupo de productos producidos por una única unidad de negocio. Puede esperarse que haya una gran correlación entre las líneas de productos y las unidades de negocio, pero conceptualmente son cosas distintas. Una unidad de negocio se forma por razones financieras o de organización, y puede ser responsable de una o más líneas de productos. Mientras que en una línea de productos se comparten y gestionan un conjunto común de características que satisfacen las necesidades específicas de un mercado seleccionado. Las líneas de productos que provienen de diferentes unidades de negocio pueden mezclarse para formar una “super línea de productos”.

Una LPS no es sinónimo de dominio. Un dominio es un cuerpo especializado de conocimiento, un área de experiencia o una colección de funcionalidad relacionada: La línea de productos hace

referencia a los sistemas software que comparten las características particulares de un sector de mercado.

La reutilización del software es uno de los objetivos fundamentales dentro de la ingeniería del software. Las LPS potencian la reutilización estratégica. Los "*assets*" de una LPS van más allá de sólo la reutilización de código. Cada producto de la línea de productos toma la ventaja de las etapas de análisis, diseño, implementación, planificación y prueba, realizados en cada uno de los productos desarrollados previamente en la LPS.

De los componentes de bajo nivel inicialmente reutilizados (básicamente código fuente), se ha ido ampliando el espectro a niveles de abstracción cada vez más altos. [Krueger, 1992] clasifica en los niveles más altos a los "*assets"*, como generadores de aplicaciones o arquitecturas software. [Mili et al., 1995] consideran sistemas transformacionales reutilizables y lenguajes de muy alto nivel. Una tendencia general es crear elementos reutilizables de grano más grueso, como por ejemplo bibliotecas de clases y "*frameworks"* [Wirfs-Brock y Johnson, 1990].

Analizando la historia sobre la reutilización del software, se observa que en los últimos años de la década de los 60`s, la idea de construir sistemas mediante la composición de componentes software fue presentada como solución a la crisis del software por [McIlroy, 1976]. Posteriormente y durante la década de los setenta se amparó la reutilización de los módulos. En la década de los 80`s , la influencia del paradigma orientado a objetos hizo que la clase se convirtiera en la unidad de reutilización. Sin embargo, estas tendencias fallaban en conseguir un enfoque sistemático de reutilización porque daban lugar a iniciativas individuales,

realizadas a pequeña escala, dejando de lado la reutilización de elementos software de mayor grano. Para solucionar esto surgen los *frameworks* [Wirfs-Brock et al., 1990] y la programación orientada a componentes [Szyperski, 1998] como aproximaciones a la reutilización del software.

Los "*frameworks"* capturan las decisiones comunes del diseño a un tipo de aplicaciones, estableciendo un modelo común a todas ellas, asignando responsabilidades y estableciendo colaboraciones entre las clases que forman dicho modelo. Este modelo común contiene puntos de variabilidad, conocidos como puntos calientes [Pree, 1995], tomando en cuenta los distintos comportamientos de las aplicaciones representadas por el "*framework"*. Los "*frameworks"* constituyen un hito importante en la reutilización de software orientado a objetos, ya que permiten la reutilización de código y también la reutilización del diseño de sistemas o subsistemas. La utilización de "*frameworks"*, como base de arquitecturas de referencia adaptables, junto con la utilización de técnicas de expresión (e implementación) de los puntos de variación de una familia de productos ha permitido el éxito del concepto de Línea de Producto Software [Bosch, 2000], [Clements et al., 2002].

Desde el año 1968 [Naur et al, 1969], el término factoría de software ha sido usado en diferentes técnicas de factoría. Entre estas técnicas se encuentran: división del trabajo en roles y especialización, métricas y procesos de software rigurosos, reutilización en las diferentes etapas de la producción, entornos y herramientas de automatización, y flexibilidad en la producción de los productos.

Todas éstas son facetas de la aplicación de ideas de factoría a la producción de software [Cusumano, 1989]. A finales del siglo XX se realizaron importantes estudios sobre los caminos organizativos necesarios para una eficaz implantación de una programa de reutilización [Tracz, 1995], estableciéndose que estos cambios pasan por la creación de un nuevo rol, el analista o ingeniero del dominio, quién se encarga de identificar, capturar, empaquetar y organizar, para su posterior reutilización, conocimiento sobre cómo se llevan a cabo las distintas tareas de producción en dominios de aplicación específicos [Prieto-Díaz,1990].

Asimismo, los trabajos en metaprogramación empezaron a aportar nuevas técnicas para crear generadores de aplicaciones y capturar en ellos los secretos de la producción de software en dominios específicos [Czarnecki et al., 2000]. Con ello, se llevaron a cabo estudios sobre la reutilización, que concluyen que la identificación, configuración e integración de artefactos reutilizables requiere de herramientas de automatización, como generadores de aplicaciones o sistemas de información, y del uso de lenguajes de muy alto nivel de abstracción [Griss, 1993], [Krueger,2002].

Con ello, aparece el concepto de Líneas de Producto de Software [Clements et al, 2002], formalizando técnicas de fábrica de software. Este paradigma propone la producción de familias (variantes similares) de productos a partir de un conjunto de activos (*assets*) siguiendo planes de producción predefinidos. Los activos reutilizables son artefactos con puntos de variabilidad así como metainformación y/o procesos asociados que describen cómo especializarlos para diferentes contextos de uso [Mc Gregor, 2004]. Asimismo, se anota la importancia de los planes de producción,

preferiblemente automáticos, al ofrecer tiempos y costes de ejecución predecibles.

Adicionalmente, aparece la Ingeniería Dirigida por Modelos (Model Driven Enginnering-MDE) [Schmidt, 2006] para ofrecer una prometedora técnica implementando planes de producción más automáticos y flexibles. MDE propone la definición de todo artefacto producido y/o reutilizado durante el proceso de través de transformaciones. En los últimos años se han realizado varios estudios sobre las ventajas de cómo utilizar LPS para mantener familias de modelos [Czarnecki et al., 2005], [Ávila_García et al., 2006], [Ávila-García, 2007].

2.1.3.1 Repositorios de las LPS

La línea de productos de software requiere almacenar sus activos de software en repositorios.

Un repositorio LPS es una base de datos especializada que almacena activos de software y facilita la recuperación y el mantenimiento de los activos de software. Su objetivo es asegurar la disponibilidad de activos para apoyar el desarrollo de productos de la LPS.

2.1.3.2 Arquitectura de una LPS

Uno de los activos de software más importantes de una LPS es la definición de la arquitectura de la línea de productos (ALP), debido a que esta arquitectura determinará el alcance de la línea de

productos y las características de los productos que pueden desarrollarse.

Una arquitectura software es la estructura que contiene los componentes software, las características externamente visibles de esos componentes y las relaciones entre ellas, de manera que se satisfagan los requisitos funcionales y de calidad del sistema. La ALP es la llave para la reutilización sistemática, ya que describe la estructura de los productos del dominio, mostrando sus componentes y las relaciones entre los mismos. Definir una adecuada ALP requiere armonizar las cualidades que se persiguen para los productos de la LPS con la definición de elementos opcionales, alternativos o variables. La ALP debe ser instanciada cada vez que se desarrolla un producto de la línea.

La ALP (también denominada arquitectura de dominio) es una arquitectura software genérica que:

1. describe la estructura de toda la familia de productos y no solamente la de un producto particular, y
2. captura los aspectos comunes y variables de un familia de productos de software:
    - los aspectos comunes de la arquitectura son capturados por los componentes de software que son comunes a toda la familia.
    - los aspectos variables de la arquitectura son capturados por los componentes de software que varían entre los miembros de la familia.

Por ello el desarrollo de arquitecturas estándares y de elementos software reutilizables requiere la comprensión de las características

comunes y de las que pueden variar en los sistemas que conforman un dominio.

En [Bosch, 2000] se describen tres dimensiones en las que se pueden descomponer los conceptos involucrados en las LPS. De esas tres, la que nos ocupa en esta sección corresponde a su primera dimensión, la cual divide al dominio de la línea de productos de acuerdo a sus "assets" principales, que son parte del desarrollo basado en reutilización. Esta aproximación también es presentada en [Jacobson et al., 1997], recibiendo el nombre de ingeniería de familia de aplicaciones, ingeniería del sistema de componentes e ingeniería de aplicación, respectivamente.

1 *Arquitectura software*.- el primer conjunto de "assets" està formado por la arquitectura software. La arquitectura software es la estructura o estructuras de un sistema, consistente en unos componentes software, las propiedades externamente visibles de dichos componentes y en las relaciones entre ellos [Bass et al., 1998b]. Por lo tanto la arquitectura software es el principal "asset" de la línea de productos. El diseño de una arquitectura software deberá dar cobertura a todos los productos de la LPS e incluir a todas las características que se comparten entre los productos.
2 *Componente*.- el segundo conjunto de "assets" está formado por los componentes que han sido identificados en la arquitectura. Estos deben reflejar la funcionalidad requerida, pero además deben soportar la variabilidad identificada en la arquitectura de la línea de producto. Este conjunto de "assets" generalmente son componentes de grano grueso, cercanos al concepto de "framework".
3 *Sistema*.- el tercer conjunto de "assets" está formado por los sistemas construidos sobre la base de la arquitectura y los

componentes de la línea de productos. Esta actividad requiere la adaptación de la arquitectura de la línea de producto para ajustarse a la arquitectura de sistema, que puede requerir eliminar o añadir componentes o relaciones entre ellos, desarrollar extensiones a los componentes existentes, configurar los componentes y desarrollar elementos software específicos para el sistema.

El desarrollo de elementos software reutilizables requiere determinar sus arquitecturas. Dado que una arquitectura define cómo los elementos software se integran para crear un sistema, el contar con arquitecturas estandarizadas permite definir los contextos en los que los elementos reutilizables pueden ser desarrollados. Así pues, la ingeniería de dominio surge como una evolución de la reutilización sistemática del software basada en modelos donde las arquitecturas software juegan un papel importante.

2.1.3.3 Ingeniería del Dominio

La reutilización del software dentro de un dominio de aplicación pasa por el descubrimiento de elementos comunes a los sistemas pertenecientes a dicho dominio [García et al., 2002]. Este enfoque produciendo un cambio de un desarrollo orientado a un único producto software a un desarrollo de varios productos que contienen unas características comunes, formando una familia de productos. Esto implica una reestructuración en el desarrollo del software, surgiendo dos procesos distintos: la ingeniería del dominio y la ingeniería de aplicación.

La ingeniería del dominio se centra en el desarrollo de elementos reutilizables que formarán la familia de productos, mientras que la ingeniería de aplicación se orienta hacia la construcción o desarrollo de productos individuales, pertenecientes a la familia de productos, y que satisfacen un conjunto de requisitos y restricciones expresados por un usuario específico, reutilizando, adaptando e integrando los elementos reutilizables existentes y producidos en la ingeniería de dominio [García et al., 2002].

Es posible definir la ingeniería del dominio como el proceso que se necesita para el diseño sistemático de una arquitectura y de un conjunto de elementos software reutilizables, que pueden ser usados en la construcción de una familia de aplicaciones relacionadas o subsistemas. [Griss et al., 1998] definen la ingeniería del dominio como "el proceso sistemático que incorpora criterios de negocio y produce un soporte racional, modelos y arquitecturas que permiten tomar mejores decisiones, llevar un registro del dominio, obtener nuevas versiones y mejorar el proceso de desarrollo gracias al conocimiento que se tiene del sistema". El objetivo fundamental de la ingeniería del dominio es la optimización del proceso de desarrollo del software en un espectro de múltiples aplicaciones que representan un negocio común o problema de dominio [Simos et al., 1996].

La forma más efectiva de organizar la ingeniería de dominio es hacerlo en el contexto de líneas de productos específicas. Dentro de una línea de productos, los dominios son analizados y la información sobre éstos capturada y organizada en modelos del dominio y en elementos software reutilizables ("assets"), facilitando además su evolución [Klingler y Solderitsch, 1996].

La ingeniería del dominio tiene sus orígenes, a principios de la década de los ochenta, en los trabajos de [Neigbors, 1984] sobre la reutilización basada en la generación a través del paradigma Draco. A partir de entonces, se han publicado varias aproximaciones que incluyen un amplio rango de métodos y procesos, tanto formales como informales, para llevar a cabo las diferentes actividades en que se descompone la ingeniería de dominio. Entre las diversas propuestas existentes en la ingeniería del dominio cabe destacar a: Feature-Oriented Domain Analysis-FODA [Kang et al., 1990], Organization Domain Modeling-ODM [Simos, 1995], [Simos et al., 1996], Domain Architecture-based Generation for Ada Reuse-DAGAR [Klingler y Solderitsch, 1996], Object Oriented Domain Analysis-OODA [Cohen y Northrop, 1998], Feature Reuse-Driven Software Engineering Business-FeatureRSEB [Griss et al., 1998], FODAcom [Vici y Argentieri, 1998], y Feature-Oriented Reuse Method-FORM [Kang, 1998], [Kang et al., 1998a], [Lee et al., 2000]. La diferencia entre estas propuestas es fundamentalmente la manera en la que se identifica al dominio y hacen uso de la experiencia sobre el dominio, la arquitectura y los sistemas existentes.

La diversidad en las propuestas trae consigo una diversidad en la terminología, existiendo una falta de consenso en la misma. Por ingeniería de dominio se hace referencia a todo el proceso de creación de los elementos reutilizables propios del dominio; pero existe uniformidad en el nombre de las fases que componen el proceso. Existen dos fases muy diferenciadas:

1 análisis del dominio [Arango y Prieto-Díaz, 1991], donde se adquiriere y se modela el conocimiento sobre el dominio, realizando una búsqueda de elementos comunes y diferencias en la familia de sistemas que conforman el dominio;

2 ingeniería de componentes del dominio, en la que los elementos software reutilizables propios del dominio son creados.

2.1.3.4 Metamodelo de ingeniería de procesos de software

Los principios de MDA pueden aplicarse a otras áreas como el modelado de procesos de negocios donde la independencia de la tecnología y de la arquitectura (PIM), es adaptado a los sistemas como y a los procesos manuales.

El término arquitectura en los metamodelos no se refiere al modelo de la arquitectura del sistema sino a la arquitectura de los distintos estándares y formas del modelo que sirven de base tecnológica al MDA.

El modelo MDA está relacionado con múltiples normas, incluyendo: Unified Modeling Language-UML, Meta_Object Facility-MOF, XML Metadata Interchange-XMI, y el Software Process Engineering Metamodel-SPEM.

Los procesos de desarrollo de software se han vuelto tan complejos que es necesario el uso de lenguajes formales para modelarlos [Derniane et al., 2004]. Para ello, la OMG en el año 2001 ha creado el estándar Software Process Engineering Metamodel-SPEM [SPEM], que define un lenguaje estándar de modelado que permite describir tales procesos.

Esta línea de investigación tuvo su origen en un estudio que exploraba las sinergias entre las Líneas de Producto Software, la Ingeniería Dirigida por Modelos y la Ingenierìa de Procesos Software [Avila-García et al, 2006 SPEM es un metamodelo y un UML Profile dedicado al modelado de procesos software. SPEM es utilizado para modelar una familia de procesos software relacionadas, usando la notación UML. El metamodelo de SPEM es muy extenso, permitiendo el modelado de muchos aspectos y problemas del proceso de desarrollo.

2.1.3.5 Programación Orientada a Características

La programación orientada a características (Feature Oriented Programming-FOP) es el estudio de la modularidad de características, donde las características son consideradas como ciudadanos de primera especie en el diseño. Una característica es una parte relevante de un objeto o cosa [López-Herrejón, 2005].

FOP es una metodología de programación que sostiene que el crear varias características y conectarlas a través de procedimientos/métodos/funciones, cuidando la funcionalidad principal, es la mejor manera de quitar la redundancia y mejorar la eficiencia. FOP se enfoca principalmente sobre las características de un sistema, en lugar de los objetos que lo comprenden (como lo sería en un lenguaje orientado a objetos). FOP puede ser visto como una metodología modular. Las funciones deben ser vistas como una herramienta específica para completar una tarea general, mientras que el programa mismo debe tener un propósito específico claro [López-Herrejón, 2005].

FOP es un paradigma emergente para sintetizar, analizar y optimizar aplicaciones. Una aplicación es especificada declarativamente como un conjunto de características, de la misma manera que se hace en muchos productos de la elección de ofertas en una compra (e.g., automóviles, ordenadores y sus periféricos). La tecnología FOP traduce tales especificaciones declarativas en programas eficientes [Batory, 2004].

FOP es una aproximación para desarrollar LPS en donde los programas se construyen por medio de la composición de características, como bloques de los programas. Estas características son unidades que incrementan la aplicación de la funcionalidad mediante el cual diferentes productos pueden ser distinguidos y definidos con una LPS. Cada una de las características puede estar incluida en varios artefactos software. En general una LPS es caracterizada por el conjunto de características que soporta.

Un modelo FOP de una LPS ofrece un conjunto de operaciones, donde cada operación implementa una característica. En [Trujillo, 2007] se define M={f,h,i,j} como el modelo M con características f,h,i,j, las cuales pueden ser constantes o funciones. Las constantes representan programas base. Por ejemplo:
f , que significa que un programa con característica f
Las funciones representan refinamientos de programas que extienden un programa que es recibido como entrada. Por ejemplo:
j.x, que significa: agregar característica i al programa x
donde . denota la aplicación de la función.
Un producto se obtiene a través de la construcción de características. Un programa es una expresión del tipo:

prog1=i.f // que significa que el programa prog1 tiene las características i y f.

Un conjunto de programas que puede ser creado a partir de un modelo, es su línea de producto.

2.1.3.6 Modelo de características clásico

El análisis de mercado, en el caso del software, corresponde a lo que se conoce como modelado del dominio. Este análisis no se centra en lo que necesita un usuario concreto sino en lo que necesitan un conjunto de usuarios.

Los modelos utilizados en el análisis de las LPS han de permitir el modelado de requisitos tanto comunes como variables en la línea de productos. Cada conjunto de características define un programa único en una LPS.

La primera notación del modelo de características fue introducida por el método FODA [Kang et al., 1990]. Posteriormente, [Griss et al., 1998] integraron esta notación con procesos y productos de RSEB. Éste fue extendido con nuevos constructores en [Svahnherg et al., 2000]. [Clauss et al., 2001] describieron los modelos de características como una extensión de la anotación del diagrama de clases de UML. [Riebisch et al., 2003] presentan una visión general del modelado de la variabilidad en las LPS orientadas a objetos. [Batory et al., 2006] definen a un modelo de características como un conjunto de características organizadas jerárquicamente. Otra definición dada por [Batory et al., 2006] dice que un modelo de

características es un lenguaje deductivo para especificar productos en una LPS.

Los modelos de características son la clave técnica de las líneas de producto para desarrollar software. Ellos son modelos formales que usan características para especificar productos.

Un diagrama de características es una representación gráfica de un modelo de características. El trabajo realizado por [Batory et al., 2006] presenta una notación gráfica utilizada en el modelo de características clásico. Se señala que las relaciones entre un padre (o mezcla) de características y sus hijos (o subcaracteristicas) son categorizadas como:

1 *and (y)*.- todas las subcaracterísticas deben ser seleccionadas
2 *or alternative (o)*.- solamente una subcaracterística puede ser seleccionada
3 *xor (xor)*..- una o más pueden ser seleccionadas
4 *mandatory (obligatorio)*.- características que son requeridas
5 *optional (opcional)*.- características que son opcionales

Su notación gráfica es mostrada en la siguiente figura:

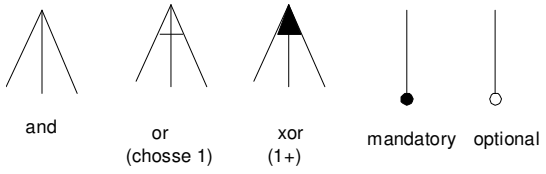


Figura 2. Notación gráfica utilizada en el modelo de características clásico

2.1.4 El modelo PRISMA

En nuestros días existen una gran cantidad y variedad de trabajos relacionados con el desarrollo de sistemas software complejos: modelos arquitectónicos, reconfiguración dinámica de arquitecturas, componentes, aspectos, lenguajes de definición de arquitecturas. Todo ello persigue un consenso, aún no logrado, en los conceptos básicos de sistemas software complejos, su definición precisa y la metodología a aplicar para su desarrollo.

De los complejos requisitos software surge el modelo PRISMA (Plataforma OASIS para Modelos Arquitectónicos) [Pérez et al., 2002], su potencia expresiva, facilidad de uso y novedad. De esta manera se cubre el hueco existente en el modelado de sistemas software altamente reconfigurables y reutilizables dentro de un marco de calidad controlada. PRISMA presenta propiedades y ventajas en la construcción de modelos arquitectónicos complejos.

El modelo arquitectónico PRISMA integra dos aproximaciones: el Desarrollo de Software Basado en Componentes-DSBC [Szyperski, 1998] y el Desarrollo de Software Orientado a Aspectos-DSOA [AOSD]. Esta integración se consigue definiendo los elementos arquitectónicos mediante aspectos: coordinación, funcional, persistencia, distribución [Ali et al., 2005], presentación, etc. De esta forma, el modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las propiedades necesarias de cada uno de ellos. Por ello un elemento arquitectónico de PRISMA puede ser analizado desde dos vistas diferentes: interna y externa. La vista interna muestra al elemento arquitectónico como un prisma, de forma que cada lado del prisma

corresponde a un aspecto. Mientras que la vista externa encapsula la funcionalidad del elemento arquitectónico teniendo sólo la visibilidad de los servicios que éste publica a través de sus puertos.

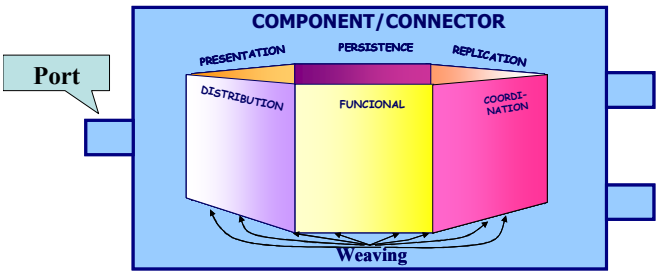


Figura 3. Vistas interna y externa de un elemento arquitectónico PRISMA

El modelo PRISMA consta de tres tipos de elementos arquitectónicos: componentes, conectores y sistemas. Un componente captura la funcionalidad del sistema, mientras que un conector actúa como coordinador entre otros elementos arquitectónicos. Un sistema es un elemento arquitectónico de mayor granularidad que permite encapsular un conjunto de componentes, conectores y otros sistemas, correctamente conectados entre sí.

Un elemento arquitectónico PRISMA está formado por un conjunto de aspectos de diferente tipo, las relaciones de entretejido entre estos aspectos (*weavings*) y uno o más puertos.

Un aspecto PRISMA es un asunto de interés del sistema (*concern*) que es común y compartido por el conjunto de tipos del sistema (*crosscutting concerns*). Los tipos de aspectos (funcional, coordinación, distribución, etc.) que forman un elemento arquitectónico varían dependiendo del sistema software.

El *weaving* establece las sincronizaciones necesarias entre los aspectos que conforman un elemento arquitectónico. El *weaving* define que la ejecución de un servicio de un aspecto puede generar la invocación de un servicio de otro aspecto.

Los puertos representan los puntos de interacción entre los elementos arquitectónicos.

Existen dos tipos de relaciones para interconectar los elementos arquitectónicos: *attachments* y *bindings*. Los *attachments* establecen el canal de comunicación entre el puerto de un componente y el puerto de un conector., mientras que los *bindings* establecen la comunicación entre un puerto de un sistema y un puerto de un elemento arquitectónico de los que encapsula. De este modo, los *bindings* permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman el modelo.

PRISMA especifica la arquitectura en dos niveles: el de definición de tipos y el de configuración. En el nivel de tipos se definen los tipos de la arquitectura: interfaces, aspectos, componentes, conectores y sistemas, los cuales son guardados en una librería para su reutilización. A nivel de configuración se especifican los modelos arquitectónicos de un determinado sistema software. Esta diferenciación proporciona importantes ventajas, ya que permite

gestionar de forma independiente los tipos de elementos y las topologías específicas de cada sistema, obteniendo de este modo un mayor nivel de reutilización y un mejor mantenimiento de las librerías de tipos.

PRISMA ha tenido en cuenta varios trabajos para su modelo, que incorporan las distintas áreas de interés, entre ellos:

El trabajo realizado por [Garlan et al., 2001] es un punto de referencia muy importante a la hora de establecer los elementos de un sistema software complejo, en el cual se introducen conceptos como componente, conector, sistema, puertos de entrada y de salida.

Se han considerado los trabajos de [Andrade et al., 1999] acerca del concepto de contrato, ya que se han definido los conectores como una extensión del contrato propuesto por su trabajo. Dicha extensión se produce debido a la incorporación del concepto de coreografía en el conector, concepto utilizado con una semántica distinta en los trabajos de [Monge et al., 2002] y [Bracciali et al., 2002].

Existe una gran variedad de lenguajes de definición de arquitecturas (LDA), cada uno de ellos presenta ventajas y desventajas, tal y como presenta el estudio realizado en [Medvidovic et al., 2002]. Uno de los lenguajes que más se aproxima al LDA de PRISMA es el propuesto por [Loques et al., 2000] en su modelo R-RIO, en el cual incorporan aspectos de reconfiguración dinámica a través del metanivel. Estos también se encuentran en enfoques orientados a un nivel de abstracción inferior, es decir, orientados a implementación como [McGurren et

al., 2002]. En Guaraná [Oliva et al., 1998] se presenta un metanivel complejo que hay que definir durante la compilación, siendo su estructura ajena al modelo.

Otro marco de referencia es el desarrollo de software orientado a aspectos en el que se encuentran trabajos como [AOSD]. Esta aproximación se aplica principalmente en la implementación y no se aplica para la especificación a un alto nivel de abstracción de sistemas arquitectónicos que integren componentes y aspectos. El DSOA está teniendo un gran auge en la comunidad informática debido a que su código modular consigue que los costes de desarrollo, mantenimiento y evolución del software se reduzcan. Por este motivo, existe una tendencia hacia la incorporación del concepto de aspecto desde las primeras fases del ciclo de vida, existiendo trabajos elaborados en el área de la especificación de requisitos [Rashid et al., 2002].

Los aspectos se definen en [Kiczales et al., 1997] como asuntos de un sistema que tienden a estar presentes en varias componentes funcionales, dando lugar al término *crosscutting concerns*. En PRISMA se entiende a un aspecto como la definición completa de la estructura y el comportamiento de un tipo desde un determinado punto de vista o *concern*. Por ello, el *crosscutting* no sólo se realiza sobre propiedades funcionales sino sobre cualquier *concern*.

En PRISMA se define el concepto de aspecto sin tener en cuenta las peculiaridades sintácticas de los lenguajes de programación orientados a aspectos. Su definición se basa en las características comunes de un sistema susceptibles de ser reutilizadas. Atendiendo a esta definición, menos dependiente del lenguaje de desarrollo, la funcionalidad es susceptible de reutilización en

distintos tipos e incluso dicha funcionalidad se puede subdividir en distintos aspectos. Los últimos trabajos de [Brito et al., 2003] empiezan a apuntar hacia esta última idea.

La mayoría de los modelos arquitectónicos analizan cuáles son los tipos básicos para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir sus tipos, también especifica los aspectos que cubren las necesidades de cada uno de ellos. Un tipo del modelo PRISMA puede ser visto como un prisma con tanta caras como aspectos considere, los cuales están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código (no monotonicidad) que puede producir la programación orientada a aspectos [Kiczales et al., 2001].

Dentro de los distintos modelos de aspectos que se ha creado hasta el momento, se pueden observar dos claras tendencias que permiten clasificarlos en dos grades grupos: los modelos estáticos y los dinámicos: La diferencia entre ambos se encuentra en el proceso de generación de código. Los modelos estáticos generan un solo componente en el que se encuentra mezclado el código funcional y el código de los aspectos, mientras que en los modelos dinámicos se generan distintas entidades para los distintos aspectos y el componente (modelo de disfraces) [Herrero, 2003]. La ventaja de los modelos dinámicos es la facilidad que proporcionan para incluir o eliminar aspectos durante el proceso de ejecución.

Algunas de las aproximaciones que han integrado el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA) a nivel de implementación

han extendido los “java beans” utilizando aspectos para su descripción y añadiendo el concepto de conectores [Suvée et al., 2003]. Además, los aspectos pueden ser manipulados en tiempo de ejecución. Un trabajo que describe los asuntos que se tienen que tener en consideración para la integración de ambas aproximaciones es introducido en [Constantinides et al., 2000]. Los asuntos y requisitos descritos en este trabajo son contemplados en PRISMA, ya que PRISMA aglutina estas dos aproximaciones actuales del DSBC y DSOA

El metanivel del modelo PRISMA y las propiedades reflexivas de los lenguajes diseñados, dan soporte a la evolución de los elementos arquitectónicos y a la reconfiguración dinámica de la topología. Esta característica permite definir un metanivel por aspecto que reifica las propiedades que deseen ser evolucionadas basándose en la reflexión de OASIS [Carsi, 1999]. Con ello, PRISMA da soporte a la evolución de sus modelos reduciendo el esfuerzo de mantenimiento de sus productos; la evolución proporciona la dinámica del tipo, i.e. la capacidad de cambio de su estructura y comportamiento basándose en la evolución del software que plantea la reflexión de OASIS.

Los elementos arquitectónicos PRISMA se obtienen identificando escenas funcionales del sistema y asignando a cada elemento una escena funcional, dependiendo de si la escena es simple o compleja, el elemento será un componente (componente simple) o un sistema (componente complejo), respectivamente. El concepto de escena aparece en el trabajo de [Noriega et al., 1998], y es definido con un enfoque PRISMA por [Pérez, 2003] al considerar que “Una escena se caracteriza por las tareas que se desempeñan

en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla"

Finalmente, es muy conveniente mencionar la investigación que se ha estado realizando en el marco del modelo PRISMA, en el grupo denominado Ingeniería de Software y Sistemas de Información (ISSI) del Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia, a través de un importante análisis y una especificación detallada, de los aspectos funcional, de distribución y de replicación, que aparecen en los trabajos de [Ali et al., 2005] y [Pérez, 2006].

2.2 TRABAJOS RELACIONADOS

Existen un gran número de trabajos relacionados con la aproximación presentada en esta tesis. Las metodologías y aplicaciones en esta temática han producido una amplia variedad de productos de investigación, ofreciendo sugerencias y soluciones en dominios específicos.

Entre los más importantes se pueden señalar las siguientes:

1 Las investigaciones realizadas por [Liao, 2005]:
   - al examinar las metodologías de los sistemas expertos y clasificarlos en once categorías. Dos de esas categorías corresponden a los sistemas basados en reglas y los sistemas basados en el conocimiento. Estas dos categorías han sido tomadas en cuenta en esta tesis al utilizar el conocimiento representado en forma de reglas (cláusulas de Horn) y hechos (variables observables, i.e. propiedades de las entidades a diagnosticar).
   - al mencionar que las aplicaciones de los sistemas expertos son construidas como sistemas orientados a un problema en un dominio específico. Esta tesis está enfocada al dominio del diagnóstico, con varios casos de estudio en el dominio específico de aplicación.
   - al mencionar que el desarrollo de los sistemas expertos ha sido caracterizado por separar el conocimiento de los procesos, como unidades independientes. En el modelo arquitectónico presentado en esta tesis, los elementos arquitectónicos en el nivel de tipos son definidos

tomado en cuenta este concepto, específicamente cuando es considerado un componente que contiene el conocimiento del dominio de aplicación y otro componente que ejecuta los procesos de inferencia para llevar a cabo el diagnóstico.

2 Las investigaciones realizadas por [Giarratano et al., 2004] y otros autores, en el campo de los sistemas expertos consideran que las arquitecturas de esos sistemas están basadas solamente en componentes. La arquitectura de la LPSD ha integrado dos aproximaciones que combinan componentes (DSBC) con aspectos (DSOA), incrementando de esta forma, la reusabilidad y el mantenimiento de los sistemas de diagnóstico.

3 La integración de las aproximaciones que combinan DSBC y DSOA, introducida por [Constantinides et al., 2000]. Los asuntos y requisitos descritos en este trabajo son contemplados en el modelo arquitectónico propuesto, obteniendo las ventajas de cada una de ellas, al definir los elementos arquitectónicos mediante sus aspectos.

4 La implementación de los sistemas expertos que ha sido realizada en diferentes paradigmas de programación, tales como la estructurada, la lógica y la orientada a objetos. Estos paradigmas están orientados a lenguajes de cuarta generación y métodos de programación visual para dar una comunicación amigable con el usuario. PRISMA provee un Lenguaje de Descripción de Arquitecturas (ADL) para definir un modelo arquitectónico que sigue la aproximación MDA para generar código automáticamente.

5 La definición de un modelo de alto nivel de abstracción, independiente de la tecnología, contemplado en MDA, independiente de la tecnología (PIM). En esta tesis, se ha considerado esta línea con un enfoque hacia los sistemas expertos basados en líneas de producto.

6 La detección de componentes basada en la descomposición funcional del problema, es decir del sistema, compatible con la metodología Architecture Based Design Method [Bachman, 2000], propuesta por The Software Engineering Institute of The Carnegie Mellon University para diseñar arquitecturas software de un dominio de aplicación. Esta metodología ha sido aplicada para la construcción del modelo arquitectónico de diagnóstico.

7 El trabajo realizado por [Garlan, 2001], el cual es un punto de referencia muy importante para establecer los elementos de un sistema software complejo, en el cual se introducen conceptos como componente, conector, sistema, puertos de entrada y de salida. Dichos conceptos han sido contemplados en el modelo propuesto.

8 El concepto de contrato de [Andrade et al., 1999], ya que se han definido los conectores de los modelos arquitectónicos de la LPSD como una extensión de dicho concepto; incorporando la coreografía en los conectores, especificada como el protocolo del aspecto de coordinación de dichos elementos arquitectónicos.

9 La existencia de la gran variedad de lenguajes de definición de arquitecturas (LDA), donde cada uno de ellos tiene ventajas y

desventajas., tal y como presenta el estudio realizado en [Medvidovic et al., 2000]. Uno de los lenguajes que más se aproxima al LDA de PRISMA es el propuesto por [Loques et al., 2000] en su modelo R-RIO, el cual tiene capacidades de reconfiguración al igual que PRISMA, sin embargo no incorpora la noción de aspecto.

10 Las líneas de producto software, tópico de discusión importante en la última década, que incorporan técnicas, metodologías y varios trabajos relacionados con BOM. Entre ellos se encuentran los trabajos de:

- [Batory et al., 2006] donde se expresan las características del dominio en un Modelo de Características, y el uso de FOP como técnica para insertar características.
- [González et al., 2006] quienes aplican la propuesta del MDA y la Ingeniería de Requisitos para Líneas de Productos.
- [Clements et al., 2002] que usan la aproximación de desarrollo de Líneas de Producto Software, considerando una división entre la ingeniería del dominio u la ingeniería de la aplicación, para el reuso y la automatización de los procesos software.
- [Trujillo, 2007] quien ha desarrollado la herramienta XAK para insertar características en documentos XML por medio de plantillas XSLT.
- [Ávila-García et al., 2006] que han desarrollado una herramienta MDA con funcionalidades de metamodelado sobre MOF y transformaciones en ATC. Ellos integran las funcionalidades de SPEM como

procesos de modelado y de RAS para empaquetar activos reutlizables.

- [Santos, 2007] que propone el desarrollo de una técnica basada en MDA para gestionar la variabilidad en las Líneas de Producto Software.
- En [ACM, 2006] han sido publicados varios trabajos relacionados con la Ingeniería de la Línea de Productos Software.

Capítulo 3

ANÁLISIS DE LA APROXIMACIÓN MDA PARA LÍNEAS DE PRODUCTO ORIENTADAS AL DIAGNÓSTICO

En este capítulo se presenta la primera parte del objetivo de esta tesis, que consiste en abordar el dominio del diagnóstico de forma genérica, a través del análisis de una aproximación metodológica basada en MDA para crear líneas de producto software orientadas al diagnóstico, denominada BOM (Base-Line Oriented Model Diagnosis): un Generador Automático de Sistemas de Diagnóstico basado en Líneas de Producto.

3.1 Líneas de Producto Software en el diagnóstico

Una línea de productos software es un conjunto de sistemas de software que comparten un conjunto de características comunes que satisfacen las necesidades específicas de un dominio particular. Este nuevo paradigma permite mejorar la calidad del software, así como reducir los costes, disminuir los esfuerzos y acortar los tiempos de desarrollo. Una vez realizada la inversión de desarrollo de LPS, los productos son desarrollados de forma rápida, barata y con alto nivel de calidad.

Una familia de productos de software tiene aspectos comunes que son compartidos por todos sus productos, y aspectos variables que establecen diferencias entre los productos. Por ello, las LPS permiten la construcción sistemática de productos similares, diferenciados por algunas características.

Una familia de productos de software es un conjunto de productos de software asociados a un dominio determinado. Dicho domino representa el tipo de tarea que desempeña el sistema software final. Uno de los dominios más utilizados en nuestros días es el diagnóstico médico, en el que se insistirá en esta tesis.

Cabe mencionar que los sistemas de diagnóstico han sido desarrollados considerando la arquitectura de los sistemas expertos o sistemas basados en el conocimiento. Dicha arquitectura cuenta con varios componentes, pero tres de ellos siempre están presentes en este tipo de sistemas: un componente que involucra los procesos de inferencia, un componente que

contiene la información del dominio y un componente que representa la interfaz del usuario.

Estos sistemas han sido desarrollados considerando la construcción de todos sus componentes en forma "ad-hoc" al caso de estudio (o aplicación del dominio), o bien, considerando un motor de inferencia que puede ser utilizado con diversas bases de conocimientos. Esto implica que existe variabilidad en la base de conocimientos, ya que las propiedades de las entidades a diagnosticar difieren de un caso al otro.

Sin embargo para llegar a una conclusión diagnóstica, el experto del dominio trata de aplicar todas aquellas estrategias de razonamiento que le permitan obtener un diagnóstico de la forma más eficiente. En otras palabras, el tipo de razonamiento que resulta más adecuado para solucionar una parte del proceso del diagnóstico, no tiene porque ser el más adecuado para resolver otra parte del mismo. Además, no todas las aplicaciones del dominio obtienen el diagnóstico aplicando la misma técnica de razonamiento. Por ello las estrategias de razonamiento del mecanismo de inferencia difieren de un caso de estudio a otro, lo que implica que existe variabilidad en el motor de inferencia.

La interfaz del usuario también involucra variabilidad, ya que un operario del sistema puede desempeñar más de un rol.

Así mismo, consideramos que existen varias razones para que los sistemas que realizan tareas de diagnóstico estén más inclinados al paradigma LPS que al software tradicional:

La primera razón es que los sistemas de diagnóstico en dominios específicos han sido tratados de forma particular para realizar el diagnóstico de un caso de estudio, sin embargo las aplicaciones del diagnóstico involucran una amplia gama de espacios tecnológicos en el dominio..

La segunda razón es la heterogeneidad de las aplicaciones del dominio, dado que en cada caso de estudio, las características del diagnóstico difieren de un caso al otro. La variabilidad de las LPS permite abordar esa heterogeneidad.

Una tercera razón es la rapidez con la que cambia la tecnología. La LPS facilita el desarrollo de los productos en distintas plataformas y su uso en distintas tecnologías.

Finalmente, la cuarta razón es el propio proceso de inferencia del diagnóstico, el cual cambia según la aplicación del dominio, por lo que varias estrategias de razonamiento están implicadas en los procesos del diagnóstico. La LPS permite considerar estas estrategias de razonamiento como una más de las "características" que diferenciarán a la LPS.

3.2 Aproximación para los procesos de la ingeniería de la línea de productos software

Una línea de productos está formada por un conjunto de aplicaciones muy parecidas que pertenecen a un determinado

dominio, como por ejemplo, el dominio del diagnóstico. Las aplicaciones en este dominio son muy diversas, entre las que se encuentran el diagnóstico médico para detectar la enfermedad que padece un paciente, el diagnóstico de emergencias para clasificar accidentados en desastres, el diagnóstico televisivo que indica si un video es considerado como adecuado o no para ser transmitido al aire, y el diagnóstico educativo para calificar la etapa de desarrollo de un programa educativo, entre otros.

El proceso de desarrollo de la LPS no intenta construir una aplicación, sino un número de ellas. Esto conlleva realizar un cambio en los procesos de ingeniería realizando una distinción entre ingeniería del dominio e ingeniería de la aplicación. En general, la ingeniería del dominio (desarrollo de los componentes básicos reutilizables) determina la concordancia y la variabilidad de la LPS. La construcción de los componentes básicos reutilizables y su variabilidad es separada de la producción de las aplicaciones de la línea de producto.

El Generador de Sistemas de Diagnóstico basado en Líneas de Producto se fundamenta en la aproximación para desarrollar LPS mencionada por [Clements et al., 2001]. La tabla 1 bosqueja el proceso total:

Ingeniería del dominio		
análisis del dominio	desarrollo de los componentes básicos reutilizables	planeamiento de la producción

Ingeniería de la aplicación		
caracterización del producto	síntesis del producto	construcción del producto

Tabla 1. Procesos de la ingeniería en la LPS

El **análisis del dominio** estudia la variabilidad del dominio. Frecuentemente este estudio es dado en términos de características del dominio y es representado usando un modelo de características.

El **desarrollo de los componentes básicos reutilizables** concibe, diseña e implementa los componentes básicos reutilizables. Esto no sólo involucra el desarrollo de la funcionalidad del dominio, sino también define cómo los componentes básicos reutilizables deben ser extendidos.

El **planeamiento de la producción** define cómo los productos individuales son creados. En general implica la capacidad de producción de la LPS.

La **caracterización del producto** elige las características que diferencian un producto seleccionado. Este proceso se inicia con la selección de las características.

La síntesis del producto reúne los componentes básicos reutilizables para obtener la "materia prima" (un producto está compuesto de). Las técnicas de variabilidad son usadas en este proceso.

La construcción del producto procesa la "materia prima" siguiendo el proceso de construcción (e.g. compilar, generar código, ejecutar, etc.), para obtener un producto final.

La división entre la ingeniería del dominio y la ingeniería de la aplicación es fundamental en la aproximación de las LPS [Clements et al., 2002]. Esta partición es la base de cualquier intento de reutilización y automatización en procesos de software [Czarnecki et al, 2000]. Mientras en la ingeniería del dominio creamos un conjunto de activos para reutilizar y automatizar en un dominio de aplicación específico, en la ingeniería de aplicación los usamos para producir, en dicho dominio, productos de software de mayor calidad, empleando menos coste y tiempo.

3.3 Conceptos utilizados en los procesos de la ingeniería de la LPSD

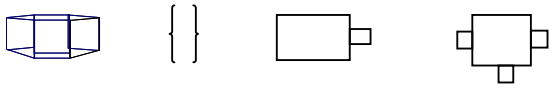
Con el fin de clarificar la terminología utilizada en este apartado de la tesis para la descripción y el modelado de las tareas realizas en los procesos de la ingeniería de la LPSD, a continuación se definen y/o describen algunos conceptos manejados.

3.3.1. Esqueletos.- El ingeniero del dominio debe ser suficientemente experto para determinar la plataforma común que comparte toda la familia de productos. Una plataforma de una LPS es "el conjunto de subsistemas software e interfaces que forman una estructura común desde la cual un conjunto de productos

pueden ser producidos y desarrollados eficientemente" [Meyer et al., 1997]. Las partes que componen la plataforma son denominados componentes básicos reutilizables [Clements et al., 2002]. Estas pueden incluir la arquitectura, los componentes software, los modelos diseñados, etc. En general, puede ser utilizado cualquier artefacto [Pohl et al., 2006]. La plataforma es la base sobre la cual los productos pueden ser creados adicionando características (variabilidad).

Nuestra plataforma consiste en un conjunto de esqueletos o plantillas de elementos arquitectónicos en el marco del metamodelo PRISMA. Por ello las partes que componen la plataforma de nuestra LPS serán un conjunto de esqueletos de componentes, conectores, aspectos e interfaces.

Los esqueletos son plantillas especificadas en el ADL de PRISMA que contienen "huecos" de valor semántico, que posteriormente serán rellenados con las características del dominio de aplicación del diagnóstico. Los esqueletos representan plantillas de los elementos arquitectónicos, sus aspectos e interfaces. Estos esqueletos son representados metafóricamente como lo muestra la figura X. Nótese que estos iconos están vacíos, por representar las plantillas con huecos que al ser rellenados conformarán los tipos. Por ello la metáfora visual de los tipos se representa como estos mismos iconos pero con color, i.e. rellenados con las características que fueron insertadas, como lo muestra la figura 4.



aspecto interfaz componente conector

Figura 4. Metáfora visual de esqueletos PRISMA

3.3.1.1 Esqueletos de los elementos arquitectónicos

Los esqueletos de los cuatro elementos arquitectónicos básicos de la LPSD representan: los componentes Motor de Inferencia, Base de Conocimientos y Módulo de Comunicación o Interfaz del Usuario, y el Conector Diagnóstico. Los aspectos necesarios para la definición de estos elementos arquitectónicos son los aspectos funcionales de cada uno de los componentes, así como el aspecto de coordinación del conector. Dichos aspectos utilizan el conjunto de servicios de las interfaces.

Componente Motor de Inferencia.- El Motor de Inferencia establece el control del sistema y es quien proporciona la estrategia general de resolución. En este componente los procesos de inferencia se llevan a cabo a través de estrategias de razonamiento. Se ha seleccionado en la LPSD las estrategias de razonamiento denominadas razonamiento deductivo, razonamiento inductivo y razonamiento diferencial, que son los que más se asemejan a los razonamientos utilizados por las personas cuando realizan un diagnóstico. Este elemento arquitectónico es independiente del conocimiento del dominio del caso de estudio; por esta razón, un mismo Motor de Inferencia puede ser la base de varios sistemas en diferentes dominios. El componente Motor de Inferencia importa un aspecto: el funcional, que define el proceso de inferencia del sistema.

Componente Base de Conocimientos.- Este componente contiene el conocimiento del dominio del caso de estudio, mediante reglas (cláusulas de Horn con cabeza) y propiedades de la entidad a diagnosticar. Dichas propiedades representan la estructura dinámica del conocimiento ya que su número puede verse incrementado a medida que se van relacionando las reglas del

dominio. Este componente tiene un almacén temporal de información dinámica, en donde ésta es almacenada. Esta información es proporcionada por el usuario (respuestas a preguntas formuladas) y por el proceso de inferencia (conclusiones obtenidas de todas las reglas disparadas) en forma de hechos. Cuando el proceso de un diagnóstico particular ha concluido, el contenido de la memoria de trabajo es eliminado, de forma que esta memoria queda limpia antes de iniciar un nuevo diagnóstico. Este componente importa un aspecto: el funcional, en donde se definen las reglas del dominio.

Componente Interfaz del Usuario.- El componente interfaz del usuario permite la comunicación entre el operario y el sistema. Este debe permitir el diálogo de forma sencilla entre el usuario y el sistema, aproximándose lo más posible al lenguaje natural. A través de éste, el operario ofrece datos iniciales al sistema o responde a preguntas formuladas por éste. Este componente contiene un aspecto que define su comportamiento: el funcional.

Conector Diagnóstico.- Este conector comunica a los componentes que une, e importa un aspecto: el de coordinación, que define las sincronizaciones entre los elementos arquitectónicos que coordina. Este elemento arquitectónico incluye en su comportamiento la coreografía del proceso de diagnóstico.

3.3.1.2 Esqueletos de los aspectos

Ya identificados los elementos básicos del modelo arquitectónico de un Sistema de Diagnóstico, es necesario identificar los *concerns* comunes al sistema (*crosscutting concerns*) con el objetivo de

separar cada uno de ellos en distintos aspectos para su reutilización.

El Sistema de Diagnóstico tiene como función el realizar el diagnóstico, a partir de datos observados y que son introducidos al sistema por un usuario. Esto hace emerger un *concern* de gran relevancia, el *concern* funcional.

Cabe destacar la necesidad de establecer los comportamientos de cada uno de los componentes de forma coordinada, así como la sincronización de los servicios enviados y recibidos entre ellos, por lo que surge la aparición del *aspecto de* coordinación.

Finalmente, se identifica un *aspecto* que cobra relevancia al estar presente en todos y cada uno de los elementos arquitectónicos del sistema propuesto, es el *aspecto de* distribución, ya que sus propiedades despliegan a los componentes y conectores en la(s) máquina(s).

Es importante separar cada uno de los *concerns* identificados en el modelo arquitectónico de diagnóstico, en una entidad reutilizable denominada aspecto, con el fin de mejorar la reutilización y mantenimiento de la arquitectura.

Los aspectos de cada uno de los elementos del modelo arquitectónico, se describen a continuación:

Aspectos del componente Motor de Inferencia.- Los dos aspectos que están presentes en el Motor de Inferencia son los aspectos funcional y de distribución.

Aspecto funcional.- Captura la semántica de la organización, mediante la definición de su estructura y comportamiento, importando la interfaz IInference.
Aspecto de distribución.- especifica la ubicación de la instancia en la que se encuentra.

Aspectos del componente Base de Conocimientos.- Los aspectos funcional y de distribución, son los aspectos que conforman una Base de Conocimientos.
Aspecto funcional.- captura la semántica de la organización del conocimiento del dominio que utiliza el sistema mediante la definición de su comportamiento, a través de la interfaz IDomain.
Aspecto de distribución.- especifica la ubicación de la instancia en la que se encuentra.

Aspectos del componente Interfaz del Usuario.- Los cuatro aspectos que son importados por el componente Interfaz del Usuario son el funcional, el de presentación, el de distribución y el de replicación.
Aspecto funcional.- consiste en la solicitud de los servicios que le otorga el sistema, importando la interfaz IOperator. Captura la semántica de la organización del componente, mediante la definición de su comportamiento.
Aspecto de distribución.- especifica la ubicación de la instancia desde la cual el usuario manipula el sistema asi como los servicios de movilidad para desplazarse a otra máquina, en caso de que se requiera, ya que se puede gestionar el sistema desde distintas máquinas, p.e. el usuario podrá utilizar el sistema desde su oficina, su casa, etc. Se hace notar que, aunque no se contempló en este trabajo, en el aspecto de distribución se podrían establecer las restricciones en las direcciones IP de acceso al sistema (p.e. el

usuario sólo puede acceder al sistema desde su oficina, pero no en su casa).

Aspecto de replicación.- permite el acceso simultáneo de diferentes usuarios, ya que es posible que haya más de un usuario utilizando el sistema (médico general o especialistas).

Aspectos del Conector Diagnóstico

Los aspectos presentes en el conector Diagnóstico son el de coordinación y de distribución.

Aspecto de coordinación.- define la sincronización entre elementos arquitectónicos durante su comunicación, mediante las interfaces IInference, IDomain e IOperator. También carece de estado, únicamente envía los servicios que se le solicitan a los usuarios o recibe los servicios del componente complejo, por lo tanto, sus servicios no tienen valuaciones asociadas, solamente participan en un proceso de modelado a través de un protocolo.

Aspecto de distribución.- especifica las características que definen la localización dinámica del elemento arquitectónico en el cual se integra. También define los servicios necesarios para implementar estrategias de distribución de los elementos arquitectónicos (como movilidad) con el objetivo de optimizar la distribución de la topología del sistema resultante [Ali, 2003].

3.3.1.3 Esqueletos de las interfaces.- Dado que los servicios que se envían o solicitan entre los elementos arquitectónicos del sistema, se han de publicar mediante interfaces que se reutilizarán en diferentes puertos y aspectos, en este apartado se identificarán a continuación las diversas interfaces del modelo arquitectónico, haciendo mención a los aspectos y puertos de los componentes o conectores que les da semántica.

IInference.- los servicios que se publican por esta interfaz son solicitados y recibidos por el componente Motor de Inferencia a través de su aspecto funcional y del Conector Diagnóstico a través de su aspecto de coordinación.
IDomain.- esta interfaz contempla los servicios entre el aspecto funcional del componente Base de Conocimientos y el aspecto de coordinación del Conector Diagnóstico.
IOperator.- el aspecto funcional del componente Interfaz del Usuario y el aspecto de coordinación del Conector Diagnóstico utilizan esta interfaz.

3.3.2 Artefacto.- Un artefacto (software) es cualquier producto de trabajo creado durante un proceso de desarrollo del software, tal como modelos, documentos de requerimientos, ficheros de código fuente, ficheros de configuración XML, DOC, etc., que resuelven problemas recurrentes en el desarrollo de software. Estos artefactos aceptarán cierta variabilidad, por lo que contarán con puntos de variabilidad que podrán ser configurados para adaptarlos a las características del problema concreto donde aplicarlos.

3.3.3 Activo.- Un activo es definido como una colección cohesiva de artefactos que resuelven un problema específico o un conjunto de problemas, así como la metainformación para facilitar su reutilización. Un activo se almacena como un fichero comprimido que empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos. Un activo de la LPSD está conformado por un esqueleto y su proceso de inserción de características.

3.3.4 Metainformación.- La metainformación que describe el activo, es definida por un fichero configurado en XML, que contendrá el modelo RAS del activo.

3.3.5 Modelo RAS del activo.- El modelo RAS de un activo es un modelo que conforma con el metamodelo RAS. Este modelo hace referencia a los artefactos contenidos en el activo, así como a actividades que definen procesos o guías de uso del mismo. Estas actividades permitirán configurar la variabilidad de los artefactos contenidos en el activo para obtener del mismo soluciones concretas. Este modelo se empaqueta junto al resto de artefactos dentro del fichero comprimido que representa al activo, conteniendo la identificación y la clasificación del activo, la descripción de sus artefactos, sus puntos de variabilidad y cómo configurarlos

3.3.6 Activo empaquetado.- Un activo empaquetado es un paquete software conformado por el propio activo y su correspondiente modelo RAS del activo. Un activo empaquetado es una colección cohesiva de artefactos que resuelven un problema específico o un conjunto de problemas, así como metainformación para facilitar su reutilización. De hecho, un activo se almacena como un fichero comprimido que empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos.

3.3.7 Base-Line.- La línea de productos de software requiere almacenar sus activos de software en repositorios. Un repositorio LPS es una base de datos especializada que almacena activos de software y facilita la recuperación y el mantenimiento de los activos de software. Su objetivo es asegurar la disponibilidad de activos para apoyar el desarrollo de productos de la LPS.

Tomando en cuenta esto, el conjunto de esqueletos, sus procesos de inserción de características y los modelos conceptuales del dominio de aplicación, son depositados en un repositorio que conforma la "*Base-Line*" de la LPSD. Por ello se define la *Base-Line* como un repositorio de artefactos software que van a ser seleccionados según la variabilidad del dominio, y usados como base sobre la que se añadirán las características ("features") que conformarán los tipos de artefactos software de la LPSD.

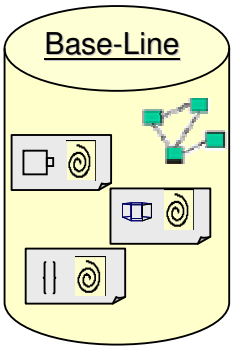


Figura 5. Metáfora visual de la Base-Line

3.3.8 Tipos de la LPSD.- De esta manera, se puede considerar que los tipos (artefactos software PRISMA) de la LPSD son los esqueletos rellenos de los elementos arquitectónicos. Es decir, son las plantillas o esqueletos conteniendo ya la información de cada una de las características o "features" del dominio de aplicación del diagnóstico, i.e. el dominio específico del caso de estudio.

Figura 6. Metáfora visual de tipos PRISMA

Cabe señalar que, como un mismo esqueleto puede ser aplicado a distintos tipos. En el estudio de campo realizado en esta tesis (ver capítulo X de esta tesis), contamos con un par de casos que comparten los mismos esqueletos. Estos casos de estudio son el diagnóstico televisivo y el diagnóstico educativo.

3.3.9 Arquitectura de la LPSD.- Uno de los puntos más importantes (o el más importante) de una LPS es la definición de la arquitectura de la línea de productos (ALP), debido a que esta arquitectura determinará el alcance de la línea de productos y las características de los productos que pueden desarrollarse. La ALP es la llave para la reutilización sistemática, ya que describe la estructura de los productos del dominio, mostrando sus elementos arquitectónicos y las relaciones entre los mismos. Definir una adecuada ALP requiere armonizar las cualidades que se persiguen para los productos de la LPS con la definición de elementos opcionales, alternativos o variables.

La arquitectura de una LPS (también denominada arquitectura del dominio) es una arquitectura software genérica que:

- describe la estructura de toda la familia de productos y no solamente la de un producto particular
- captura los aspectos comunes y variables de un familia de productos de software:
    - los aspectos comunes de la arquitectura son capturados por los componentes de software que son comunes a toda la familia
    - los aspectos variables de la arquitectura son capturados por los componentes de software que varían entre los miembros de la familia.

La arquitectura LPS debe ser instanciada cada vez que se desarrolla un producto de la línea.

Las instancias de los tipos (PRISMA) configurarán las arquitecturas software de la LPSD. Por lo tanto, la LPSD serán los sistemas de diagnóstico de cada uno de los dominios específicos.

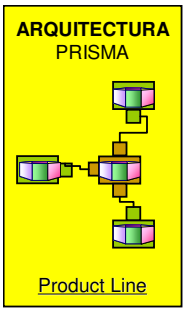


Figura 7. Metáfora visual de las arquitecturas de la LPSD

3.4 La variabilidad

Las características que definen la LPSD implican dos formas de variabilidad ortogonales. La **primera variabilidad** es dada por el propio proceso del diagnóstico, i.e. son las características del dominio del diagnóstico, las cuales son utilizadas por el ingeniero del dominio para crear los activos reutilizables (*assets)* (conformados por los esqueletos y los procesos de la ruta del árbol de decisión guiada por los puntos de variabilidad del dominio). Así mismo la primera variabilidad es utilizada por el ingeniero de la aplicación para elegir los activos de la *Base-Line* a través del árbol de decisión. La **segunda variabilidad** corresponde al campo de

aplicación del diagnóstico, i.e. son las características del dominio de aplicación, las cuales son utilizadas por el ingeniero de la aplicación para crear los tipos que conformaran el producto de la LPSD, a través de la inserción de la información de dichas características en los esqueletos.

Debido a que este trabajo de tesis se encuentra en el marco de las arquitecturas software, a las características del dominio del diagnóstico y del dominio de aplicación se les ha considerado, así mismo, como características de estructura y de comportamiento, respectivamente.

Las características de estructura son los puntos de variabilidad involucrados en el árbol de decisión, y que son utilizados para seleccionar los esqueletos.

Las características de comportamiento están relacionadas con los atributos, las evaluaciones, los protocolos y los *played_roles*, de los aspectos que contienen los elementos arquitectónicos Prisma. Esto es, son las características de estado, de XXXXXX, de procesos y de roles, respectivamente.

Por ello, las características de estructura son aquellas que metafóricamente permiten captar el estado de los elementos arquitectónicos PRISMA. Y similarmente, las características de comportamiento se corresponderán con la vista interna de dichos elementos.

Sin embargo, existen dos características que se comparten en los modelos del dominio de aplicación: las evaluaciones de los servicios de los aspectos y los protocolos de los aspectos. Por ello

para optimizar el proceso de inserción de las características, en lugar de repetir estas características en cada uno de los tipos, se fijan en los esqueletos de la *Base-Line*.

En la figura 8 se muestra un esquema de dicha clasificación:

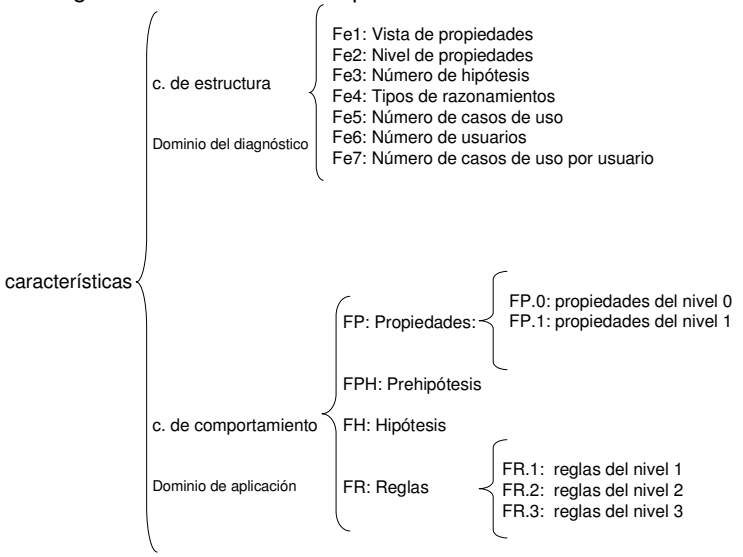


Figura 8. Clasificación de las características de la LPSD

Por ello, se define un modelo de la línea de producto software de diagnóstico LPSD como:

```
M-X = {Fe1, Fe2, Fe3, Fe4, Fe5, Fe6, Fe7, FP.i, FPH,
                    FH, FR.i},
```

donde M-X es el modelo del dominio específico X y el resto de elementos son las características (o "features") de dicha línea de producto.

Así mismo, el diseño de un programa se define como:

```
progLPSD-X =Fe1 Fe2 Fe3 Fe4 Fe5 Fe6 Fe7 FPi FPH FH FR.i
```

lo que significa que el programa LPSD-X tiene las características Fe1, Fe2, Fe3, Fe4, Fe5, Fe6, Fe7, FPi, FPH, FH, FR.i.

Por ello, el conjunto de programas que puede ser creado desde un modelo es una línea de producto. Es decir, el conjunto de programas que se corresponden con los elementos arquitectónicos que son creados desde el modelo M-DM del diagnóstico médico, es la línea de producto del caso médico.

Por ejemplo: para el caso de estudio del diagnóstico médico, se tiene:

```
X = DM = Diagnóstico Médico,
Fe1.= cambian las propiedades entre las entidades a
diagnosticar
Fe2 = 2 niveles de propiedades
Fe3 = 14 hipótesis/ 4 prehipòtesis
Fe4 = razonamiento diferencial
Fe5 = 3 casos de uso
Fe6 = 2 usuarios
Fe7 = usuario1 con 2 casos de uso, usuario2 con 1 caso
de uso
FP.0 = tos, fiebre, dificultad respiratoria, ........
FP.1 = tos seca, tos con flema, fiebre continua,
dificultad respiratoria grave, .......
FPH = ira, parotiditis,....
FH = bronquiolitis, neumonía, crup espasmódico,
paperas, parotiditis bacteriana, .........
FR.1 =
{fiebre = true and tos = true and dificultad
respiratoria=true} síndrome=”ira”
{fiebre = true and dolor a masticación = true and
parótidas anormales=true} síndrome=”parotiditis”
FR.2 =
{síndrome=”ira” } enfermedad= “bronquiolitis”
```

```
{síndrome=”ira” }  enfermedad= “neumonía”
{síndrome=”ira” }  enfermedad= “crup espasmódico”
{síndrome=” parotiditis” }  enfermedad= “paperas”
{síndrome=” parotiditis” }   enfermedad= “parotiditis
bacteriana”
.......
FR.3 =
{ fiebre continua=true  and  fiebre mayor a 38 =true
and  dolor y crecimiento de parótidas =true and dolor a
masticación espontáneo agudo=true }     enfermedad=
“paperas”
```

Por otro lado, debido a que las características pueden ser distinguidas como constantes o funciones, las dos variabilidades de esta tesis son clasificadas como:

La primera variabilidad (representada por las características del dominio del diagnóstico), es manejada tanto en la etapa de la ingeniería del dominio como en la de la ingeniería de la aplicación. Esta variabilidad comprende la creación de los activos de la *Base-Line* por el ingeniero del dominio, y así mismo implica puntos de variabilidad que al ser seleccionados por el ingeniero de la aplicación, se obtendrá el grupo de activos correspondientes al caso aplicado. Estos activos se corresponden con las hojas del árbol de decisión, como lo muestra la figura X.

La figura 9 presenta los esqueletos de la LPSD de acuerdo a la selección realizada a través del árbol de decisión y conforme al estudio de campo realizado (ver apéndice B).

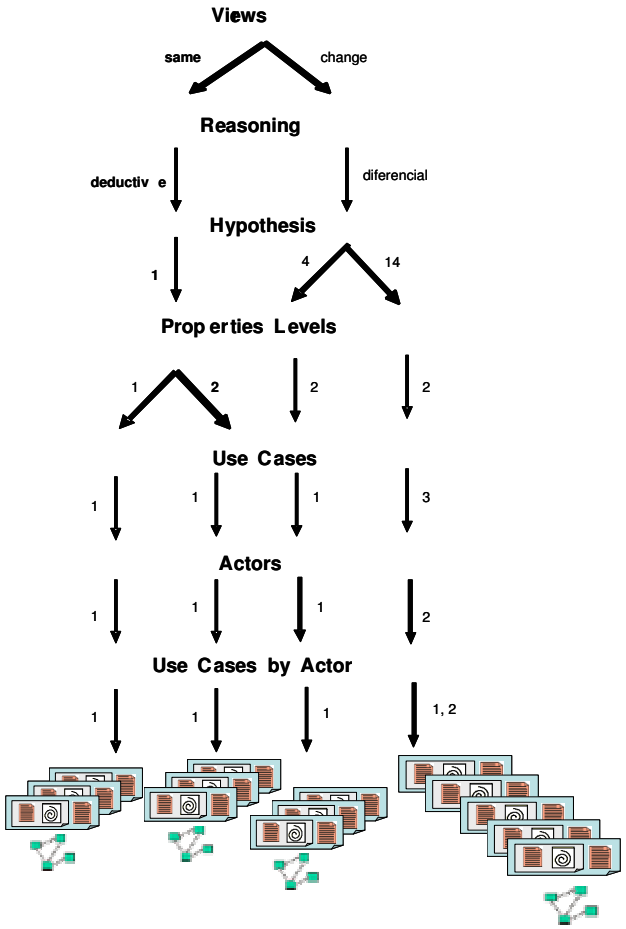


Figura 9 Selección de los esqueletos de elementos arquitectónicos de la LPSD, a través del árbol de decisión

Con lo anteriormente descrito, es posible considerar que las características de la primera variabilidad, en la etapa de la ingeniería del dominio son las características de estructura, y en la etapa de la ingeniería de aplicación son consideradas como puntos de variabilidad para elegir los esqueletos, mismos que se convierten en características constantes.

De esta manera, las características consideradas como constantes (esqueletos) representan programas base. Por ejemplo:

E−MI−DM,

que significa "un programa E-MI-DM", que es, a su vez, una característica constante E-MI-DM. Cabe aclarar que E-MI-DM es la nomenclatura utilizada para el esqueleto del motor de inferencia correspondiente al caso del diagnóstico médico.

La segunda variabilidad (representada por las características del dominio de aplicación), es aplicada por el ingeniero de la aplicación, de tal forma que se irán rellenando los esqueletos seleccionados en el paso anterior y que darán lugar a la creación de los tipos respectivos. Estos tipos incluirán las características consideradas como funciones, las cuales representan refinamientos de programas que extienden un programa que es recibido como entrada. Por ejemplo:

F· E−MI−DM,

que significa: "agregar la característica F al programa E-MI-DM", donde · denota la aplicación de la función.

Por ello se tiene que:

$E\text{−}MI\text{−}DM_0 = FP.0 \cdot E\text{−}MI\text{−}DM_1 = FP.0\ (E\text{−}MI\text{−}DM_1)$

$E\text{−}MI\text{−}DM1 = FP.1 \cdot E\text{−}MI\text{−}DM_2 = FP.1\ (E\text{−}MI\text{−}DM_2)$

$E\text{−}MI\text{−}DM_2 = FPH \cdot E\text{−}MI\text{−}DM_3 = FPH\ (E\text{−}MI\text{−}DM_3)$

$E\text{-}MI\text{-}DM_3 = FH \cdot E\text{-}MI\text{-}DM_4 = FH\ (E\text{-}MI\text{-}DM_4)$

$E\text{-}MI\text{-}DM_4 = FR.1 \cdot E\text{-}MI\text{-}DM_5 = FR.1\ (E\text{-}MI\text{-}DM_5)$

$E\text{-}MI\text{-}DM_5 = FR.2 \cdot E\text{-}MI\text{-}DM_6 = FR.2\ (E\text{-}MI\text{-}DM_6)$

$E\text{-}MI\text{-}DM_6 = FR.3 \cdot E\text{-}MI\text{-}DM_7 = FR.3\ (E\text{-}MI\text{-}DM_7)$

Para aclarar que un tipo se obtiene mediante la inserción de características, en la figura 10 se presenta una metáfora gráfica del proceso de inserción de las características en los esqueletos seleccionados de la *Base-Line* para crear sus tipos. Los esqueletos se representarán en documentos XML a los cuales se les insertarán una a una las características, a través de transformaciones XSLT, obteniendo así los tipos que serán también documentos XML.

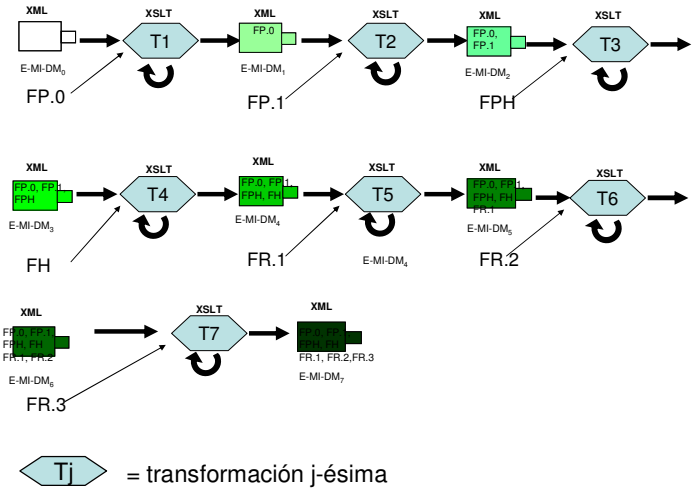


Figura 10. Proceso de inserción de las características en los esqueletos para forman los tipos

La figura 11 muestra el proceso desde la inserción de características en los esqueletos, hasta la configuración de la arquitectura del modelo específico de la LPSD.

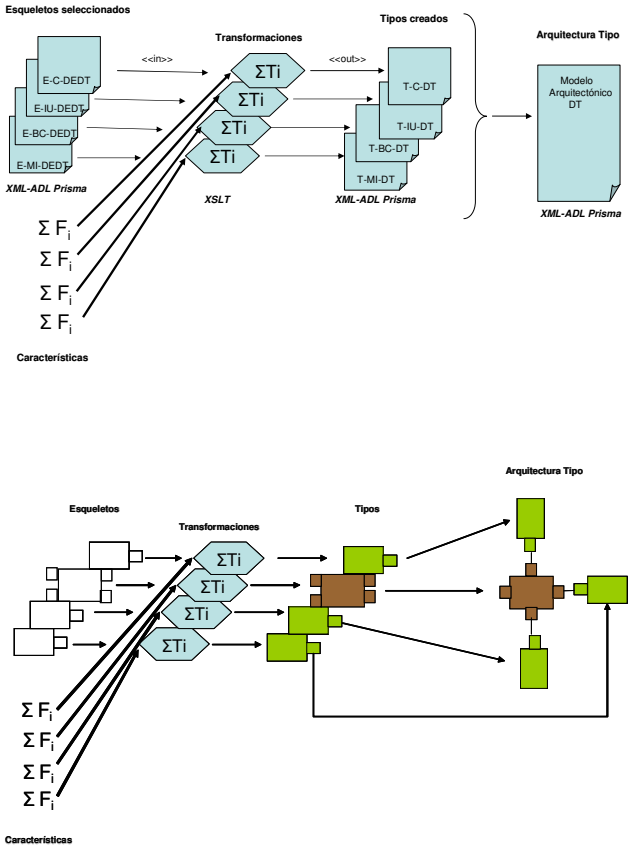


Figura 11. Proceso desde la inserción de características en los esqueletos, hasta la configuración de la arquitectura

Por lo anteriormente expuesto se observa que existe variabilidad tanto en los elementos arquitectónicos básicos que conforman un sistema de diagnóstico, como en el modelo arquitectónico. Por ello, en esta tesis se ha propuesto un procedimiento para construir arquitecturas software en el dominio del diagnóstico a partir de los requisitos software.

3.5 Metodología BOM para construir arquitecturas software en sistemas de diagnóstico a partir de los requisitos software

Este proceso parte de la especificación de requisitos resultante de la etapa de obtención de requisitos. Para ello, en este trabajo se realizó un análisis del dominio del diagnóstico para conocer cuáles son los requisitos funcionales que el producto final ha de satisfacer.

En esta etapa, es primordial tener presente la definición de componente. Existen dos tendencias, una es más orientada a implementación y otra es más genérica. La primera recoge definiciones relacionadas con el hecho de que una componente es un paquete de código [Sou, 1999], mientras que la segunda define un componente como un artefacto que ha sido desarrollado específicamente para ser reutilizado. Adicionalmente dicho artefacto reutilizado es identificado como un activo o "*asset*" en la aproximación del desarrollo de las líneas de producto software [Clements et al., 2002].

De estas dos tendencias, en este trabajo se eligió la segunda, ya que la hace independiente a la solución del problema, manteniendo un alto nivel de abstracción. Esta segunda definición permite que un componente sea tanto un caso de uso como una clase o cualquier otro elemento que surja durante el proceso de desarrollo.

Los casos de uso constituyen una división del sistema basada en la funcionalidad. El diagrama de casos de uso muestra las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuario).

El concepto de escena usado aparece en el trabajo de [Noriega, 1998]. Este trabajo considera la escena como un criterio para la detección de componentes en la especificación de requisitos. El concepto de escena es definido con un enfoque PRISMA [Pérez, 2003b] al considerar que "Una escena se caracteriza por las tareas que se desempeñan en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla"

Los elementos arquitectónicos PRISMA [Pérez, 2003] se obtienen identificando escenas funcionales del sistema y asignando a cada elemento una escena funcional, dependiendo de si la escena es simple o compleja, el elemento será un componente (componente simple) o un sistema (componente complejo), respectivamente.

Un diagrama de secuencias muestra la interacción de un conjunto de objetos de una aplicación a través del tiempo. Esta descripción es importante porque puede dar detalle a los casos de uso, aclarándolos al nivel de mensajes entre los objetos existentes,

como también muestra el uso de los mensajes de las clases diseñadas en el contexto de una operación. Se puede crear un diagrama de secuencias por cada caso de uso. Con los diagramas de secuencia se puede modelar la implementación del escenario.

En el contexto de PRISMA, cada diagrama de secuencia de las realizaciones del caso de uso, se corresponderá con un *played_role* del aspecto funcional de los componentes y del aspecto de coordinación de los conectores, del modelo arquitectónico configurado.

Tomando en cuenta lo anteriormente expuesto, la propuesta metodológica en esta tesis para configurar arquitecturas software consiste en lo siguiente:

1.- Los requisitos funcionales de mayor abstracción que el producto final ha de satisfacer son especificados mediante un diagrama con todos los casos de uso.

2.- Los casos de uso son representados por escenas modeladas como diagramas de secuencias, de la siguiente forma:

- crear un diagrama de secuencias para cada caso de uso, describiéndose el proceso general del sistema (vista externa de caja negra). En este diagrama de secuencias únicamente se establece la interacción entre el sistema y el usuario final.
- crear los diagramas de secuencias de las realizaciones de los casos de uso, con el fin de describir cómo se realiza cada caso de uso (vista interna del funcionamiento del sistema). Para ello se deberán de identificar los elementos arquitectónicos que definirán la arquitectura del modelo, a través de la detección de

los componentes del sistema y de los conectores que permiten su comunicación.

3.- Con los elementos arquitectónicos definidos por cada caso de uso, se configura el modelo arquitectónico final del sistema de diagnóstico, i.e. un producto de la LPSD, tomando en cuenta el siguiente criterio:

- conservar los conectores tipo (donde el número de conectores del modelo arquitectónico final es igual al número de casos de uso).
- unir los componentes tipo (el número de puertos del componente final será igual al número de casos de uso en los que es utilizado dicho componente; i.e. un puerto del componente final se corresponde con el puerto del componente tipo del caso de uso respectivo).

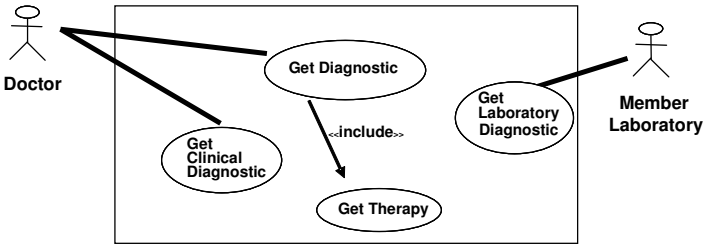
4.- En consecuencia, en la arquitectura final del modelo propuesto, los elementos arquitectónicos básicos que a continuación se listan, son construidos de la siguiente manera:

- **conector Diagnóstico**.- habrá un conector por cada caso de uso,
- **componente Motor de Inferencia** - el número de puertos del Motor de Inferencia es igual al número de casos de uso,
- **componente Base de Conocimientos**.- el número de puertos de la Base de Conocimientos es igual al número de casos de uso,
- **componente Interfaz del Usuario**.- el número de componentes Interfaz del Usuario es igual al número de actores del diagrama de casos de uso. Así mismo los roles que desempeña un usuario (número de casos de uso de un actor)

se corresponde con los puertos que tiene el componente Interfaz del Usuario respectivo.

Con el fin de aclarar este proceso de construcción de arquitecturas de sistemas de diagnóstico, que representan la LPSD, a continuación se presenta un ejemplo: el caso de estudio del diagnóstico médico.

1.- Especificar los requisitos funcionales de mayor abstracción mediante un diagrama de tres casos de uso.



2.- Representar cada caso de uso, a través de una escena, modelada con un diagrama de secuencias del caso de uso y con un diagrama de secuencias de las realizaciones del caso de uso.

CU1: Diagnóstico clínico

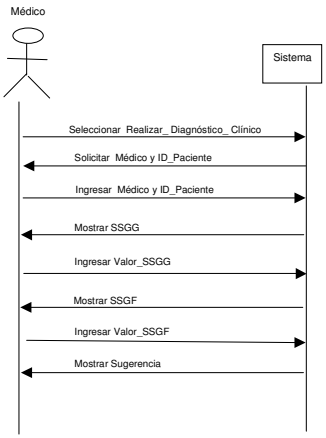


Diagrama de secuencia del caso de uso: Realizar diagnóstico clínico

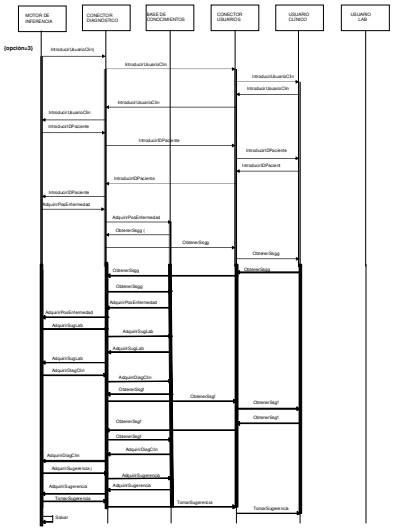


Fig. 21 Diagrama de secuencia de las ¨del¨caso de uso Realizar diagnóstico clínico

CU2: Diagnóstico de laboratorio

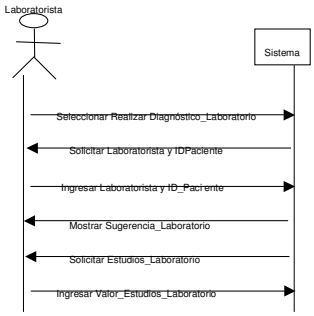


Diagrama de secuencia del caso de uso: Realizar diagnóstico de laboratorio

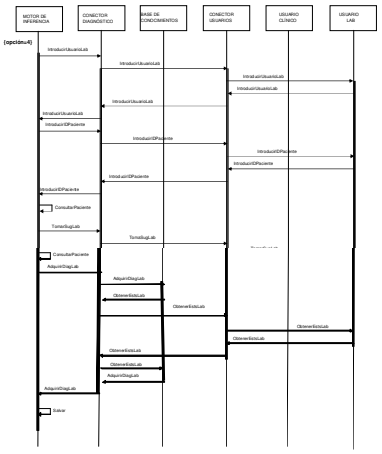
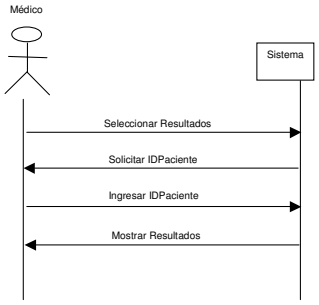


Fig. 22 Diagrama de secuencia de las realizaciones del caso de uso Realizar diagnóstico de laboratorio

CU3: Resultados del diagnóstico



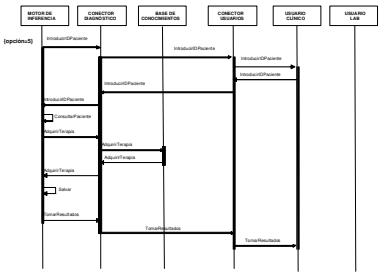
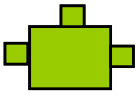


Fig. 23 Diagrama de secuencia de las realizaciones del caso de uso Resultados del diagnóstico

Diagrama de secuencia del caso de uso:
Obtener resultados del diagnóstico

3.- Construir los elementos arquitectónicos:

En este ejemplo tenemos tres casos de uso y dos actores, donde uno de los actores desempeña dos roles y el otro actor desempeña un sólo rol. Por ello le corresponden los siguientes elementos arquitectónicos:

Motor de Inferencia (con 3 puertos): 

Base de Conocimientos (con 3 puertos): 

Interfaz del Usuario 1 i.e. el médico (con 2 puertos) 

Interfaz del Usuario 2 i.e. el encargado del laboratorio (con 1 puerto) 

Conector Diagnóstico 1:

Conector Diagnóstico 2:

Conector Diagnóstico 3:

4.- Configurar el modelo arquitectónico:

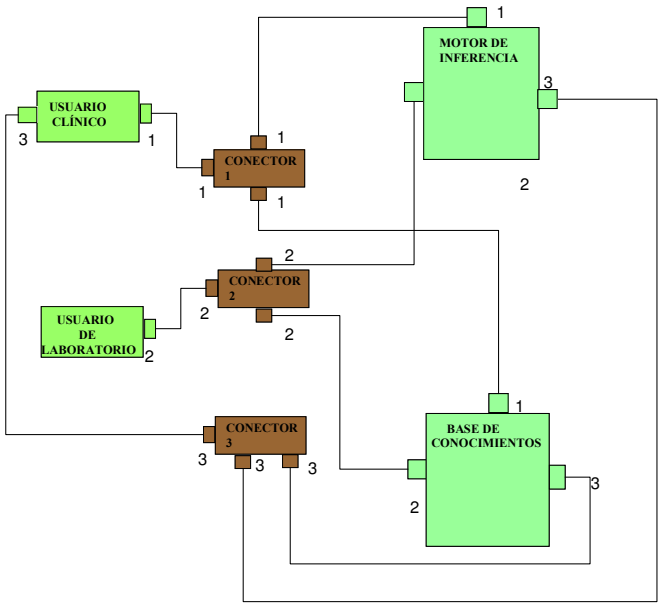


Figura 12. Modelo de un sistema de diagnóstico médico

3.6 Análisis del dominio del diagnóstico

El análisis del dominio orientado a características (Feature Oriented Domain Análisis-FODA) es una metodología de la ingeniería del dominio desarrollada por el SEI (Software Engineering Institute) de la CMU (Carnegie Mellon University) [Kang et al., 1990]. Se enfoca al desarrollo de LPS a través de un proceso de ingeniería del dominio estructurado, basado en la notación de modelos de características.

La programación orientada a características (Feature Oriented Programming-FOP) es un paradigma de las LPS donde las características son los bloques de los productos construidos, i.e. los componentes básicos reutilizables. A través de las características los diferentes productos pueden ser distinguidos y definidos en una LPS [Batory et al., 2004]. En general, una LPS es identificada por un conjunto de características que soporta, donde un producto es obtenido como la síntesis de algunas de esas características en la base o plataforma de la LPS.

El modelado de características (*feature*) es un producto principal de la ingeniería del dominio. Una *feature* es "una característica del producto que es usado para distinguir un producto de una familia de productos relacionados" [Batory et al., 2004].

El modelo de características identifica la LPS en términos de la variabilidad soportada. Entre las distintas variantes disponibles del dominio, el analista del dominio de la LPS debe seleccionar (incluir o excluir) las características de acuerdo a las alternativas que son presentadas.

Los modelos de características son la clave técnica de las líneas de producto para desarrollar software [Batory et al., 2006]. Estos son modelos formales que usan características para especificar productos. Un modelo de características es un conjunto de características ordenado jerárquicamente. Un diagrama de características es una representación gráfica de un modelo de características.

Después de haber realizado un estudio y un análisis del dominio del diagnóstico, se puede concluir que el diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a través de sus propiedades (variables observables). Así mismo se han observado siete características o fuentes de variabilidad, y que a continuación se enuncian:

Dominio del diagnóstico =

{

vista de las propiedades// una entidad puede tener siempre las mismas propiedades (misma vista), o bien una entidad puede tener diferentes propiedades (diferentes vistas) durante el proceso del diaganóstico,

nivel de las propiedades// las propiedades de las entidades pueden tener uno o varios niveles de abstracción. Las reglas que relacionan las propiedades de la entidad tienen n-1 niveles, donde n es el nivel de las propiedades de la entidad,

número de hipótesis// es el objetivo del diagnóstico. Este permite tener una o varias hipótesis, las cuales deberán ser validadas hasta obtener una única hipótesis,

tipo de razonamiento// indica la forma en que las reglas se aplican para inferir un diagnóstico final. Los razonamientos realizados pueden ser el deductivo y el inductivo,

número de casos de uso// indica la división del sistema basada en la funcionalidad; i.e. las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuarios finales),

número de usuarios// permite representar al número de usuarios finales del sistema,

número de casos de uso por usuario// un usuario final puede acceder a uno o varios casos de uso.

}

Dado que los modelos utilizados en el análisis de líneas de producto permiten el modelado de requisitos tanto comunes como variables en la línea de productos, se han modelado los requisitos variables de la línea de productos software de diagnóstico (LPSD) a través del modelo de características de la LPSD, que identifica la LPSD en términos de la variabilidad.

La figura 13 representa el modelo de características de la LPSD usando una notación similar a [Czarnecki et al., 2005] y a [Batory et al., 2006], a excepción de intercambiar los conectivos “and” y “or” por así convenir a nuestros intereses (notación aplicada a los grafos de las reglas o relaciones entre las entidades de las propiedades). El modelo organiza características en una composición jerárquica donde la opcionalidad y la obligatoriedad están presentes. Los productos finales o esqueletos son caracterizados en términos de estas características, y el ámbito de la LPS es dado por su característica en dicho modelo.

Consecuentemente, una LPS debe soportar la variabilidad de esas características que tienden a diferir de producto a producto.

Además de las relaciones jerárquicas, los modelos de características también permiten restricciones de cruce en el árbol. Tales restricciones son típicamente sentencias de inclusión o exclusión de la forma IF característica C es incluida (i.e. seleccionada), entonces las características A y B deben ser incluidas (o excluidas). Un ejemplo de una restricción de cruce en el árbol en nuestra línea de producto es que la entidad que tiene una vista en donde cambian sus propiedades implica obtener varias hipótesis y realizar un razonamiento diferencial.

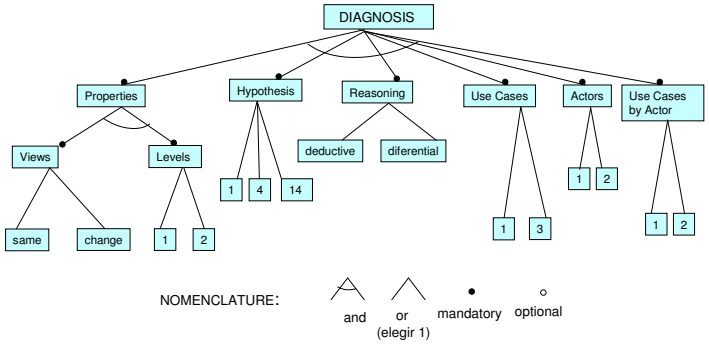


Figura 13. Modelo de características del dominio del diagnóstico

La especificación del modelo de características del dominio del diagnóstico, tiene la siguiente sintaxis:

```
Diagnosis:  properties  hypothesis  reazonings
use_cases  actors  use_cases_by_ actor ;
   Properties: views  levels;
   Views: same  change
   Levels: 1  2
   Hypothesis:  1  4  14
   Reazonings:  deductive  diferential
   Use_cases:  1  3
   Actors:  1  2
   Use_cases_by_actor:  1  2
```

```
//Restrictions of tree cross
Same  implies 1 hypothesis    deductive;
Change implies 4 hyphotesis 14 hypothesis  diferential;
```

Con el fin de ayudar a la comprensión del modelo de características de la figura X, se definirán cada uno de las clases o conceptos contemplados en dicho modelo.

a) Diagnóstico: el diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a través de sus propiedades (variables observables). Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una anomalía o propiedad a partir de datos observables y razonamientos.

b) Usuario: un usuario se define como una persona que solicita realizar un diagnóstico.

c) Casos de Uso: un caso de uso se define en UML como "un conjunto de acciones que realiza el sistema y que tiene un resultado observable y de interés para un actor particular del mismo" [OMG]. Los casos de uso constituyen una especificación del sistema basada en la funcionalidad. El diagrama de casos de uso muestra las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuario).

c) Casos de uso por usuario: un usuario puede acceder a uno o más casos de uso.

e) Hipótesis: una hipótesis es el objetivo o resultado del proceso de diagnóstico. Cuando se identifica el problema o disfunción que

presenta una entidad, se obtiene una hipótesis validada. El diagnóstico realizado a una entidad puede llegar a tener una sola hipótesis o bien varias hipótesis.

f) Propiedades: las propiedades de una entidad son la caracterización mediante datos de las causas que provocan una anomalía o disfunción de la misma. Una entidad puede estar caracterizada siempre por las mismas propiedades o bien pueden variar esas propiedades, i.e. la entidad está caracterizada por un conjunto específico de propiedades que varían según la hipótesis a validar. Por ello las propiedades se clasifican en dos tipos: propiedades iguales y propiedades distintas.

g) Razonamientos: los razonamientos o estrategias de razonamiento son la forma en que el mecanismo de inferencia realiza el diagnóstico. El mecanismo de inferencia aplica estrategias de razonamiento, utilizando la información del dominio para obtener las propiedades. Los razonamientos más utilizados para las tareas de diagnóstico son el deductivo, el inductivo y el diferencial.

Las fuentes de variabilidad (características de la LPSD) observadas en el modelo de características del diagnóstico, son mostradas a través del árbol de decisión de la figura 14.

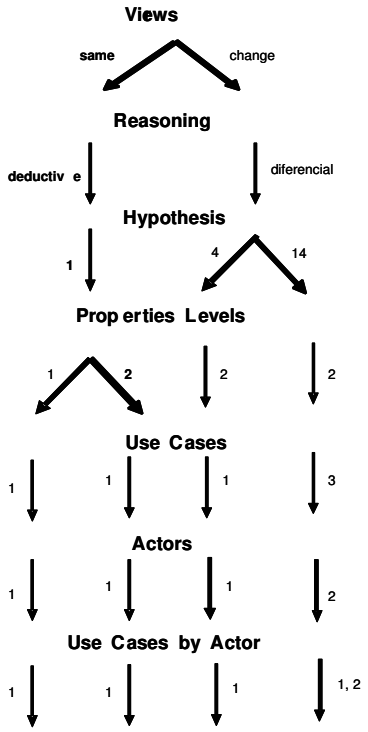


Figura 14 Árbol de decisión de las características de la LPSD

Cada una de las características de la LPSD involucradas en el árbol de decisión y en el modelo de características son comentadas a continuación:

3.6.1 Vista de las propiedades.- Una entidad puede estar caracterizada siempre por las mismas propiedades (misma vista) o bien pueden variar esas propiedades (diferentes vistas), i.e. la entidad está caracterizada por un conjunto específico de propiedades para cada hipótesis. Por ello las propiedades se

clasifican en dos vistas: mismas propiedades y propiedades que cambian.

3.6.2 Número de hipótesis.- Una hipótesis es el objetivo o resultado intermedio del proceso del diagnóstico. Cuando se identifica el problema o estado que presenta una entidad, se obtiene una hipótesis validada. El diagnóstico realizado a una entidad puede llegar a tener una sola hipótesis o bien varias hipótesis.

Cuando las propiedades de una entidad no cambian, se genera una sola hipótesis. Sin embargo cuando las propiedades de una entidad si cambian, se generan varias hipótesis, que deberán ser validadas para llegar a un resultado del diagnóstico.

3.6.3 Nivel de las propiedades.- El proceso de diagnóstico contempla varios niveles de abstracción de las propiedades de una entidad, y su relación entre las propiedades a través de reglas, de forma que:

- Las propiedades del nivel n y las propiedades del nivel n+1, se relacionan a través de las reglas de nivel n+1,
- Las propiedades del nivel 0 obtienen su valor del usuario,
- Las propiedades del nivel n+1 se infieren aplicando las reglas del nivel n+1,
- El valor de las propiedades superiores al nivel 0, se puede obtener del usuario o bien inferirse a través de las reglas.

3.6.4 Número de los casos de uso, número de los usuarios finales y número de roles que desempeñan los usuarios finales.- Los casos de uso son un punto muy importante en la variabilidad, dado que al especificar los requisitos funcionales de mayor abstracción mediante un diagrama de casos de uso, se

identifican el nombre de las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (número de usuarios finales o actores, y número de casos de uso a los que puede acceder el usuario final).

3.6.5 Tipo de razonamientos.- Las estrategias de razonamiento que forman parte de la variabilidad en la LPSD son simulaciones de los tipos de razonamientos más comunes que tiene una persona que realiza un diagnóstico Éstos son:

3.6.5.1 El razonamiento deductivo o guiado por los datos (también llamado razonamiento con encadenamiento hacia delante o razonamiento "forward") consiste en enlazar los conocimientos a partir del uso de datos (propiedades de la entidad a diagnosticar) con el fin de obtener una solución de un problema. Dado que en este razonamiento se generan nuevas propiedades o hechos, existen dos formas de tratarlos, que son: profundidad cuando un hecho en cuanto se genera se introduce a la Base de Conocimientos, o en anchura cuando no se incorporan los hechos a la Base de Conocimientos hasta que se ha terminado de aplicar la misma. En general, las soluciones pueden ser más de una. De un hecho así deducido se puede asegurar su verdad. La ventaja desde el punto de vista técnico de utilizar este modo de encadenar el conocimiento es su sencillez y que la entrada de datos es única y se da al principio del programa. Este encadenamiento está basado en el "modus ponens" de la lógica formal que dice que: Si conocemos la regla: si A entonces B, y A es cierto, entonces podemos deducir que B es también cierto.

El encadenamiento hacia adelante se desarrolla según el siguiente procedimiento:

- Hasta que ninguna regla produzca una nueva afirmación o la hipótesis sea identificada:
    - Para cada regla:
        - Trátese de corroborar cada antecedente de la regla mediante su apareamiento con hechos conocidos;
        - Si se corroboran todos los antecedentes de la regla, hágase valer el consecuente, a menos que ya exista una afirmación idéntica;
        - Repítase el procedimiento para todos los apareamientos y alternativas de sustitución.

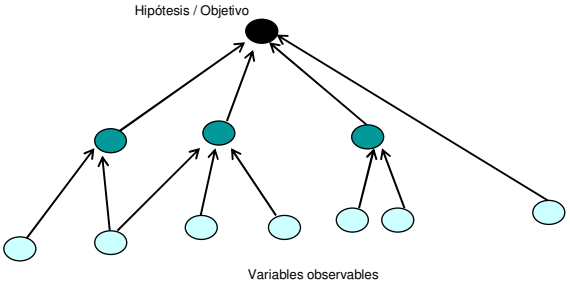


Figura 15. Grafo que muestra el razonamiento deductivo

3.6.5.2 El razonamiento inductivo o guiado por los objetivos (también llamado razonamiento con encadenamiento de reglas hacia atrás o razonamiento "backward") consiste en comprobar que un objetivo es cierto en base a unos hechos que forman el universo del sistema. Este encadenamiento tiene su fundamento en el "modus tollens" de la lógica formal que dice que si conocemos la regla: Si A entonces B y también conocemos que A es falso, entonces podemos inducir que B también lo es.

El encadenamiento hacia atrás se desarrolla según el siguiente procedimiento:

- Hasta que todas las hipótesis se hayan identificado y ninguna se pueda comprobar o hasta que las hipótesis sean verificadas:
    - Para cada hipótesis:
        - Para cada regla cuyo consecuente coincida con la hipótesis en cuestión,
            - Inténtese corroborar cada uno de los antecedentes de la regla mediante su apareamiento con afirmaciones de la memoria en funcionamiento o mediante encadenamiento regresivo a través de otra regla, creando nuevas hipótesis. Asegúrese de verificar todas las alternativas de unificación y sustitución.
            - Si todos los antecedentes de la regla logran ser corroborados, notifíquese el éxito y concluya que la hipótesis es verdadera.

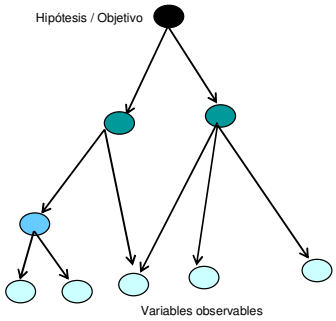


Figura 16. Grafo que muestra el razonamiento inductivo

3.6.5.3 El razonamiento diferencial.- Un diagnóstico diferencial se presenta cuando hay que comparar entre dos o más posibilidades diagnósticas o hipótesis. En el razonamiento diferencial se lleva a cabo el siguiente proceso: primero se realiza el razonamiento deductivo con el fin de llegar a un diagnóstico; si se llegase a dos o más posibilidades diagnósticas, se realizará el razonamiento diferencial, invocando al razonamiento inductivo para reformular preguntas al usuario o buscar datos que permitan determinar cuál de las posibilidades diagnósticas es la más certera al problema analizado. i.e. verificar las soluciones o hipótesis.

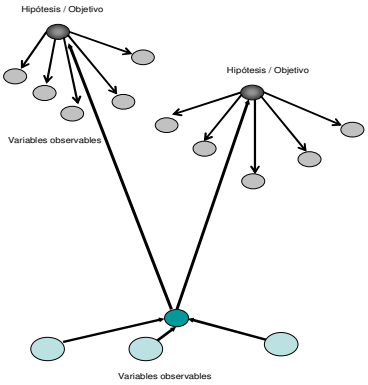


Figura 17. Grafo que muestra el razonamiento diferencial

3.6.6 Grafos de las reglas entre propiedades de las entidades

Las propiedades de las entidades al relacionarse entre sí pueden cumplir ciertas reglas. Estas reglas son cláusulas de Horn con cabeza, las cuales son del tipo IF-THEN. La Base de Conocimientos contiene este tipo de reglas que representan la información del dominio de aplicación del diagnóstico y se representan mediante una estructura de árbol.

Un ejemplo de grafo es mostrado en la figura X.

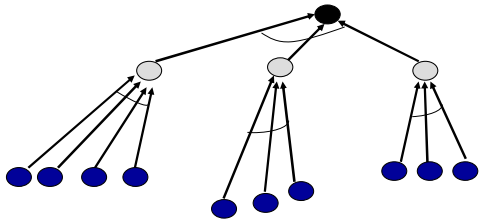


Figura 18 Grafos de las reglas entre propiedades de las entidades

El Motor de Inferencia al aplicar las estrategias de razonamiento, recorre el árbol, i.e. aplica las reglas. La figura X representa una metáfora visual de la estrategia de razonamiento deductiva.

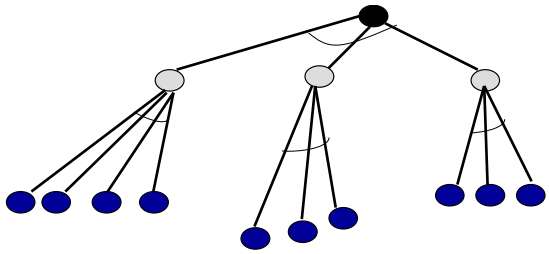


Figura 19 Metáfora visual de la estrategia de razonamiento deductiva

Una estructura de árbol, como la generada por el encadenamiento de reglas, puede ser recorrida en dos sentidos:

- en anchura (niveles) , donde se contemplan todas las posibilidades de un nudo antes de pasar al siguiente nodo o bifurcación del árbol (ABCDEFG)

- en profundidad (ramas).- no se toma otra posibilidad en un nodo hasta que no se ha desarrollado completamente una rama (ABDECFG)

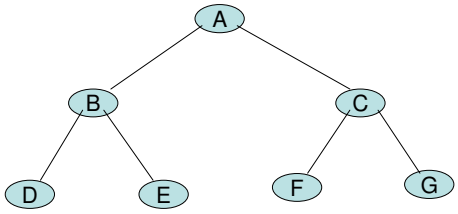


Figura 20. Recorrido de un árbol

En la LPSD, el Motor de Inferencia aplica el recorrido en profundidad en el árbol de los grafos que representan las reglas de la Base de Conocimientos.

La figura 21 muestra el modelo conceptual del dominio del diagnóstico sobre el cual se especifica la ontología del diagnóstico, involucrando características y particularidades del diagnóstico en el que se desenvolverá el sistema que se desea construir.

.

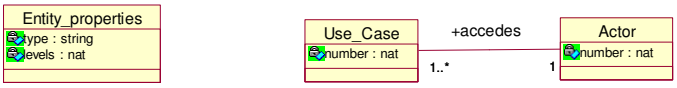


Figura 21 Modelo del dominio del diagnóstico

Los conceptos mostrados en el modelo de la figura 21, a través de un modelo de clases UML, son los necesarios para el modelado de un sistema basado en el conocimiento desde una aproximación orientada al diagnóstico. Los conceptos de la figura 21 que se corresponden con la perspectiva CIM del diagnóstico y describen elementos propios del diagnóstico.

3.7 Análisis del dominio de aplicación

En este trabajo se ha contemplado una familia de sistemas de diagnóstico en diferentes dominios específicos de aplicación. Algunos ejemplos de estos dominios son comentados y analizados en el apéndice B de esta tesis, para mostrar la aproximación de la LPSD.

Algunas de las características seleccionadas del modelo de características deben de ser explicitadas de acuerdo al caso de estudio en el que se realiza la aplicación del dominio. Para ello, el analista de la aplicación del dominio deberá proporcionar la información específica del caso de estudio, dando valor a las características que son necesarias para conformar la ontología del dominio específico. Dichas características son enunciadas a continuación:

Dominio de aplicación =

{

nombre y tipo de las propiedades// por nivel de abstracción, las propiedades de la entidad a diagnosticar adquieren un nombre, así como los tipos posibles de dichas propiedades,

reglas por niveles de abstracción// por niveles, las reglas que relacionan las propiedades de la entidad adquieren el nombre y tipo de las propiedades que están presentes en el antecedente y el consecuente de cada una de las reglas,

nombre y tipo de las hipótesis (y en su caso, de prehipótesis)// se adquiere el nombre y el tipo de las hipótesis probables (y en su caso, prehipótesis) que produce el proceso del diagnóstico,}

A continuación se muestra un ejemplo de las propiedades, reglas e hipótesis de un caso de estudio:

```
properties of level 0:  cough, fever
properties of level 1: dry cough, constant_fever
pre-hypotheses: warth, parotiditis
hypotheses: pneumonia, bronchitis
rules:
      IF (cough=true and fever=true and
      respiratory_dificulty=tue)
      THEN syndrome=warth

      IF (fever=true and pain_to_ mastication=true
      and abnormal_parotids= ture)
      THEN  syndrome=parotiditis
```

La figura 22 muestra un modelo conceptual del dominio de aplicación. Dicho modelo corresponde al caso del diagnóstico médico.

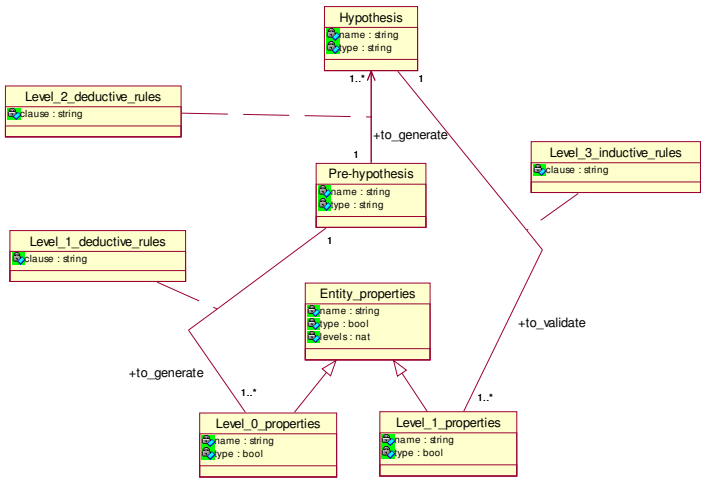


Figura 22. Modelo conceptual del dominio de aplicación (caso: diagnóstico médico)

Capítulo 4

DESARROLLO DE LA APROXIMACIÓN MDA PARA LÍNEAS DE PRODUCTO ORIENTADAS AL DIAGNÓSTICO

En este capítulo se presenta la segunda parte del objetivo de esta tesis, que consiste en abordar el dominio del diagnóstico de forma genérica, a través del desarrollo de una aproximación metodológica basada en MDA para crear líneas de producto software orientadas al diagnóstico, denominada BOM (Base-Line Oriented Model Diagnosis): un Generador Automático de Sistemas de Diagnóstico basado en Líneas de Producto.

4.1 Desarrollo de BOM

Esta tesis se basa en la aproximación de Líneas de Producto de Software, paradigma enriquecido en los últimos años por la Ingeniería Dirigida por Modelos. Esta aproximación se basa en la idea de producir familias de productos software a partir de un conjunto de activos reutilizables (*assets*), de una manera preestablecida, ofreciendo tecnologías para llevar a cabo la reutilización y la automatización en los procesos de software. Desarrollar un producto de la LPS más que una creación es una cuestión de ensamblaje y configuración. Para cada LPS hay un plan de producción bien definido que especifica el proceso de obtención de cada producto.

La aproximación propuesta en esta tesis está pensada para automatizar los procesos de desarrollo, a través de la creación de LPS en dominios de aplicación específicos. Para ello, se ofrece una solución basada en un desarrollo iterativo y desarticulado de activos, que permite automatizar las tareas del desarrollo del software a lo largo de todo su ciclo.

En este trabajo se propone el modelo reactivo, uno de los tres modelos propuestos por Krueger en 2002 para la creación de LPS. En este modelo no se establece desde un principio el dominio de aplicación de la LPS, sino que se irá ampliando, incluyendo nuevos productos a medida que se necesiten producirlos, creando los correspondientes activos. Este tipo de modelo permite una transición desde una ingeniería convencional, centrada en único producto, a una ingeniería basada en líneas de producto, centrada en la producción de variantes de productos. De esta manera se

propone un método para realizar una transición hacia la ingeniería basada en LPS.

El método propuesto se basa en dos estándares de la OMG: la Especificación de Activos Reutilizables (Reusable Asset Specification-RAS) [RAS] que permite identificar, describir y empaquetar activos de manera estándar, y el Metamodelo de Ingeniería de Procesos de Software (Software Process Engineering Metamodel-SPEM) [SPEM]. que define un lenguaje estándar para el modelado de procesos de software.

La correlación que existe entre las Líneas de Producto Software, la Ingeniería Dirigida por Modelos y la Ingeniería de Procesos Software, permite que las LPS puedan ser encapsuladas en activos reutilizables, hasta que se consiga automatizar completamente su desarrollo.

Para el desarrollo de este trabajo, se ha considerado la iniciativa del Model Driven Architecture (MDA) para la construcción de modelos de dominio (expresados como grafos de características o "features") y su posible transformación en modelos arquitectónicos. La novedad que propone MDA es la posibilidad de automatizar la transformación de las instancias del modelo de características en una aplicación ejecutable.

MDA ofrece un espectro más amplio de arquitecturas de desarrollo que las LPS, especificando facilidades de metamodelado y transformaciones independientes de plataforma [OMG, 2003]. SPEM puede ser utilizado naturalmente para definir actividades de transformación de modelos. BOM deriva, de forma automática, un producto específico en una línea de producto software a partir de

una serie de modelos previos. Los modelos son descritos en un lenguaje específico de dominio.

4.1.1 Herramientas que integra BOM

BOM, desarrollado e implementado "ex profeso" en esta tesis, más que una herramienta es un "framework" que captura en modelos conceptuales la información del diagnóstico y del dominio de aplicación específico, utilizando técnicas de MDA, para generar automáticamente sistemas expertos de diagnóstico, con arquitecturas PRISMA, basados en líneas de producto (paradigma FOP).

El uso de BOM (por el ingeniero del dominio) para construir los artefactos software necesarios para obtener un producto de la LPSD, integra el uso de dos herramientas desarrolladas "a priori" para otros ámbitos:

- RATIONAL ROSE.- para construir el modelo conceptual del dominio y los modelos conceptuales del dominio de aplicación.
- RAS.- para empaquetar los esqueletos y sus respectivos procesos de inserción de características en artefactos software denominados activos o *assets*, así como para crear la *Base-Line* conformada por dichos *assets* y los modelos conceptuales del dominio de aplicación.

El uso de BOM (por el ingeniero de la aplicación) para generar un producto de la LPSD, integra el uso de varias herramientas, las cuales han sido desarrolladas "a priori", para otros ámbitos:

- RATIONAL ROSE.- BOM permite, a través de una GUI, configurar los modelos conceptuales de dominio y los modelos conceptuales del dominio de aplicación.
- XAK.- BOM, a través de la herramienta XAK, permite crear los tipos PRISMA, a través de transformaciones XSLT (procesos de inserción de *features*) sobre documentos XML (esqueletos).
- RAS.- BOM permite seleccionar activos y modelos conceptuales del dominio de aplicación de la *Base-Line*.
- PRISMA-CASE.- BOM crea el "programa" de configuración, que es tomado por la herramienta PRISMA-CASE para configurar los modelos arquitectónicos PRISMA de los sistemas de diagnósticos, instanciando los tipos (PRISMA).
- PRISMA-MODEL-COMPILER.- BOM permite utilizar el compilador de modelos PRISMA-MODEL-COMPILER para generar automáticamente el código C# de la arquitectura software del paso anterior.
- PRISMA-NET.- el sistema final de diagnóstico lo ejecuta BOM sobre el middleware PRISMA-NET, i.e. una instancia de la LPSD.

4.2 Modelado del proceso de software para la creación de la LPSD

En este apartado se describe el modelado del proceso de creación de la línea de productos software de diagnóstico, reutilizando el mismo conjunto de recursos o activos principales (*core assets*) para crear una familia de productos software, basado en los

estándares de la OMG: la especificación de recursos reutilizables (Reusable Asset Specification-RAS) para definir de manera estándar toda la información asociada a un activo o recurso reutilizable, a través de la cual es posible manipularlo y reutilizarlo, y el metamodelo de ingeniería de procesos software (Software Process Engineering Metamodel-SPEM) para definir un lenguaje que modela procesos de desarrollo de software usando una terminología común y estándar.

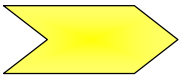
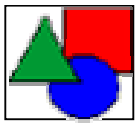
El desarrollo de este apartado, se encuentra en el marco de la propuesta de la OMG (Object Management Group), denominada arquitectura dirigida por modelos (Model Driven Architecture-MDA) que defiende el uso de estándares y potencia la independencia de plataforma en los procesos de desarrollo de software dirigidos por modelos (Model Driven Engineering-MDE), cuyo objetivo es el de promover el uso de los modelos como artefactos de primera clase a partir de los cuales generar código.

Con la finalidad de simplificar la descripción del modelado del proceso de software para la creación de la LPSD, a través del "Generador automático de sistemas de diagnóstico basado en líneas de producto", se presenta la figura X que muestra una metáfora visual de dicho generador, la cual es presentada en SPEM (por ser un estándar de la OMG para modelado de procesos de desarrollo de software) y fundamentada en la aproximación para desarrollar LPS mencionada por [Clements et al., 2002] al clasificar los procesos de la ingeniería en las LPS en dos partes: la ingeniería del dominio y la ingeniería de la aplicación.

Los procesos de la ingeniería en la LPSD serán mostrados en SPEM. El metamodelo de SPEM permite el modelado de muchos

aspectos y problemas del proceso de desarrollo. Aunque el metamodelo de SPEM es muy extenso, en este trabajo de tesis se ha centrado en el modelado de tareas, utilizando relaciones de secuencia pero sin prioridad entre las mismas. Las tareas, ejecutadas por roles, consumen artefactos de entrada y producen artefactos de salida. Una tarea puede tener asociada elementos guía que ayuden al rol a desempeñarla.

A continuación se presentan los iconos estandar de SPEM utilizados en esta tesis, así como su significado.

icono	***significado***
	actividad
	rol del proceso
	elemento guía
	proceso
	producto de trabajo

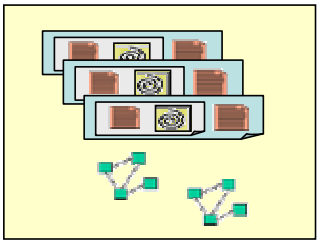
modelo UML
(especialización de producto de trabajo

documento
(especialización de producto de trabajo

Asimismo, se presentan los iconos SPEM utilizados *exprofeso* en esta tesis en los procesos de la ingeniería del dominio y de la ingeniería de la aplicación:

activo =
esqueleto + proceso de inserción de características

activo empaquetado =
activo +modelo RAS d el activo



Base-Line =
activos empaquetados + MCDAs

De esta manera, y con base en la aproximación anteriormente mencionada (ver apartado X.X de esta tesis), se describirán los procesos de la ingeniería en la LPSD, con las tareas que serán ejecutadas en cada una de las partes, de la siguiente manera:

1.- Ingeniería del dominio

i) análisis del dominio

tareas:

- crear modelo de características
- crear árbol de decisión
- crear modelo conceptual del dominio
- crear modelo conceptual del dominio de aplicación

ii) desarrollo de los componentes básicos reutilizables

tareas:

- crear esqueletos
- crear reglas de transformación y procesos de transformación para inserción de las características del dominio de aplicación específico
- crear activos
- crear modelos RAS de activos
- empaquetar activos
- crear la Base-Line
- crear procesos para elegir activos y modelos conceptuales del dominio de aplicación

iii) planeamiento de la producción

tareas:

- modelar plan de producción

2.- Ingeniería de la aplicación

i) caracterización del producto

tareas:

- configurar características del dominio
- configurar características del dominio de aplicación
- seleccionar activos
- crear tipos

ii) síntesis del producto

tareas:

- configurar arquitectura

iii) construcción del producto

tareas:

- compilar modelo (generar código)
- crear sistema ejecutable

De manera general se puede concluir que el ingeniero de dominio, es quien crea el plan de producción y los artefactos software que son utilizados para llevar a cabo sus tareas; y el ingeniero de aplicación, que es quien realiza el plan de producción de la LPSD.

Comparando las actividades que deberán realizar los analistas de BOM, en sus roles de ingeniero de dominio y/o ingeniero de aplicación, es notable que el trabajo del ingeniero de dominio es más complejo, al tener que diseñar y modelar con más detalle el proceso de desarrollo: modelos, recursos reutilizables, y los correspondientes guías de configuración adecuadas para generar cada uno de los productos de la línea. El ingeniero del dominio, especifica al ingeniero de aplicación como obtener una solución concreta a través de lenguajes específicos de domino.

En cambio el trabajo del ingeniero de aplicación se vuelve mucho más sencillo, trabajando a mayor nivel de abstracción sin preocuparse de las tareas de configuración de los componentes básicos reutilizables, dirigiendo sus esfuerzos en la especificación de las características deseadas para el producto a generar. El

ingeniero de la aplicación sólo debe de realizar la tarea de configuración; éste será el ejecutor de este proceso de desarrollo, que previamente había sido diseñado por un ingeniero de dominio. Con esta aproximación el ingeniero de la aplicación es obligado a seguir el plan predefinido por el ingeniero de dominio.

4.2.1 Ingeniería del Dominio

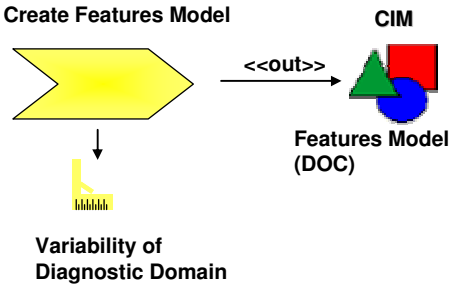
En esta primera fase crean los activos y se describe el proceso de configuración del activo como el plan de producción de la familia de productos. A continuación se muestran en SPEM todas las actividades realizadas por el analista que desempeña el rol de ingeniero del dominio, con el fin de ejecutar las distintas tareas con las que serán creados los artefactos software necesarios para realizar el plan de producción de la LPSD (en la ingeniería de la aplicación).

Todas estas tareas son realizadas en tres partes: análisis del dominio, desarrollo de los componentes básicos reutilizables y la planeación de la producción.

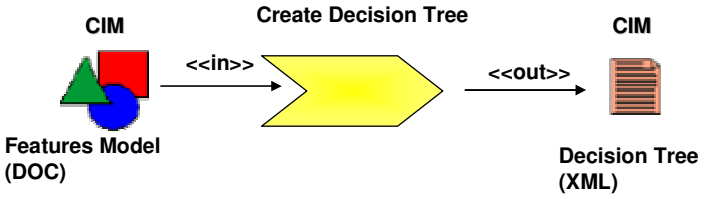
4.2.1.1. Análisis del dominio

En el análisis del dominio se estudia la variabilidad del dominio del diagnóstico, en términos de las propias características del dominio.

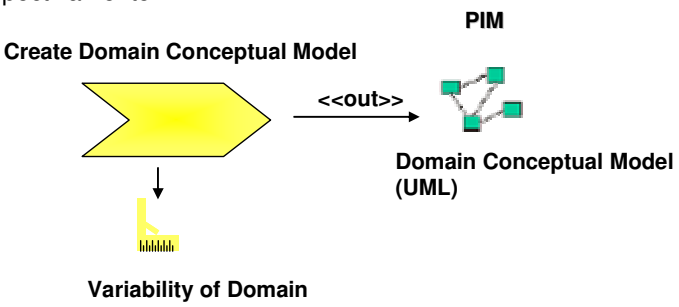
En esta etapa, el ingeniero del dominio, crea por un lado **el modelo de características (CIM)** que identifica las familias de la LPSD en términos de la variabilidad del dominio del diagnóstico.

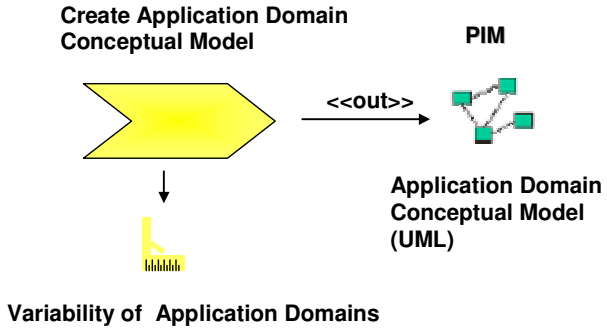


Las fuentes de variabilidad observadas en el modelo de características son mostradas en los puntos de variabilidad del **árbol de decisión (CIM)**, y cuyas hojas representarán las familias de los *assets* de la LPSD.



Asimismo crea **el modelo del dominio (PIM) y los modelos del dominio de aplicación (PIM)**, que captan la variabilidad del dominio del diagnóstico y de los dominios de aplicación; respectivamente.



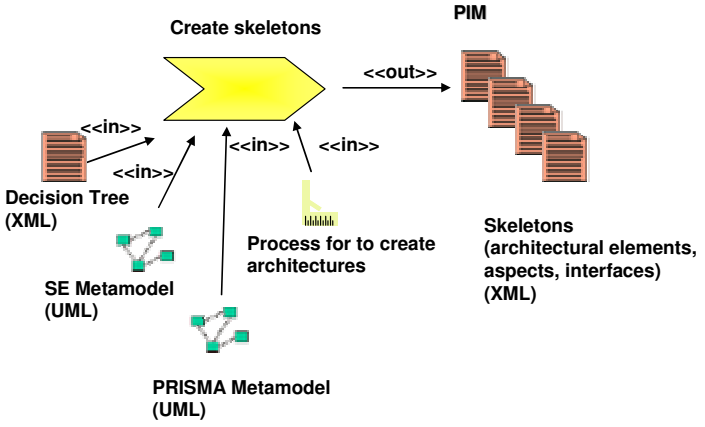


4.2.1.2 Desarrollo de los componentes básicos reutilizables

En el desarrollo de los componentes básicos reutilizables (*assets*), se concibe, diseña e implementan los componentes básicos reutilizables. Esto no sólo involucra el desarrollo de la funcionalidad del dominio, sino también define cómo los componentes básicos reutilizables deben ser extendidos.

En esta etapa, primeramente son creados todos los **esqueletos (PIM):** componentes, conectores, aspectos e interfaces (que formarán parte de los activos de la LPSD). Explícitamente, el ingeniero del dominio crea las plantillas software (ficheros configurados en XML utilizando el ADL de PRISMA) o esqueletos que conforman parte de la *Base-Line* de la LPSD. Para ello debe de tomar en cuenta la arquitectura básica de un sistema experto, con el fin de detectar los componentes que conformarán los sistemas de la LPSD. Debido a que este trabajo de investigación se encuentra en el marco de PRISMA, será necesario, basado en el metamodelo PRISMA, adicionar a esta arquitectura genérica, otro elemento arquitectónico adicional: el conector, así como los aspectos necesarios con sus interfaces. Para ello se

considerará, como elemento guía, la metodología BOM propuesta en esta tesis para la construcción de la estructura de los modelos arquitectónicos que conforman la LPSD. Asimismo es tomada en cuenta la información sobre la variabilidad de las características del dominio del diagnóstico a través del árbol de decisión.

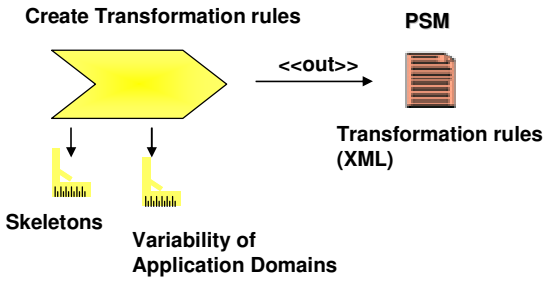


A continuación se muestra un ejemplo de esqueleto de aspecto (correspondiente al aspecto funcional de la Base de Conocimientos del diagnóstico médico) especificado en el LDA de PRISMA:

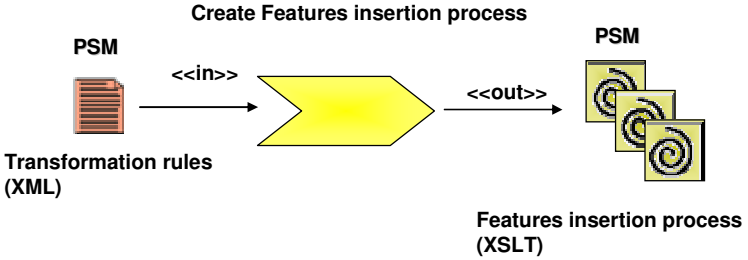
```
Functional Aspect <Functional Aspect Name> using
IDomainMD       Attributes Variable  (< Feature
FP.0.n >: bool )*;
  (< Feature FP.1.n >: bool )*;

 Derived       (< Feature FPH.n >: string )*;
  derivation
    (< Feature FR.1 >) *

  (< Feature FH.n >: string )*;
  derivation
    (< Feature FR.2 >)*;
    (< Feature FR.3 >)*;

Services  begin;
  in cleanDB ();   Valuations    [in cleanDB ()]
(< FP.0.n > := nil; )*;
    (< FP.1.n > := nil; )*;
    (< FPH.n > := nil; )*;
    (< FH.n > := nil; )*;
```

```
in inferPreHypothesis (input PROPERTY_0:list_bool,
  output PREHYPOTHESIS:string)
 Valuations
  [in inferPreHypothesis ()]
  (< FP.0.n >:= PROPERTY_0)*,
  PREHYPOTHESIS:= < FPH.n >;
in inferHypotheses (input PREHYPOTHESIS:string,
  output HYPOTHESES:list_string)
 Valuations
  [in  inferHypotheses ()]
  (< FPH.n >:= PREHYPOTHESIS,
  HYPOTHESES:= < FH.n >)*;  in  validateHypothesis
  (input PROPERTY_1:list_bool, output
  HYPOTHESIS:string)
  Valuations
  [in  validateHypothesis ()]
  (< FP.1.n >:= PROPERTY_1,
  HYPOTHESIS:= < FH.n>)* ;  end;

Played_Roles
 …
Protocols
 …
End_Functional Aspect <Functional Aspect Name>
```

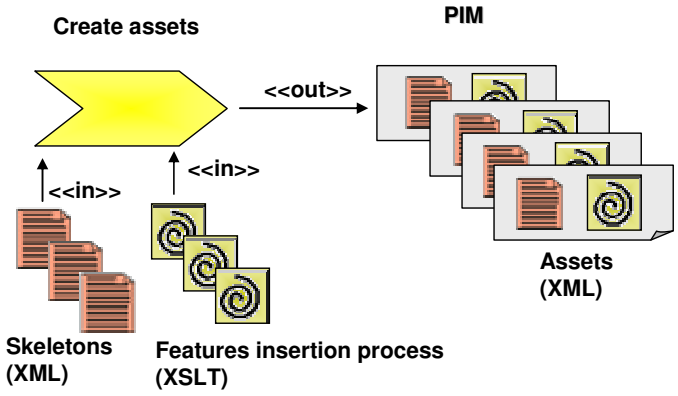
Estos esqueletos y la información de la variabilidad de los dominios de aplicación específicos, sirven de guía para la elaboración de las **reglas de transformación de inserción de las características del dominio de aplicación (PSM)**.



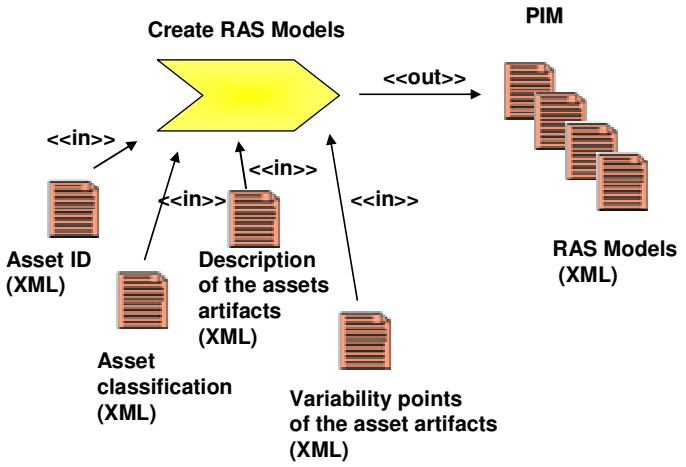
Dichas reglas son utilizadas junto con los esqueletos para crear los **procesos de transformación (PSM)** de dichas inserciones. Es decir, se crean los procesos de transformación para la inserción de las características del dominio de aplicación en los esqueletos.



El ingeniero del dominio asociará a cada esqueleto su correspondiente proceso de inserción de características Con dichos procesos y los esqueletos son creados los **activos (PIM)**, de tal forma que un activo estará conformado por un esqueleto y su correspondiente proceso de transformación de inserción de características.



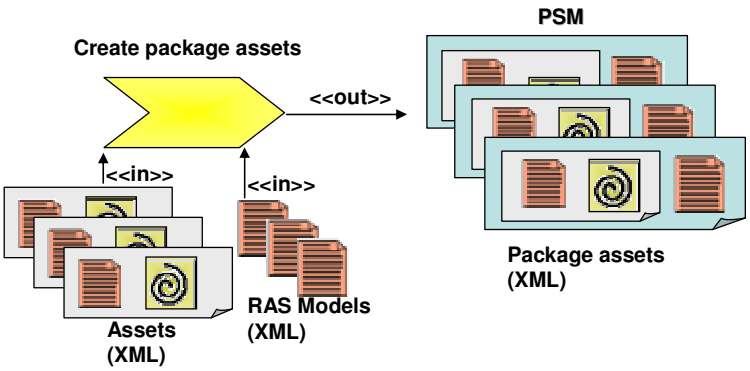
Además de los activos es conveniente crear los **modelos RAS (PIM)** (que conforma con el metamodelo RAS) de dichos activos, con el fin de obtener la información de cada uno de ellos. Esta actividad requiere de varios elementos guía: identificador del activo, clasificación del activo, descripción de los artefactos del activo, puntos de variabilidad de los artefactos del activo.



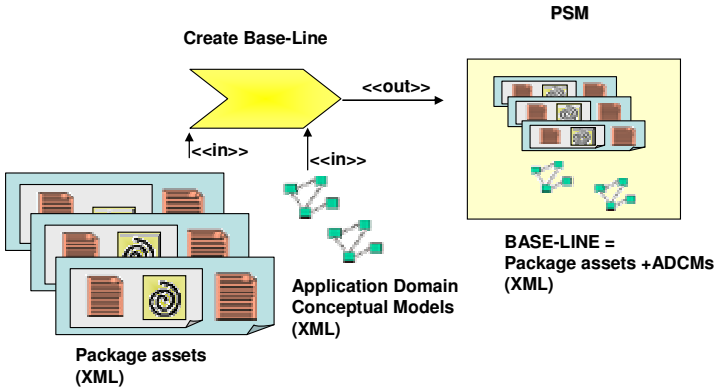
De esta forma, es posible crear artefactos software que contengan tanto a cada uno de los activos como su correspondiente información. Estos artefactos son denominados **activos empaquetados (PSM).** Por ello, para realizar la tarea de empaquetar el activo, es necesario contar como elementos guía al propio activo y su correspondiente modelo RAS.

Con esto, y de acuerdo con RAS**,** se define una manera estándar de empaquetar activos reutilizables, permitiendo identificar, describir y empaquetar un activo, cumpliendo los requisitos para una reutilización efectiva. Siguiendo la tecnología RAS, para el empaquetamiento de activos se ha utilizado el editor de modelos

RAS del *plugin* de Eclipse realizado por [Ávila-García et al, 2007] para la manipulación de activos RAS.



Con los modelos conceptuales del dominio de aplicación y los activos empaquetados, el ingeniero del dominio tiene la posibilidad de crear la ***Base-Line*** **(PSM)** que servirá como un repositorio de todos los assets.

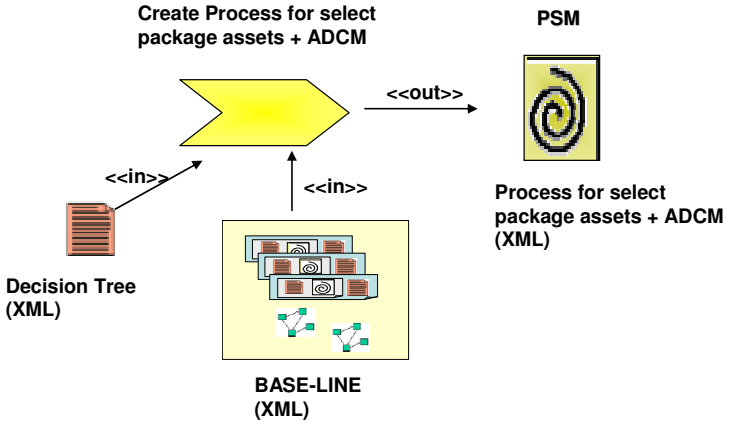


Cabe aclarar que el ingeniero del dominio no aborda la creación directa de la LPSD, sino la generación de familias de activos empaquetados. Las familias creadas serán organizadas en forma de LPS. De esta forma el ingeniero del dominio creará una línea de productos para generar la familia de activos empaquetados, creados para desempeñar esta tarea, realizando el siguiente procedimiento:

Cuando el ingeniero del dominio desee crear un activo, inspeccionará la *Base-Line* en busca de uno que cubra sus requerimientos. Si ninguno de los miembros de la familia satisface sus necesidades, deberá crear un activo por su cuenta, a partir del miembro que más se adecué a ellas. Cuando termine la tarea de creación, deberá realizar la ingeniería del dominio para incluir la nueva variante en la familia de productos y capturarla así a partir de entonces dentro de la *Base-Line*. A partir de este momento, el proceso de configuración del activo permitirá generar la nueva variante de activos, para ser aplicada siempre que se repitan las condiciones que originaron su aparición.

A medida que el ingeniero del dominio va encontrando similitudes entre los esqueletos que va creando en los diferentes dominios específicos, a través de sus puntos de variabilidad, formará familias de esqueletos, organizándolos en forma de LPS. Con ello, está en posibilidad de crear los **procesos de selección de *assets* (PSM)**, i.e. de los activos empaquetados y los modelos conceptuales de dominio de aplicación, utilizando la *Base-Line* y el árbol de decisión para llevar a cabo dicha tarea.

Es importante considerar que, la familia de productos que crea el ingeniero del dominio, deberá tener un buen conocimiento del diagnóstico, con el fin de seguir patrones y establecer similitudes entre las variantes de los activos que desea agrupar.

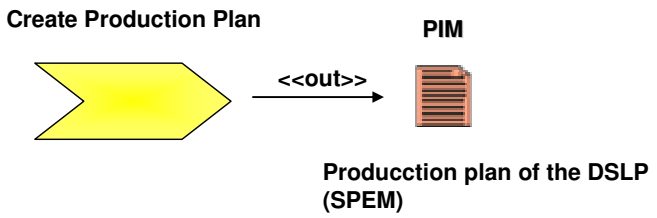


4.2.1.3 Planeamiento de la producción

El Planeamiento **de la producción (PIM)** implica la capacidad de fábrica de la LPSD. En esta etapa se define cómo los productos software individuales son creados. Para ello es creado el plan de producción de la familia de productos de la LPSD, que es descrita en el punto X.X de esta tesis, ya que la realización de dicha producción es parte de la ingeniería de la aplicación.

El ingeniero del dominio describirá el proceso de configuración del activo como el plan de producción de la LPS. La figura X muestra en SPEM el proceso a seguir en la ingeniería de la aplicación para el plan de producción de la LPS en el dominio del diagnóstico, i.e. LPSD.

Cabe señalar que este proceso de configuración del activo (que es el plan de producción de una línea de producto) es automático y obligatorio. Con esta situación, se obliga al ingeniero de la aplicación a utilizar las variantes ofrecidas por el activo reutilizable. De esta manera, el ingeniero de la aplicación no puede crear cualquier sistema de diagnóstico, sino que se le obliga a utilizar una de ellas, pudiendo seleccionar únicamente una de las variantes de la línea seleccionando las características que lo identifican. Con ello se obtienen dos ventajas: la primera es obligar a seguir patrones y buenas prácticas de modelado que el ingeniero del dominio realizando esta tarea ha ido estableciendo progresivamente mientras creaba el activo, y la segunda es centralizar el conocimiento para ayudar a nuevos ingenieros de la aplicación a aprender a realizar dicha tarea.



4.3.1 Ingeniería de la Aplicación

En esta segunda fase, las diversas tareas del proceso de desarrollo del software para el plan de producción de la LPSD son ejecutadas por el rol del ingeniero de aplicación. Es importante hacer notar que este plan de producción será el proceso de configuración del activo del diagnóstico específico que está siendo desarrollado.

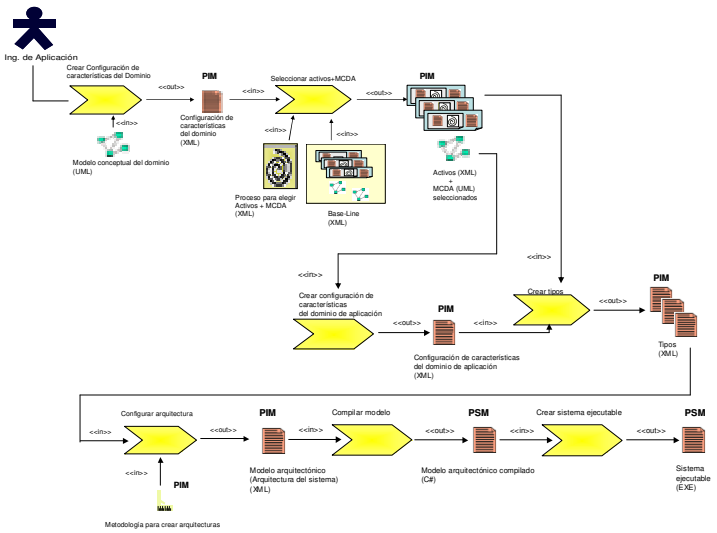


Figura 23. Plan de producción de la LPSD

Las LPS proponen la reutilización del mismo conjunto de activos principales para generar una línea de productos [Clements et al., 2002]. En esta estrategia es de vital importancia la capacidad de los activos de ser configurados por el ingeniero de aplicaciones. El proceso de configuración de activos para generar una aplicación concreta puede ser modelado a través de SPEM.

En la ingeniería de la aplicación las tareas son realizadas en tres partes: la caracterización del producto, la síntesis del producto y la construcción del producto.

4.3.1.1 Caracterización del producto

En la caracterización del producto se eligen las características que diferencian un producto seleccionado. Este proceso involucra tres

tareas ejecutadas a través del rol del ingeniero de aplicación: crear la configuración de las características del dominio, seleccionar los *assets*, configurar características del dominio de aplicación y crear los tipos.

Cabe señalar que el proceso de caracterización del producto, tiene como punto de partida un modelo conceptual de diagnóstico (i.e. dominio del diagnóstico) y un modelo conceptual del dominio específico (i.e. dominio de la aplicación). Se aplica una secuencia de transformaciones automáticas de dichos modelos a una arquitectura PRISMA para la LPSD. La transformación automática entre los modelos es esencialmente una transformación dirigida por la selección de características efectuada por el ingeniero de aplicación, configurando un modelo conceptual de características del diagnóstico y un modelo conceptual del dominio, que a su vez (por las relaciones de trazabilidad) generarán el modelo arquitectónico de la aplicación. Las variantes seleccionadas lo son en función de los requisitos particulares de la aplicación, que deben ser tomadas en cuenta por el ingeniero.

Tarea 1: Crear configuración de características del dominio
El ingeniero introduce la información del dominio del caso de estudio. BOM captura la variabilidad de la información a través de las características del dominio expresadas en el modelo conceptual del dominio. Este modelo permite al ingeniero (a través de la GUI) introducir la información de dicho modelo seleccionando las características del producto usando *check boxes* y *pull-down menus.*

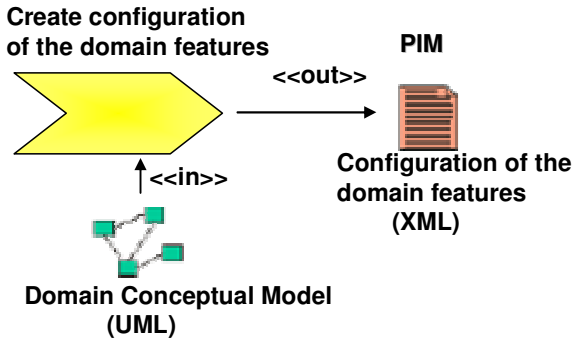


Figura 24. Proceso en SPEM para crear la configuración de características del dominio

Tarea 2 Seleccionar los activos y los modelos conceptuales del dominio de aplicación (MCDA)

En la tarea "seleccionar activos y MCDA" el ingeniero inserta la variabilidad de las características del dominio, con ello BOM selecciona los *assets* adecuados, i.e. los activos empaquetados y el modelo conceptual del dominio de aplicación de la *Base–Line*, por medio del proceso de selección de *assets* utilizando técnicas de árbol de decisión. En esta tarea de entrada se consume el modelo conceptual del dominio y los puntos de variabilidad del árbol de decisión para seleccionar los artefactos software; y de salida se producen los artefactos seleccionados para el dominio específico de la aplicación. Para llevar a cabo esta tarea, el ingeniero de aplicación utiliza la herramienta para proyectos RAS del *plugin* de Eclipse para la manipulación de activos RAS [Ávila-García et al, 2007].

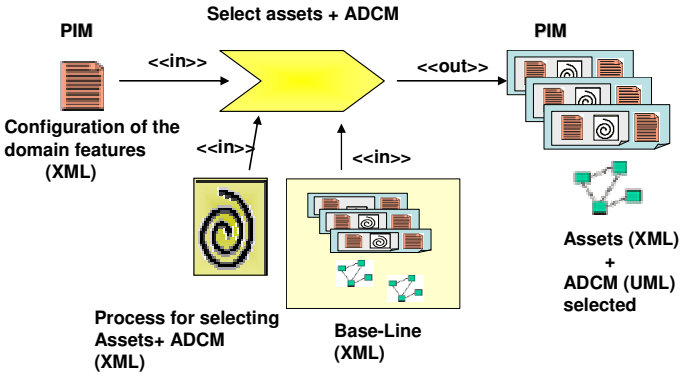


Figura 25. Proceso en SPEM para seleccionar activos+MCDA

Tarea 3 Crear configuración de características del dominio de aplicación

El ingeniero introduce la información del dominio de aplicación del caso de estudio. BOM captura la información de la variabilidad por medio de las características del dominio de aplicación contenidas en el modelo conceptual del dominio de aplicación correspondiente. Este modelo permite al ingeniero (por medio de la GUI) introducir la información presente en ese modelo usando *check boxes* y *pull-down menus*.

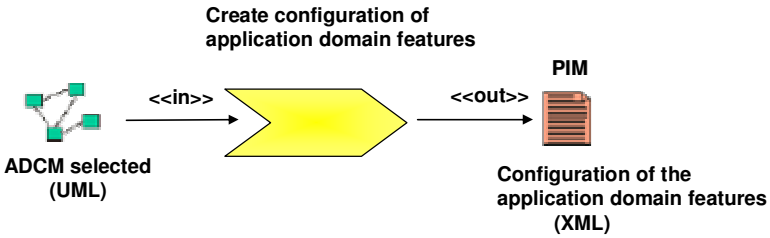


Figura 26. Proceso en SPEM para crear configuración de características del dominio de aplicación

Tarea 4 Crear tipos PRISMA

BOM rellena los esqueletos seleccionados con los datos de las características específicas del caso de estudio que fueron definidas por el ingeniero en la tarea anterior. De esta manera BOM crea los artefactos software de los tipos PRISMA. La transformación se realiza a través de plantillas XSLT aplicadas sobre documentos XML, que representan dichos artefactos software (especificados en el ADL de PRISMA). BOM utiliza la herramienta XAK [Trujillo, 2007] para llevar a cabo dicha transformación.

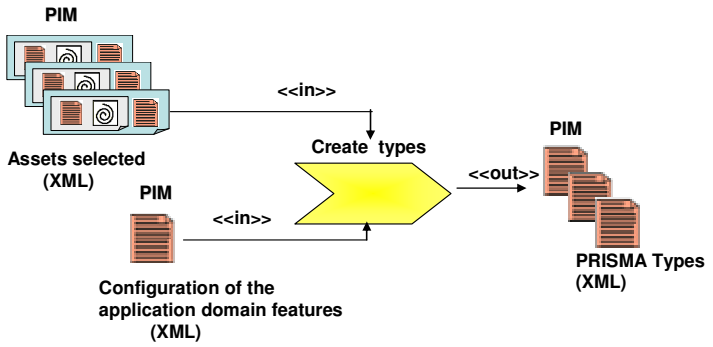


Figura 27. Proceso en SPEM para crear tipos

A continuación se muestra un ejemplo de tipo (el aspecto funcional de la Base de Conocimientos del diagnóstico médico) especificado en el LDA de PRISMA:

```
Functional Aspect  FBaseMD  using IDomainMD
Attributes Variable  cough: bool;
  fever: bool;
  respiratory_dificulty: bool;
  …
  dry cough: bool;
  constant fever: bool;
```

```
     grave respiratory_dificulty:bool;
     …
    Derived       syndrome: string;
     derivation
     (cough=true and fever=true and
     respiratory_dificulty=true) syndrome="warth"

     disease: string;
     derivation
     (constant_fever=true and greater_39_fever=true and
     2_3_days_fever=true and phlegmatic_cough=true and
     frequent_cough=true and grave_
     respiratory_dificulty) disease="pneumonia"
     …

   Services  begin;
     in cleanDB ();   Valuations    [in cleanDB ()]
   cough:= nil;
       fever:= nil;
       dry cough: nil;
       pneumonia: nil;
      …
     in inferPreHypothesis (input PROPERTY_0:list_bool,
       output PREHYPOTHESIS:string)
      Valuations
       [in inferPreHypothesis ()]
       (cough:=PROPERTY_0[0],
        fever:=PROPERTY_0[1],
        resp_dificulty:=PROPERTY_0[2],
        PREHYPOTHESIS:= warth;
       …    end;

   Played_Roles
    …
   Protocols
    …
End_Functional Aspect FBaseMD
```

4.3.1.2 Síntesis del producto

Un producto está compuesto por activos. La síntesis del producto reúne los activos para obtener la materia prima. En este proceso se

ejecuta la tarea "configurar arquitectura" a través del rol del ingeniero de aplicación.

Tarea 5 Configurar el modelo arquitectónico

BOM produce el programa de configuración que es usado por la herramienta PRISMA-CASE [Cabedo et al., 2005] con el fin de configurar la arquitectura software PRISMA al instanciar los tipos PRISMA. Dichas instancias configuran las arquitecturas software de la LPSD, i.e. la LPSD son los sistemas de diagnóstico de cada uno de los dominios específicos.

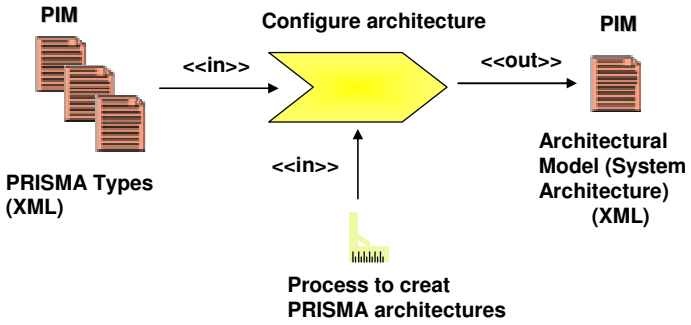


Figura 28. Proceso en SPEM para configurar la arquitectura

Un ejemplo de la metáfora visual de la herramienta PRISMA-CASE, es presentado en la figura 29. Este ejemplo corresponde al componente Base de Conocimientos del caso de estudio del diagnóstico médico.

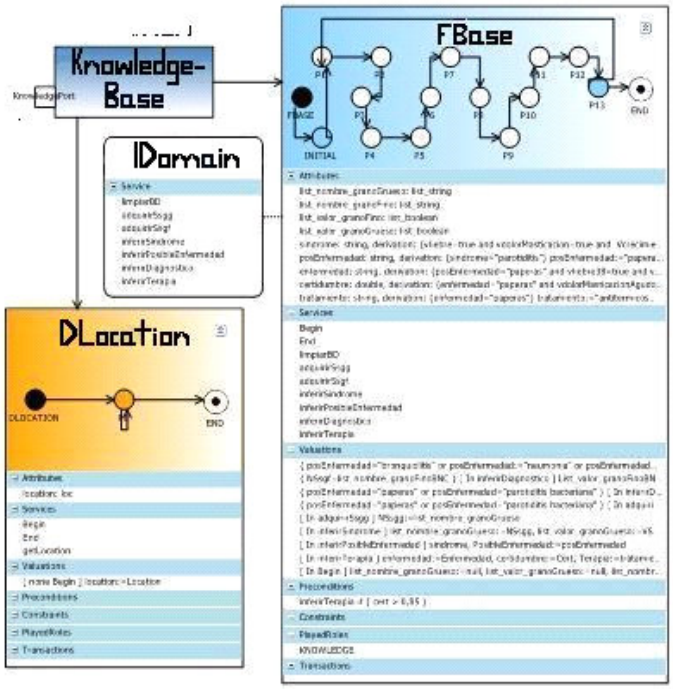


Figura 29. Metáfora visual de un elemento arquitectónico PRISMA sobre la herramienta PRISMA-CASE

4.3.1.3 Construcción del producto

La construcción del producto procesa la materia prima siguiendo el proceso de construcción (compilar, generar automáticamente código, ejecutar) para obtener el producto final.

Las tareas que se realizan en esta etapa a través del rol del ingeniero de aplicación son: compilar modelo y crear sistema ejecutable.

Tarea 6 Compilar modelo/Generar código

BOM usa la herramienta PRISMA-MODEL-COMPILER [Cabedo et al., 2005] para compilar la arquitectura software de la tarea anterior, generando automáticamente el código (en .NET, C#) de dicha arquitectura.

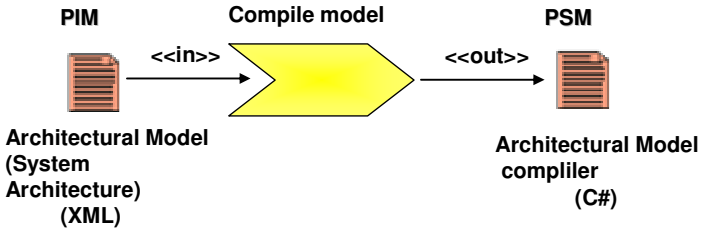


Figura 30. Proceso en SPEM para compilar el modelo arquitectónico

A continuación se muestra un ejemplo de código C# del componente Base de Conocimientos del caso de estudio del diagnóstico médico, generado por el Compilador de Modelos PRISMA-CASE.

```
namespace KBMD
{
    [Serializable]
    public class
    KnowledgeBaseMedicalDiagnosis:
    ComponentBase
   {
       public class
       KnowledgeBaseMedicalDiagnosis
       string name):  base (name)
      {
          AddAspect (new FBaseMD ( )
      );

     InPorts.Add

     ("KnowledgeClinicalPort",
     "IDomainMD", "KNOWLEDGE_CLIN");
     OutPorts.Add
     ("KnowledgeClinicalPort",
     "IDomainMD", "KNOWLEDGE_CLIN");
     InPorts.Add
```

```
        ("KnowledgeLaboratoryPort",
        "IDomainMD", "KNOWLEDGE_LAB");
         OutPorts.Add
        ("KnowledgeLaboratoryPort",
        "IDomainMD", "KNOWLEDGE_LAB");
         InPorts.Add
        ("KnowledgeResultsPort",
        "IDomainMD", "KNOWLEDGE_RES");
         OutPorts.Add
        ("KnowledgeResultsPort",
        "IDomainMD", "KNOWLEDGE_RES");
         }
      }
   }
```

Tarea 7 Crear sistema ejecutable

El sistema final de diagnóstico, i.e. una instancia de la LPSD, es ejecutada sobre el middleware PRISMA-NET [Costa et al., 2005].

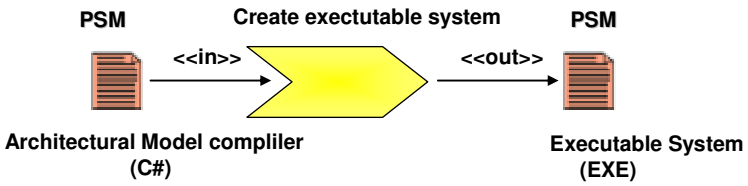


Figura 31. Proceso en SPEM para crear el sistema ejecutable

4.3.2 Ejecutar el sistema

Para clarificar la ejecución del sistema final de diagnóstico por el usuario final, se ha utilizado la notación en SPEM, aunque ésta no es parte del proceso del desarrollo del software.

Para realizar la tarea "ejecutar sistema", se consumen el sistema ejecutable correspondiente al modelo del sistema diagnóstico del caso específico (o el producto final de la LPSD), y los valores de las propiedades de la entidad a diagnosticar (introducidos a través de la GUI), produciendo como salida el resultado del diagnóstico de dicha entidad (en la GUI).

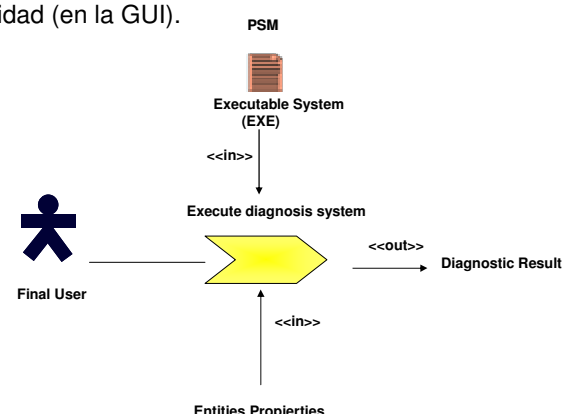


Figura 32. Ejecución de los sistemas de diagnóstico de la LPSD (con notación en SPEM)

El ejemplo de un caso de estudio del diagnóstico médico, es mostrado en la figura 33, en la que se expone la información de entrada al sistema por el usuario final, así como la salida del sistema que ofrece al usuario.

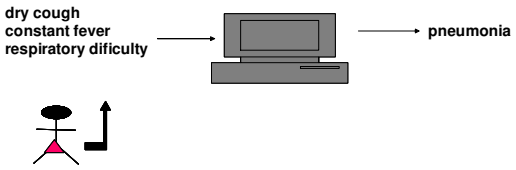


Figura 33. Información de entrada y salida de un sistema de diagnóstico desde el punto de vista del usuario final

Capítulo 5

CONCLUSIONES

En este capítulo se presentan las conclusiones más importantes derivadas de la investigación de esta tesis, así como algunas posibles ideas para realizar trabajos futuros con el fin de enriquecer BOM.

5.1 Conclusiones generales

Esta tesis ha propuesto una aproximación denominada BOM: un *framework* para generar automáticamente sistemas de diagnóstico basados en líneas de producto.

BOM ha sido diseñado para crear un software que permita la generación automática de sistemas de diagnóstico basado en líneas de producto, mejorando el desarrollo de los sistemas de diagnóstico de la siguiente manera:

- Usando las ventajas de los sistemas expertos, incorporando varias estrategias de razonamiento para resolver un problema aplicando la manera más eficiente, y separando los procesos de inferencia de la información del conocimiento de un dominio de aplicación.

- Construyendo sistemas de una manera simple usando ontologías del diagnóstico y de los dominios de aplicación. De esta manera, se ofrece un acercamiento al lenguaje específico del dominio del problema, facilitando la interacción con el usuario.

- Aplicando las técnicas de Líneas de Producto al construir un diseño que compartan todos los miembros de una familia de programas. De esta manera, un diseño específico puede ser usado en diferentes productos, reduciendo los costos, los tiempos de producción, el esfuerzo y la complejidad.

- Construyendo las arquitecturas de la línea de productos en el marco de PRISMA, obteniendo las ventajas de los sistemas distribuidos, facilitando su mantenimiento y complejidad.

- Creando una aproximación integrada y flexible para describir modelos arquitectónicos de diagnóstico, complejos, distribuidos y reutilizables, mejorando el desarrollo de los sistemas expertos que realizan tareas de diagnóstico, siguiendo el modelo PRISMA al integrar componentes y aspectos.

- Aplicando técnicas de MDA para implementar los sistemas en diferentes plataformas, y transformarlos automáticamente, incorporando las características de las instancias del Modelo de Características del Diagnóstico para obtener una aplicación ejecutable.

- Desarrollando sistemas independientes de plataforma, abordados desde la perspectiva del problema y no de la solución, lo cual provee generalidad en la aproximación desarrollada y aplicabilidad en diferentes dominios.

Con base en lo anteriormente descrito, se puede considerar que si es factible generar de forma automática sistemas que realizan tareas de diagnóstico basados en líneas de producto, utilizando técnicas de MDA y de las líneas de producto.

Relacionado y derivado de este trabajo de investigación, se han realizado las siguientes publicaciones:

- **Cabello Ma. Eugenia**, Alí Nour, Pérez Jennifer, Ramos Isidro y Carsí José Ángel. "DIAGMED: Un modelo arquitectónico para el DIAGnóstico MÉDico", IV Jornadas de Trabajo DYNAMICA, Archena, Murcia, España, nov. 2005.

- **Cabello Ma. Eugenia**, Ramos Isidro and Carsí José Ángel. "An architectural model for medical diagnosis". IADIS International Conference Applied Computing, San Sebastian, Spain, feb. 2006, pp. 700-701, ISBN 972-8924-09-7. (poster)

- **Cabello Ma. Eugenia**, Costa Cristóbal, Ramos Isidro and Carsí José Ángel. "Generic architecture of an expert system for diagnosis". In Proceedings of the 10 th WSEAS

International Conference on Computers, Athens, Greece, jul. 2006, ISSN 1790-5117.

- **Cabello Ma. Eugenia,** Ramos Isidro y Carsí José Ángel. "Uso de un modelo arquitectónico de componentes, aspectos y reflexión en sistemas de diagnóstico". Informe técnico DSIC-II/11/06, Universidad Politécnica de Valencia, España, jul. 2006.

- **Cabello Ma. Eugenia**, Costa Cristóbal, Ramos Isidro and Carsí José Ángel. "Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System". Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3, oct. 2006, pp. 1901-1908, ISSN 1790-0832, ISBN 960-8457-47-5.

- **Cabello Ma. Eugenia**, Costa Cristóbal y Ramos Isidro. "Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico". XIX Congreso Nacional y V Congreso Internacional de Informática y Computación, Chiapas, México, oct. 2006. ISBN 970-31-0751-6.

- **Cabello Ma. Eugenia** and Ramos Isidro. "A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines". INSTICC International Conference of Health Informatics HEALTHINF 2008, Madeira, Lisboa (pendiente de aceptación)

Sin embargo, al finalizar la implementación de BOM, se validará la propuesta y se realizarán publicaciones y comunicaciones de los resultados obtenidos de este trabajo.

5.2 Trabajos futuros

En el futuro, y por el hecho de haber establecido la aproximación reactiva sobre el uso de las líneas de producto software, se podrá extender el análisis del campo en otros dominios de aplicación con el fin de incrementar la variabilidad y la *Base-Line*. De esta manera, la Línea de Productos Software de Diagnóstico podrá ofrecer más productos.

Asimismo, es necesario integrar todas las herramientas que contempla BOM con el fin de crear un *framework* que ofrezca una funcionalidad conjunta y coordinada de todos los espacios tecnológicos contemplados.

Por otro lado, es necesario validar BOM en varios dominios específicos, tanto en el estudio de campo contemplado en esta tesis como en otros casos de estudio. De esta manera, la hipótesis planteada en esta tesis podrá ser corroborada con la validación de BOM, de acuerdo a la aproximación reactiva de la incorporación de la LPS realizada en este trabajo de tesis.

BIBLIOGRAFÍA

[Ali et al., 2005] Ali N., Ramos I. and Carsí J. A., *Conceptual Model for Distributed Aspect-Oriented Software Architectures.* International Conference on Information Technology Coding and Computing, ITCC 2005, IEEE Computer Society, Las Vegas, NV, USA, March 2005.

[Andrade et al., 1999] Andrade L. and Fiadeiro J., *Interconnecting Objects via Contracts*. OOPSLA´99.

[AOSD] Aspect Oriented Software Development, http://aosd.net

[Arango et al., 1991] Arango G. y Prieto-Díaz R., *Domain Analysis Concepts and Research Directions. Domain Analysis and Software Systems Modeling.* R. Prieto-Díaz y G. Arango editores. Páginas 9-32. IEEE Computer Society Press., 1991.

[ATL] http://www.sciences.univ-nantes.fr/lina/atl/

[ATLAS] ATLAS Group, LINA & INRIA, Nantes. *ATL User Manual. Version 0.7,* http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual[v0.7].pdf

[Ávila-García et al, 2006] Ávila-García, O., García A. E. Rebull V. S., y García J. L. R., *Integrando modelos de procesos y activos reutilizables en una herramienta MDA*. XI Jornadas de Ingeniería de Software y Bases de Datos JISBD'2006, Barcelona, España, Oct 2006, pp. 483-488.

[Bachman et al., 2000] Bachman F., Bass L., Chastek G., Donohoe P. and Peruzzi F., *The Architecture Based Design Method.* Technical Report CMU/SEI-2000-TR-001, Carnegie Mellon University, USA, January 2000.

[Bass et al., 1997] Bass L., Clements P., Cohen S., Northrop L. M., and Withey J., *Product Line Practice Workshop Report*. Technical Report CMU/SEI-97-TR-003 (ESC-TR-97-003), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1997.

[Bass et al., 1998a] Bass L., Chastek G., Clements P., Northrop L. M., Smith D., and Withey J., *Second Product Line Practice Workshop Report.* Technical Report CMU/SEI-98-TR-015 (ESC-TR-98-015), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1998.

[Bass et al., 1998b] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*. Addison-Wesley Longman, 1998.

[Batory et al., 2004] Batory D., Sarvela J.N., and Rauschmayer A., *Scaling Stepwise Refinament.* IEEE Transactions on Software Engineering (TSE), 30(6):3555-371, June 2004

[Batory et al., 2006] Batory D., Benavides D., and Ruiz-Cortés A., *Automated Analyses of Feature Models: Challenges Ahead*. CACM on Software Product Lines, 2006

[Batory, 2004] Batory D., *Feature-oriented programming and the AHEAD tool suite*. In Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, May 2004, pp. 702 – 703, ISSN: 0270-5257, ISBN: 0-7695-2163-0.

[Bernstein et al., 2000] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: *A Vision for Management of Complex Models*. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00).

[Boronat et al., 2004] Boronat, A., Pérez, J., Carsí, J. Á., and Ramos I., *Two experiencies in software dynamics*. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering, Vol. 10, (Issue 4), April 2004.

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

[Bosch, 2000b] Bosch J., *Product Line Architectures*. ObjectiveView, (4):13-18, http://www.ratio.co.uk

[Bracciali et al., 2002] Bracciali A., Brogi A. and Canal C., *Sistematic Component Adaptation*. In Proceedings of IDEAS, La Habana, Cuba, 2002.

[Brito et al., 2003] Brito I., and Moreira A., *Towards a Composition Process for Aspect-Oriented Requeriments*. Early Aspects 2003: Aspect_Oriented Requeriments Engineering and Architecture Design, Workshop of The 2nd. International Conference on Aspect_Oriented Software Development, Boston, USA, March 2003.

[Butler Group, 2003] Butler G., *Application Development Strategies. New Technologies and Methods for the Full Life Cicle*. 2003.

[Calad H, 2001] Calad H., *Common KADS-RT: Una metodologìa para el desarrollo de sistemas basados en el conocimiento de tiempo real*, septiembre 2001.

[Carsi, 1999] Carsí J A, *OASIS como Marco Conceptual para la Evolución de Software*. Tesis doctoral. Universidad Politécnica de Valencia, Valencia, España, 1999.

[Chastek et al., 2002] Chastek G. and McGregor J.D. *Guidelines for Developing a Product Line Production*

Plan. Technical Report, CMU/SEI, June 2002, CMU/SEI-2002-TR-06.

[Clauss et al., 2001] Clauss M., Muller U., and Franczyk B., *Modeling Variability with UML.* Young Researchers Workshop, GCSE, 2001.

[Clavel et al., 2002] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J.F.: *Maude: specification and programming in rewriting logic*. Theoretical Computer Science, 285(2):187-243, 2002.

[Clements et al., 1998] Clements P., Northrop L. M., Bachmann F., Bass L., Bergey J., Chastek G., Cohen S., Donohoe P., Jones L., Krut R., Little R., Smith D., Tilley S., Weiderman N., Withey J., and Woods S., *A Framework for Software Product Line Practice – Version 1.0. Product Line Systems Program.* Software Engineering. Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1998.

[Clements et al., 2002] Clements P. and Northrop L.M., *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison Wesley, 2002.

[Cohen et al., 1995] Cohen S. G., Friedman S., Martin L., Solderitsch N., and Webster R., *Product Line Identification for ESC-Hanscom.* Special Report CMU/SEI-95-SR-024, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1995.

[Cohen y Northrop, 1998] Cohen S. G., and Northrop L. M., *Object-Oriented Technology and Domain Analysis.* In Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 86-93., Victoria, B.C., Canada. IEEE-CS, 2 - 5 June 1998.

[Constantinides et al., 2000] Constantinides C.A., and Errad T., *On the Requeriments for Concurrent Software Architectures to Support Advanced Separation of Concerns*. In Proceedings of The OOPSLA 2000, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, 2000.

[CORPOICA, 2006] Librería virtual disponible en: http://www.corpoica.org.co/Libreria/software.asp?id_categoria=clase&id_producto=16&offsetr=1&index=1

[Costa et al., 2005] Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I. *PRISMANETMiddelweare: Soporte de la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos.* Actas de las X Jornadas de Ingeniería de Software y Bases de Datos, JISBD, Granada, España, 2005.

[Czarnecki et al., 2000] Czarnecki K., and Eisenecker U., *Generative Programming: Methods, Tools, and Applications*. Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[Czarnecki et al., 2005] Czarnecki K. and Antkiewicz M., *Mapping Features to Models: A Template Approach Based on Superimposed Variants.* In 4th International Conference on Generative Programming and

Component Engineering (GPCE 2005), Tallinn, Estonia, Sep.-Oct. 2005.

[Deelstra et al, 2003] Deelstra S., Sinnema M., Van Gurp J., and Bosch J., *Model Driven Architecture as Approach to Manage Variability in Software Product Families.* Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAAAFA 2003), pp. 109-114, CTIT Technical Report TR-CTIT-03-27, University of Twente, June 2003.

[EMF] http://www.eclipse.org/emf/

[EOL] http://www-users.cs.york.ac.uk/~dkolovos/epsilon/

[Feldon, 2004] Feldon S.E., *Computerized expert system for evaluation of automated visual fields from the ischemic optic neuropathy decompression trial: methods, baseline fields, and six-month longitudinal follow-up*. Trans Am Ophthalmol Soc. 2004;102:269-303.

[France y Bieman, 2001] France, R., and Bieman, J., *Multi-view Software Evolution: A UML based Framework for Evolving Object-Oriented Software*. Actas del International Conference on Software Maintenance (ICSM 2001), IEEE Computer Society, pp. 386-39.

[Fresnel et al, 1999] Fensel D., Benjamins R., and Motta E., *UPML: A Framework for Knowledge System Reuse*. In Proceeding of the International Joint Conference on AI (IJCAI-99), Stockholm, Sweden, 1999.

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

[Garlan et al., 2001] Garlan D., Cheng S. and Kompanek A. J. *Reconciling the Needs of Architectural Description with Object Modeling Notations*. Science of Computer Programming Journal, Special UML Edition, Elsevier Science, 2001.

[Giarratano et al., 2004] Giarratano, J., and Riley, G., *Expert Systems: Principles and Programming*. Fourth Edition: (Hardcover), ISBN: 0534384471, 842 pages, 2004

[Gomma et al., 2006] Gomaa H., Kerschberg L., Sugumaran V., Bosch C., Tavakoli I., and O'Hara L., *A Knowledge Based Software Engineering Environment for Reusable Software Requirements and Architectures*. Journal of Automated Software Engineering. Vol 3. Numbers 3-4, pages 285-307, August 2006.

[Gomma, 2004] Gomma, H., *Designing Software Products Line with UML: From uses cases to pattern-based software architectures.* The Addison-Wesley Object Thecnology Series. (Hardcover) 736 pages. 2004. ISBN-10: 0201775956, ISBN-13: 978-0201775952.

[González-Baixauli et al., 2005] González-Baixauli B. y Laguna M. A., *MDA e Ingeniería de Requisitos para Líneas de Producto*. Taller sobre Desarrollo Dirigido por Modelos. MDA y Aplicaciones. (DSDM´05), Granada,

España, 2005. pags10, http://ftp.informatik.rwthaachen.de/Publications/

[Greenfield et al., 2004] Greenfield J., Short K., Cook S, Kent S., and Crupi J., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[Griss et al., 1998] Griss M., Favaro J., and D´Allessandro M., *Integrating Feature Modeling with the RSEB.* In Proceedings of Fifth International Conference on Software Reuse (ICSR`5), Victoria, Canadá, June 1998, pp. 76-85.

[Herrero, 2003] Herrero ¿??, *Propuesta de una plataforma, lenguaje y diseño para el desarrollo de aplicaciones orientadas a aspectos.* Tesis Doctoral, Universidad de ¿???, Extremadura, España, 2003.

[Hickman et al., 1989] Hickman F., Killin J., and Land L., *Analysis for Knowledge-Based Systems: A Practical Guide to the KADS Methodology*. Ellis Horwood, England, 1989, pages 190.

[Iliffe et al., 2005] Iliffe S, Kharicha K, Harari D, Swift C, and Stuck AE., *Health risk appraisal for older people in general practice using an expert system: a pilot study.* Health Soc Care Community. 2005 Jan;13(1):21-9.

[Jacobson et al., 1997] Jacobson I., Griss M., and Jonsson P., *Software Reuse. Architecture, Process and Organization for Business Success*. ACM Press. Addison Wesley Longman, 1997.

[Jansson, 2005] Jansson ET. *Alzheimer disease is substantially preventable in the United States -- review of risk factors, therapy, and the prospects for an expert software system*. Med Hypotheses. 2005;64(5):960-7

[Jouault et al., 2005] Jouault F., and Kurtev I., *Transforming Models with ATL.* Actas del Model Transformations in Practice. Workshop realizado en conjunción con MODELS 2005, Jamaica, 2005.

[Kang et al., 1990] Kang K. C., Cohen S. G., Hess J. A., Novak W. E., and Peterson A. S., *Feature-Oriented Domain Analysis (FODA). Feasibility Study*. Technical Report CMU/SEI-90-TR21 (ESD-90-TR-222), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, 1990.

[Kang et al., 1998a] Kang K., Kim S., Lee J., Kim K., and Shin E., *FORM: A Feature Oriented Reuse Method with Domain Specific Reference Architectures*. Annals of Software Engineering, Vol. 5, 1998, pp. 143-168.

[Kang et al., 1998b] Kang K. C., Kim S., Lee J., Kim K., and Shim E., *Feature-Oriented Software Engineering for the Electronic Bulletin Board System (EBBS) Domain*. In Proceedings of the Third World Conference on Integrated Design and Process Technology (IDPT'98), 1998.

[Karlsson et al., 2004] Karlsson D, and Forsum U., *Medical decision-support systems and the concept of context.* Med Inform Internet Med. 2004 Jun;29(2):109-18.

[Kiczales et al., 1997] Kiczales G. et al., *Aspect Oriented Programming (ECOOP`97),* LNCS 124, Springer Verlag, 1997.

[Kiczales et al., 2001] Kiczales G., Hilsdale E., Huguin J., Kersten M., Plam J. and Griswold W., *An Overview of AspectJ.* Proceeding of the European Conference on Object-Oriented Programming, Springer Verlag, 2001.

[Klingler et al., 1996] Klingler C. D., and Solderitsch J., *DAGAR: A Process for Domain Architecture Definition and Asset Implementation.* In Proceedings of the Annual International Conference on ADA (TriAda'96), pages 231-245, December 3-7, 1996, Philadelphia, PA (USA). ACM, ACM Press., 1996.

[Knauber et al., 2001] Knauber P., and Succi G., *Perspectives on Software Product Lines*. ACM Software Engineering Notes, 26(2):29-33. March, 2001.

[Kopec et al., 2004] Kopec D, Shagas G, Selman J, Reinharth D, and Tamang S., *Development of an expert system for differentiating tension type headaches from migraines*. Study Health Technology Inform. 2004;103:81-92.

[Kotze et al., 2004] Kotze B, and Brdaroska B., *Clinical decision support systems in psychiatry in the information age*. Australas Psychiatry, 2004 Dec;12(4):361-4.

[Krueger, 1992] Krueger C.W., *Software Reuse*. ACM Computing Surveys, 24(2):131-183. 1992.

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

[Kurtev, 2005] Kurtev I., Adaptability of Model Transformations. PhD. Thesis, University of Twente, The Netherlands, 2005.

[Lee et al., 2000] Lee K., Kang K. C., Chae W., and Choi B. W., *Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse*. Software: Practice and Experience, 30(9):1025-1046, 2000.

[Liao, 2003] Liao S.-H., *Knowledge Management Technologies and Applications- Literature Review from 1995-2002.* In Expert Systems with Applications, Vol. 25, Issue 2, 2003, pp. 155-164.

[Liao, 2005] Liao S.-H., *Expert Systems Methodologies and Applications- a Decade Review from 1995-2004*. In Expert Systems with Applications, Vol. 28, Issue 1, 2005, pp. 93-103.

[Liebewitz, 1998] Liebewitz J., *The Handbook of Applied Expert Systems.* CRC Press, 1998.

[Linton et al., 1989] Linton M. A., Vlissides J. M., and Calder, P. R., *Composing User Interfaces with Interviews*. IEEE Computer, 22(2), 1989.

[Little, 1970] Little J.D.C., *Models and Managers: The Conceptual of a Decision Calculus.* Management Science, Vol. 16, No. 8, April 1970, pp. 466-485.

[López-Herrejón, 2005]. López-Herrejón R.E., *Understanding Feature Modularity in Feature Oriented Programming, 2005,* http://www.ecoop.org/phdoos/ecoop2005phd/RobertoErickLopezHerrejon.pdf

[Loques et al., 2000] Loques O., Sztajnberg A., Leite J., and Lobosco M., *On the Integration of Meta-level Programming and Configuration Programming*. In Reflextion and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato V. 1826, Lectures Notes in Computer Science, pp. 191-210, Springer-Verlag, Heidelberg, Germany, 2000.

[Marten et al., 2005] Marten K, Grillhosl A, Seyfarth T, Obenauer S, Rummeny EJ, aand Engelke C., *Computer-assisted detection of pulmonary nodules: evaluation of diagnostic performance using an expert knowledge-based detection system with variable reconstruction slice thickness settings*. Eur Radiol. 2005 Feb;15(2):203-12. Epub 2004 Dec 2.

[McGurren et al., 2002] McGurren F., and Conroy D., *X-ADAPT: An Architecture for Dynamic Systems, Workshop on Component Oriented Programming*. ECOOP, Málaga, Spain, 2002.

[McIlroy, 1976] McIlroy M. D., *Mass-Produced Software Components. Software Engineering Concepts and Techniques;* 1968 NATO Conference on Software

Engineering. J. M. Buxton, P. Naur y B. Randell editores. Páginas 88-98. Van Nostrand Reinhold, 1976.

[MDA Guide 2003] *MDA Guide Version 1.0.1. Documento OMG 2003-06-01, 2003,* http://www.omg.org/docs/omg/03-06-01.pdf

[MDA] Model Driven Achitecture, http://www.omg.org/mda

[Medvidovic et al., 2002] Medvidovic N., and Taylor R.N., *A Classification and Comparison Framework for Software Architectures.* In Proceedings of IDEAS, La Habana, Cuba, 2002.

[Melnik et al., 2003] Melnik, S., Rahm E., and Bernstein P. A., *Rondo: A Programming Platform for Generic Model Management*. SIGMOD 2003, Extended ver. in Web Semantics, vol. 1, Num. 1.

[Mens et al., 2005] Mens T., Van Gorp P., *A Taxonomy of Model Transformation.* Proceedings of Workshop on Graph and Model Transformation (GraMoT ` 2005), 2005.

[Meyer et al., 1997] Meyer M., and Lehnerd A., *The Power of Product Platforms*. Free Press, 1997].

[Mili et al., 1995] Mili H., Mili F. and Mili A., *Reusing Software: Issues and Research Directions.* IEEE Transaactions on Software Engineeering, 21(6):528-562, 1995.

[Miller et al., 2001] Miller, J., and Mukerji, J., *Model Driven Architecture*. Document number ormsc/2001-07-01, http://www.omg.org/mda.

[ModelWare Project, 2006] SmartQVT: An Open Source Model Transformation Tool Implementing the MOF 2.0 QVT-Operational Language, http://smartqvt.elibel.tm.fr.

[MOF] MOF QVT Standard Specification, http://www.omg.org/docs/ptc/05-11-01.pdf.

[MOMENT] http://moment.dsic.upv.es

[Monge et al., 2002] Monge R., Alves C. and Vallecillo A., *A Graphical Representation of COTS-Based Software Architecture*. In Proceedings of IDEAS, La Habana, Cuba, 2002.

[MYCIN] http://www.fortunecity.com/skyscraper/chaos/279/docs/mycin.htm

[Myllymaki, 2001] Myllymaki T., *Variability Management in Software Product-Lines*. Software Systems Laboratory, Tampere University of Technology, 2001.

[Neighbors, 1984] Neighbors J. M., *The Draco Approach to Constructing Software from Reusable Components*. IEEE Transactions on Software Engineering, SE-10(5):564-574, 1984.

[.NET] Microsoft.NET, http://www.microsoft.com/net.

[Noriega et al., 1998] Noriega P. y Sierra C., *Subastas y Sistemas Multiagente*. Revista Iberoamericana de Inteligencia Artificial, Número 6, pp. 68-84, 1998.

[Northrop, 2002] Northrop.L.M., *SET Software Product Line Tenets*. IEEE Software, 19(04):32-40, 2002.

[Oliva et al., 1998] Oliva A., Garcia I.C., and Buzato L.E., *The Reflective Architecture of Guaraná.* Technical Report IC-98-14, Instituto de Computación, Universidad de Campiñas, 1998.

[OMG] The Object Management Group, http://www.omg.org.

[OMG, 2005] Object Management Group and QVT-Merge Group, Revised submission for MOF 2.0 Query/View/Transformations version 2.0. Object Management Group doc.ad/2005-03-02, 2005.

[Pérez et al., 2002]. Pérez J., Ramos I., Lorenzo A., Letelier P. and Jaén J., *PRISMA: Plataforma OASIS para Modelos Arquitectónicos.* Actas de las VII Jornadas de Ingeniería de Software y Bases de Datos, JISBD, Escorial (Madrid), España, 2002.

[Pérez et al., 2005] Pérez J., Ali N., Carsí J., Ramos I. y Navarro E., *Designing Software Architectures with an Aspect-Oriented Language.* Journal on Aspect Orientation, Published by Houman Younessi, RISE ®ADVICE 2004 ISSN 1548-3851, Vol. 1.1. pags. 20, 2005

[Pérez, 2003] Pérez J., *OASIS como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuídos y Evolutivos*. Trabajo de investigación dentro del programa de doctorado de Programación Declarativa e Ingeniería de la Programación, Universidad Poltécnica de Valencia, España, 2003.

[Pérez, 2006] Pérez J., *PRISMA: Aspect-Oriented Software Architectures*. PhD. Thesis of Philosophy in Computer Science, Polytechnic University of Valencia, Spain, Dec. 2006, pages 397.

[Pohl et al., 2006] Pohl K., Bockle G., and Van der Linden F., *Software Product Line Engineering-Foundations, Principles and Techniques*. Springer, 2006.

[Pree, 1994] Pree W., *Meta Patterns – A Means for Capturing the Essential of Reusable Object-Oriented Design*. In Proceedings of the 8thEuropean Conference on Object-Oriented Programming. Bologna, Italy, 1994.

[Pree, 1995] Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[Prochaska et al., 2005] Prochaska J.O., Velicer W.F., Redding C., Rossi J.S., Goldstein M., DePue J., Greene G.W., Rossi S.R., Sun X., Fava J.L., Laforge R., Rakowski W., and Plummer B.A., *Stage-based expert systems to guide a population of primary care patients to quit smoking, eat healthier, prevent skin cancer, and receive regular mammograms.* Prev Med. 2005 Aug; 41(2):406-16.

[Queralt et al, 2006] Queralt P., Hoyos L.,Boronat A., Carsí J.A, and Ramos I., *Un motor de transformación de modelos con soporte para el lenguaje QVT Relations.* Taller Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto con JISBD'06), Sitges, España, Octubre 2006.

[Rashid et al., 2002] Rashid A., Sawyer P., Moreira A., Araujo J. and Esarly ¿?, *Aspects: a Model for Aspect-Oriented Requeriments Engineering*. IEEE Joint Conference on Requeriments Engineering, Essen, IEEE Computer Society, pp. 199-202, Germany, 2002.

[Reich et al., 2005] Reich Y. and Kapeliok., Decision Support Systems. Vol. 41, Issue 1, p.1-19, Nov. 2005.

[Riebisch et al., 2003] Riebisch M., Streitferdt D., and Pshov I., *Modeling Variability for Object-Oriented Product Lines.* In Proceedings of the Workshop Modeling Variability for Object-Oriented Product Lines, ECOOP, 2003.

[Roddick et al., 2001] Roddick J.F., Flue M., and Graco W.J., *Exploratory Medical Knowledge Discovery*, Artificial Intelligence in Medicine, July 2001, pp. 89-109.

[Sancipriano, 2005] Sancipriano GP. *Artificial intelligence--the knowledge base applied to nephrology.* G Ital Nefrol, 2005, Jan-Feb; 22(1):47-62.

[Schurink et al., 2005] Schurink C.A., Lucas P.J., Hoepelman I.M., and Bonten M.J., *Computer-assisted decision support for the diagnosis and treatment of infectious*

diseases in intensive care. Lancet Infect Dis., 2005, May; 5(5):305-12, units.

[Sendall et al., 2003] Sendall S. and Kozaczynski W., *Model Transformation – the Heart and Soul of Model-Driven Software Development*. IEEE Software, Special Issue on Model Driven Software Development, pages 42--45, Sept/Oct 2003.

[Shi, 2004] Shi H, Paolucci U., Vigneau-Callahan K.E., Milbury P.E., Matson W.R., Kristal B.S., *Development of biomarkers based on diet-dependent metabolic stereotypes: practical issues in development of expert system-based classification models in metabolomic studies.* OMICS. 2004 Fall; 8(3):197-208.

[Simos et al., 1996] Simos M., Creps D., Klingler C., Levine L., and Allemang D., *Organization Domain Modeling (ODM) Guidebook – Version 2.0.* Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, 9255 Wellington Road Manassas, VA 22110-4121, 1996.

[Simos, 1995] Simos M., *Organization Domain Modeling (ODM): Formalizating the Core Domain Modeling Life Cycle*. Symposium on Software Reusability. April, 1995.

[Sonnemann, 1995] Sonnemann R., *Exploratory Study of Software Reuse Factory*. PhD. thesis, George Mason University, Fairfax, Virginia, Spring 1995.

[SPEM] Software Process Engineering Metamodel, Version 2.0. Technical Report, 2006. http://www.omg.org/docs/ad/06-06-02.pdf

[Studer et al., 2000] Studer R., Decker S., and Fensel D., *Situation and Perspective of Knowledge Engineering*. In J. Cuenca et al (eds.), Knowledge Engineering and Agent Technology, IOS Press, 2000, 16 p.

[Suvée et al., 2003] Suvée D., Vanderperren W., Jonckers V., *JAsCo: An Aspect-Oriented Aprroach Tailored for Component-Based Software Development.* 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 21-29, ISBN: 1-58113-660-9. Boston, Massachusetts, USA, March 2003.

[Svahnherg et al., 2000a] Svahnherg M., Van Gurp J., and Bosch J., *On the Notion of Variability in Software Product Lines.* Blekinge Institute of Technology Research Report 2000:2, ISSN: 1103-1581.

[Svahnherg et al., 2000b] Svahnherg M., and Bosch J., *Issues Conserning Variabiliaty in Software Product Lines*. *In Development and Evolution of Software Architectures for Product Familie.* Proceedings of International Workshop IW-SAPF3, Vol. 1409 of Lecture Notes in Computer Science, Springer, March 2000, pp. 146-157

[Szyperski, 1998] Szyperski C., *Component Software: beyond Object-Oriented Programming*. ACM Press and Adisson Wesley, New York, USA, 1998.

[Taenatzer et al., 2000] Taenatzer G., Nagl M., Schurr A., and Munch M., *AGG: A Tool Environment for Algebraic Graph Transformation*. *Applications of Graph Transformations with Industrial Relevance*. Ed. Springer Verlag, LNCS 1779, pp. 481-488.

[Tickner et al., 2005] Tickner J., Friar J., Creely K.S., Cherrie J.W., Pryde D.E., and Kingston J., *The development of the EASE model.* Ann Occup Hyg. 2005 Mar; 49(2):103-10.

[Trujillo, 2007] Trujillo S., *Feature Oriented Model Driven Product Lines.* PhD. Thesis, The University of the Basque Country, San Sebastian, Spain, March 2007, pages.175.

[UML] Unified Modeling Language version 2.0, http://www.uml.org

[Van Hoof et al., 2004] Van Hoof V., Wormek A., Schleutermann S., Schumacher T., Lothaire O., and Trendelenburg C. *Medical expert systems developed in j.MD, a Java based expert system shell: application in clinical laboratories*. Medinfo. 2004; 11(Pt 1):89-93.

[Vici et al., 1998] Vici A. D., and Argentieri N., *FODAcom: An Experience with Domain Analysis in Italian Telecom Industry.* In Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 166-175. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS, 1998.

[Warmer et al., 2004] Warmer, J., and Kleppe, A., *The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, 2004.

[Wirfs-Brock et al., 1990] Wirfs-Brock R. J., and Johnson R.E., *Surveying current research in object-oriented design.* Communications of the ACM, 33(9):105-124, 1990.

[Yeung, 2004] Yeung D.S., Wang X.Z., and Tsang E.C., *Handling interaction in fuzzy production rule reasoning.* IEEE Trans Syst Man Cybern B Cybern. Oct. 2004; 34(5):1979-87
CEUR-WS/Vol-157/paper02.pdf.

Apéndice A:

TERMINOLOGÍA DEL DIAGNÓSTICO

En este apéndice se presentan brevemente los conceptos que son utilizados para el modelado de un sistema basado en el conocimiento desde la aproximación de líneas de producto software orientada al diagnóstico.

Dichos conceptos especifican la ontología del diagnóstico, involucrando características y particularidades del diagnóstico en el que se desenvolverá el sistema que se desea construir.

Conceptos relacionados con la variabilidad en la LPSD

Con el fin de clarificar la terminología utilizada en la variabilidad de la LPSD, a continuación se definen y/o describen los conceptos manejados en esta tesis para abordar el tema de la variabilidad.

a) Diagnóstico: El diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a través de sus propiedades (variables observables). Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una anomalía o propiedad a partir de datos observables y razonamientos.

b) Proceso de diagnóstico: Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una cuestión a partir de datos observables y razonamientos.

c) Coreografía del proceso de diagnóstico: La coreografía del proceso de diagnóstico establece la sincronización entre el mecanismo de inferencia, la información del dominio y la interfaz del usuario.

d) Entidad: Una entidad es a quien (o a lo que) se le realiza el diagnóstico, con el fin de identificar una cuestión provocado por causas internas y/o externas.

El modelo de características del diagnóstico muestra la variabilidad de la LPSD.

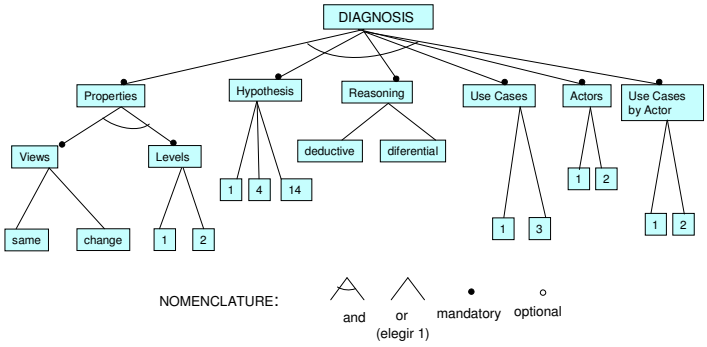


Figura 34. Modelo de características del diagnóstico

Las fuentes de variabilidad (características de la LPSD) observadas en el modelo de características del diagnóstico, son mostradas a través del árbol de decisión.

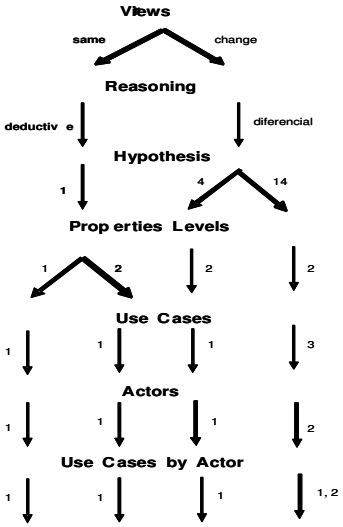


Figura 35. Àrbol de decisión

El modelo conceptual del dominio del diagnóstico especifica la ontología del diagnóstico, involucrando características y particularidades del diagnóstico en el que se desenvolverá el sistema que se desea construir.

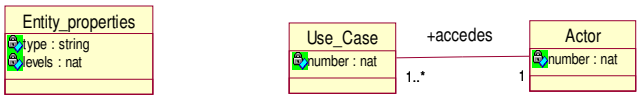


Figura 36. Modelo conceptual del dominio de diagnóstico

Cada una de las características de la LPSD involucradas en el modelo de características, el árbol de decisión y el modelo conceptual del dominio, son los necesarios para el modelado de un sistema basado en el conocimiento desde una aproximación orientada al diagnóstico. Los conceptos que se corresponden con la perspectiva CIM del diagnóstico y describen elementos propios del diagnóstico, son comentadas a continuación:

e) Hipótesis: Una hipótesis es el objetivo o resultado del proceso de diagnóstico. Cuando se identifica el problema o disfunción que presenta una entidad, se obtiene una hipótesis validada. El diagnóstico realizado a una entidad puede llegar a tener una sola hipótesis o bien varias hipótesis.

f) Número de hipótesis: Cuando las propiedades de una entidad no cambian, se genera una sola hipótesis. Sin embargo cuando las propiedades de una entidad si cambian, se generan varias hipótesis, que deberán ser validadas para llegar a un resultado del diagnóstico.

g) Propiedades: Las propiedades de una entidad son la caracterización mediante datos de las causas que provocan una anomalía o disfunción de la misma. Una entidad puede estar caracterizada siempre por las mismas propiedades o bien pueden variar esas propiedades, i.e. la entidad está caracterizada por un conjunto específico de propiedades que varían según la hipótesis a validar. Por ello las propiedades se clasifican en dos tipos: propiedades iguales y propiedades distintas.

h) Nivel de las propiedades: El proceso de diagnóstico contempla varios niveles de abstracción de las propiedades de una entidad, y su relación entre las propiedades a través de reglas, de forma que:

- Las propiedades del nivel n y las propiedades del nivel n+1, se relacionan a través de las reglas de nivel n+1,
- Las propiedades del nivel 0 obtienen su valor del usuario,
- Las propiedades del nivel n+1 se infieren aplicando las reglas del nivel n+1,
- El valor de las propiedades superiores al nivel 0, se puede obtener del usuario o bien inferirse a través de las reglas.

i) Vistas (de una entidad): Una entidad puede estar caracterizada siempre por las mismas propiedades (misma vista) o bien pueden variar esas propiedades (diferentes vistas), i.e. la entidad está caracterizada por un conjunto específico de propiedades para cada hipótesis. Por ello las propiedades se clasifican en dos vistas: mismas propiedades y propiedades que cambian.

j) Reglas: Las propiedades de las entidades al relacionarse entre sí pueden cumplir ciertas reglas. Una relación entre propiedades y/o subpropiedades es una regla del tipo IF <premisa> THEN <conclusiòn>, i.e. una cláusula de Horn con cabeza, donde la premisa y la conclusión de dichas reglas son valores de propiedades y/o subpropiedades.

La Base de Conocimientos contiene este tipo de reglas que representan la información del dominio de aplicación del diagnóstico y se representan mediante una estructura de árbol.

El Motor de Inferencia al aplicar las estrategias de razonamiento, recorre el árbol, i.e. aplica las reglas.

Una estructura de árbol, como la generada por el encadenamiento de reglas, puede ser recorrida en dos sentidos:

- en anchura (niveles) , donde se contemplan todas las posibilidades de un nudo antes de pasar al siguiente nodo o bifurcación del árbol.
- en profundidad (ramas).- no se toma otra posibilidad en un nodo hasta que no se ha desarrollado completamente una rama.

En la LPSD, el Motor de Inferencia aplica el recorrido en profundidad en el árbol de los grafos que representan las reglas de la Base de Conocimientos.

k) Usuario: Un usuario (final) del sistema se define como una persona que solicita realizar un diagnóstico.

l) Casos de Uso: Un caso de uso se define en UML como “un conjunto de acciones que realiza el sistema y que tiene un resultado observable y de interés para un actor particular del mismo” [OMG]. Los casos de uso constituyen una especificación del sistema basada en la funcionalidad. El

diagrama de casos de uso muestra las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuario).

m) Casos de uso por usuario: Un usuario (final) del sistema, puede acceder a uno o más casos de uso.

n) Razonamientos: Los razonamientos o estrategias de razonamiento son la forma en que el mecanismo de inferencia realiza el diagnóstico. El mecanismo de inferencia aplica estrategias de razonamiento, utilizando la información del dominio para obtener las propiedades. Los razonamientos más utilizados para las tareas de diagnóstico son el deductivo, el inductivo y el diferencial.
Las estrategias de razonamiento que forman parte de la variabilidad en la LPSD son simulaciones de los tipos de razonamientos más comunes que tiene una persona que realiza un diagnóstico

El razonamiento deductivo o guiado por los datos (también llamado razonamiento con encadenamiento hacia delante o razonamiento "forward") consiste en enlazar los conocimientos a partir del uso de datos (propiedades de la entidad a diagnosticar) con el fin de obtener una solución de un problema. Dado que en este razonamiento se generan nuevas propiedades o hechos, existen dos formas de tratarlos, que son: profundidad cuando un hecho en cuanto se genera se introduce a la Base de Conocimientos, o en anchura cuando no se incorporan los hechos a la Base de Conocimientos hasta que se ha terminado de aplicar la misma. En general, las soluciones pueden ser más de una. De un hecho así deducido se puede asegurar su verdad. La ventaja desde el punto de vista técnico de utilizar este modo

de encadenar el conocimiento es su sencillez y que la entrada de datos es única y se da al principio del programa. Este encadenamiento está basado en el "modus ponens" de la lógica formal que dice que: Si conocemos la regla: si A entonces B, y A es cierto, entonces podemos deducir que B es también cierto.

El encadenamiento hacia adelante se desarrolla según el siguiente procedimiento:

- Hasta que ninguna regla produzca una nueva afirmación o la hipótesis sea identificada:
    - Para cada regla:
        - Trátese de corroborar cada antecedente de la regla mediante su apareamiento con hechos conocidos;
        - Si se corroboran todos los antecedentes de la regla, hágase valer el consecuente, a menos que ya exista una afirmación idéntica;
        - Repítase el procedimiento para todos los apareamientos y alternativas de sustitución.

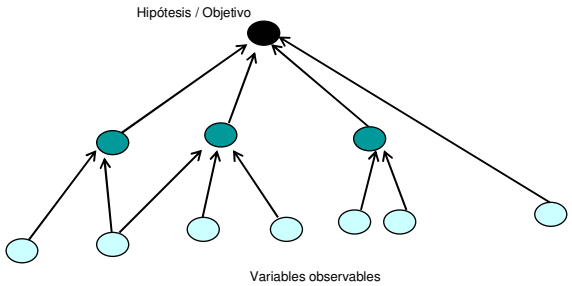


Figura 37. Grafo del razonamiento deductivo

El razonamiento inductivo o guiado por los objetivos (también llamado razonamiento con encadenamiento de reglas hacia atrás o razonamiento "backward") consiste en

comprobar que un objetivo es cierto en base a unos hechos que forman el universo del sistema. Este encadenamiento tiene su fundamento en el "modus tollens" de la lógica formal que dice que si conocemos la regla: Si A entonces B y también conocemos que A es falso, entonces podemos inducir que B también lo es.

El encadenamiento hacia atrás se desarrolla según el siguiente procedimiento:

- Hasta que todas las hipótesis se hayan identificado y ninguna se pueda comprobar o hasta que las hipótesis sean verificadas:
    - Para cada hipótesis:
        - Para cada regla cuyo consecuente coincida con la hipótesis en cuestión,
            - Inténtese corroborar cada uno de los antecedentes de la regla mediante su apareamiento con afirmaciones de la memoria en funcionamiento o mediante encadenamiento regresivo a través de otra regla, creando nuevas hipótesis. Asegúrese de verificar todas las alternativas de unificación y sustitución.
            - Si todos los antecedentes de la regla logran ser corroborados, notifíquese el éxito y concluya que la hipótesis es verdadera.

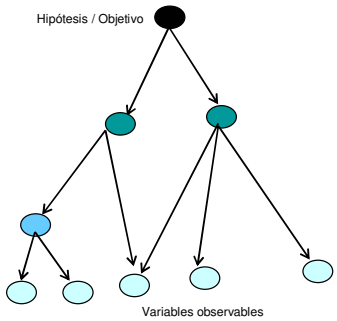


Figura 38. Grafo del razonamiento inductivo

El razonamiento diferencial.- Un diagnóstico diferencial se presenta cuando hay que comparar entre dos o más posibilidades diagnósticas o hipótesis. En el razonamiento diferencial se lleva a cabo el siguiente proceso: primero se realiza el razonamiento deductivo con el fin de llegar a un diagnóstico; si se llegase a dos o más posibilidades diagnósticas, se realizará el razonamiento diferencial, invocando al razonamiento inductivo para reformular preguntas al usuario o buscar datos que permitan determinar cuál de las posibilidades diagnósticas es la más certera al problema analizado. i.e. verificar las soluciones o hipótesis.

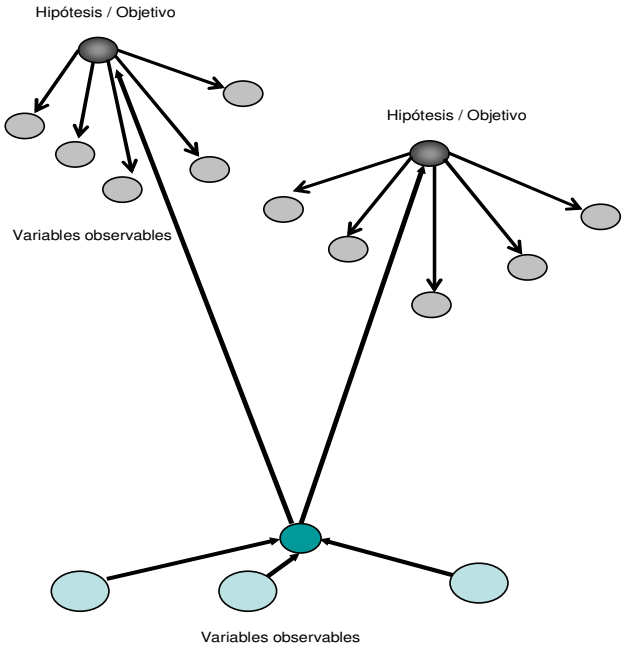


Figura 39. Grafo del razonamiento diferencial

Conceptos utilizados en los procesos de la ingeniería de la LPSD

Con el fin de clarificar la terminología utilizada en esta tesis para la descripción y el modelado de las tareas realizas en los procesos de la ingeniería de la LPSD, a continuación se definen y/o describen algunos conceptos manejados tanto en la primera fase de la ingeniería del dominio, como en la segunda fase de la ingeniería de la aplicación.

a) Esqueletos: El ingeniero del dominio debe ser suficientemente experto para determinar la plataforma común

que comparte toda la familia de productos. Una plataforma de una LPS es “el conjunto de subsistemas software e interfaces que forman una estructura común desde la cual un conjunto de productos pueden ser producidos y desarrollados eficientemente” [Meyer et al., 1997]. Las partes que componen la plataforma son denominados componentes básicos reutilizables [Clements et al., 2002]. Estas pueden incluir la arquitectura, los componentes software, los modelos diseñados, etc. En general, puede ser utilizado cualquier artefacto [Pohl et al., 2006]. La plataforma es la base sobre la cual los productos pueden ser creados adicionando características (variabilidad).

Nuestra plataforma consiste en un conjunto de esqueletos o plantillas de elementos arquitectónicos en el marco del metamodelo PRISMA. Por ello las partes que componen la plataforma de nuestra LPS serán un conjunto de esqueletos de componentes, conectores, aspectos e interfaces.

Los esqueletos son plantillas especificadas en el LDA de PRISMA que contienen “huecos” de valor semántico, que posteriormente serán rellenados con las características del dominio de aplicación del diagnóstico. Los esqueletos representan plantillas de los elementos arquitectónicos, sus aspectos e interfaces. Estos esqueletos son representados metafóricamente como lo muestra la figura X. Nótese que estos iconos están vacíos, por representar las plantillas con huecos que al ser rellenados conformarán los tipos. Por ello la metáfora visual de los tipos se representa como estos mismos iconos pero con color, i.e. rellenados con las características que fueron insertadas, como lo muestra la siguiente figura.

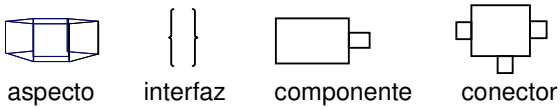


Figura 40. Metáfora visual de los esqueletos PRISMA

Los esqueletos de los cuatro elementos arquitectónicos básicos de la LPSD representan: los componentes Motor de Inferencia, Base de Conocimientos y Módulo de Comunicación o Interfaz del Usuario, y el Conector Diagnóstico. Los aspectos necesarios para la definición de estos elementos arquitectónicos son los aspectos funcionales de cada uno de los componentes, así como el aspecto de coordinación del conector. Dichos aspectos utilizan el conjunto de servicios de las interfaces.

b) Artefacto: Un artefacto (software) es cualquier producto de trabajo creado durante un proceso de desarrollo del software, tal como modelos, documentos de requerimientos, ficheros de código fuente, ficheros de configuración XML, DOC, etc., que resuelven problemas recurrentes en el desarrollo de software. Estos artefactos aceptarán cierta variabilidad, por lo que contarán con puntos de variabilidad que podrán ser configurados para adaptarlos a las características del problema concreto donde aplicarlos.

c) Activo: Un activo es definido como una colección cohesiva de artefactos que resuelven un problema específico o un conjunto de problemas, así como la metainformación para facilitar su reutilización. Un activo se almacena como un fichero comprimido que empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos. Un activo de la LPSD está conformado por un esqueleto y su proceso de inserción de características.

d) Metainformación: La metainformación que describe el activo, es definida por un fichero configurado en XML, que contendrá el modelo RAS del activo.

e) Modelo RAS del activo: El modelo RAS de un activo es un modelo que conforma con el metamodelo RAS. Este modelo hace referencia a los artefactos contenidos en el activo, así como a actividades que definen procesos o guías de uso del mismo. Estas actividades permitirán configurar la variabilidad de los artefactos contenidos en el activo para obtener del mismo soluciones concretas. Este modelo se empaqueta junto al resto de artefactos dentro del fichero comprimido que representa al activo, conteniendo la identificación y la clasificación del activo, la descripción de sus artefactos, sus puntos de variabilidad y cómo configurarlos

f) Activo empaquetado: Un activo empaquetado es un paquete software conformado por el propio activo y su correspondiente modelo RAS del activo. Un activo empaquetado es una colección cohesiva de artefactos que resuelven un problema específico o un conjunto de problemas, así como metainformación para facilitar su reutilización. De hecho, un activo se almacena como un fichero comprimido que empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos.

g) Base-Line: La línea de productos de software requiere almacenar sus activos de software en repositorios. Un repositorio LPS es una base de datos especializada que almacena activos de software y facilita la recuperación y el mantenimiento de los activos de software. Su objetivo es

asegurar la disponibilidad de activos para apoyar el desarrollo de productos de la LPS.

Tomando en cuenta esto, el conjunto de esqueletos, sus procesos de inserción de características y los modelos conceptuales del dominio de aplicación, son depositados en un repositorio que conforma la "*Base-Line*" de la LPSD. Por ello se define la *Base-Line* como un repositorio de artefactos software que van a ser seleccionados según la variabilidad del dominio, y usados como base sobre la que se añadirán las características ("features") que conformarán los tipos de artefactos software de la LPSD.

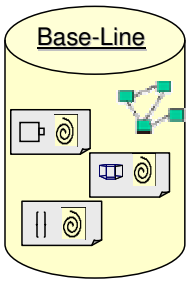


Figura 41. Metáfora visual de la Base-Line

h) Tipos de la LPSD.- De esta manera, se puede considerar que los tipos (artefactos software PRISMA) de la LPSD son los esqueletos rellenos de los elementos arquitectónicos. Es decir, son las plantillas o esqueletos conteniendo ya la información de cada una de las características o "features" del dominio de aplicación del diagnóstico, i.e. el dominio específico del caso de estudio.

Figura 42. Metáfora visual de los esqueletos PRISMA

Cabe señalar que, como un mismo esqueleto puede ser aplicado a distintos tipos. En el estudio de campo realizado en esta tesis (ver capítulo X de esta tesis), contamos con un par de casos que comparten los mismos esqueletos. Estos casos de estudio son el diagnóstico televisivo y el diagnóstico educativo.

i) Arquitectura de la LPSD: Uno de los puntos más importantes (o el más importante) de una LPS es la definición de la arquitectura de la línea de productos (ALP), debido a que esta arquitectura determinará el alcance de la línea de productos y las características de los productos que pueden desarrollarse. La ALP es la llave para la reutilización sistemática, ya que describe la estructura de los productos del dominio, mostrando sus elementos arquitectónicos y las relaciones entre los mismos. Definir una adecuada ALP requiere armonizar las cualidades que se persiguen para los productos de la LPS con la definición de elementos opcionales, alternativos o variables.

La arquitectura LPS debe ser instanciada cada vez que se desarrolla un producto de la línea.

Las instancias de los tipos (PRISMA) configurarán las arquitecturas software de la LPSD. Por lo tanto, la LPSD serán los sistemas de diagnóstico de cada uno de los dominios específicos.

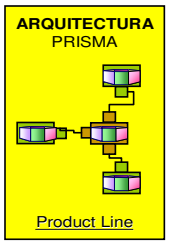


Figura 43. Metáfora visual de una arquitectura PRISMA

Apéndice B:

ESTUDIO DE CAMPO EN EL DIAGNÓSTICO

En este apéndice se presenta el estudio de campo realizado en el dominio del diagnóstico. Dicho estudio contempla cinco casos de estudio en los que se observa la variabilidad del dominio de aplicación. Cada uno de los casos de estudio representan un tipo de diagnóstico:

- diagnóstico médico
- diagnóstico de desastres
- diagnóstico de becas
- diagnóstico educativo
- diagnóstico televisivo

Estudio de campo en el diagnóstico

Se ha realizado un estudio de campo en el dominio del diagnóstico sobre el que se observa la variabilidad de las características de la LPSD. Dicho estudio abarca varios casos de estudio, que a continuación se comentan.

Caso 1

El tipo de diagnóstico es el **diagnóstico médico**, en el cual la entidad a diagnosticar es el paciente y el resultado del diagnóstico la enfermedad que padece el paciente. Las propiedades de las entidades son diferentes en cada hipótesis que puede resultar del proceso del diagnóstico.

Se realiza un diagnóstico clínico el cual debe coincidir con el diagnóstico de laboratorio para finalmente obtener un diagnóstico integral con la aportación de ambos, con el fin de obtener una buena calidad diagnóstica.

Los casos de uso son:

- caso de uso 1: realizar diagnóstico clínico
- caso de uso 2: realizar diagnóstico de laboratorio
- caso de uso 3: obtener resultados del diagnóstico (diagnóstico integral)

Los casos de uso son utilizados por dos usuarios finales: el médico y el encargado del laboratorio. Los casos de uso 1 y 3 son realizados por el médico; por ello el usuario médico desempeña dos roles. El caso de uso 3 es realizado por el encargado del laboratorio, quien desempeña un sólo rol.

Las propiedades son los signos y síntomas del paciente; los cuales están clasificados en dos niveles de abstracción: los de grano grueso y los de grano fino.

Las propiedades del nivel 0 son los signos y síntomas de grano grueso (p.e. tos y fiebre), cuyos valores son preguntados al usuario final.

Las prehipótesis son los síndromes (p.e. ira), que son inferidos por el sistema a través de las reglas del nivel 1

Las propiedades del nivel 1 son los signos y síntomas de grano fino (p.e. tos seca y fiebre continua), cuyos valores son también preguntados al usuario final.

Las hipótesis son las enfermedades (.p.e neumonía), i.e. el resultado del diagnóstico inferido a través de las reglas del último nivel.

El tipo de razonamiento aplicado es el diferencial. Al inicio se realiza un razonamiento deductivo el cual infiere varias hipótesis (i.e. posibles enfermedades), por lo que se invoca al razonamiento inductivo para poder diferenciar entre esas hipótesis, la hipótesis validada que es el resultado del diagnóstico (i.e. la enfermedad).

El escenario del proceso del diagnóstico médico es el siguiente:
Con valores de los signos y síntomas de grano grueso que el usuario final ingresa al sistema, es inferido el síndrome. Esta parte del proceso se realiza con el razonamiento deductivo. Dicho síndrome da lugar (por deducción) a dos o más

posibles enfermedades (posibles hipótesis). Estas hipótesis deben ser validadas por lo que el sistema solicita al usuario final los datos de los signos y síntomas de grano fino para así inferir la enfermedad (hipótesis validada) que padece el paciente. Este última parte del proceso se realiza con el razonamiento inductivo.

En el grafo que a continuación se presenta, por simplicidad, sólo se han incluido 5 hipótesis (enfermedades) que pueden derivarse del proceso del diagnóstico médico.

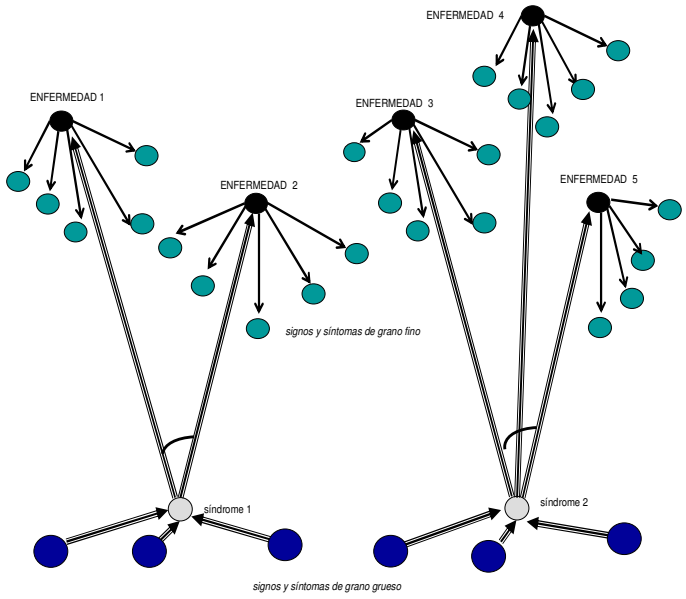


Figura 44. Grafo que muestra el caso de un diagnóstico médico

Caso 2

El tipo de diagnóstico es el **diagnóstico de desastres**, en el cual la entidad a diagnosticar es una víctima de un desastre, y el resultado del diagnóstico es la indicación del color de la etiqueta que se le coloca a la víctima, la cual identifica el tipo de atención que deberá darse a la misma en dicha emergencia. Las propiedades de las entidades son diferentes en cada hipótesis que puede resultar del proceso del diagnóstico.

En este caso de estudio se cuenta solamente un caso de uso (indicar etiqueta) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son los signos vitales que manifiesta la víctima.

Las propiedades del nivel 0 son la indicación de si una víctima es o no es ambulatoria, cuyo valor es preguntado al usuario final.

Las propiedades del nivel 1 son el resto de los signos vitales (p.e. perfusión), cuyos valores son preguntados al usuario final a través de las reglas del nivel 1.

Las hipótesis corresponden al estado de intervención de emergencia en la víctima (.p.e inmediato), i.e. el resultado del diagnóstico inferido a través de las reglas del último nivel.

El tipo de razonamiento aplicado es el diferencial. Al inicio se realiza un razonamiento deductivo el cual infiere una hipótesis que será el resultado del diagnóstico, o bien inferirse varias hipótesis, por lo que se invoca al razonamiento inductivo para

poder diferenciar entre esas hipótesis, la hipótesis validada que es el resultado del diagnóstico.

El escenario para el proceso del diagnóstico de desastres es el siguiente:

Con valores del signo vital de si es o no es ambulatoria la víctima que el usuario final ingresa al sistema, es inferida la hipótesis final (menor importancia) con lo cual termina el proceso del diagnóstico, o bien las posibles hipótesis (atención inmediata, no salvable, retardado). Esta parte del proceso se realiza con el razonamiento deductivo. En el caso de que se hayan generado varias hipótesis, éstas deben ser validadas por lo que el sistema solicita al usuario final los datos del resto de los signos vitales de la víctima para así inferir la etiqueta que colocará a la víctima del desastre (hipótesis validada). Este última parte del proceso se realiza con el razonamiento inductivo.

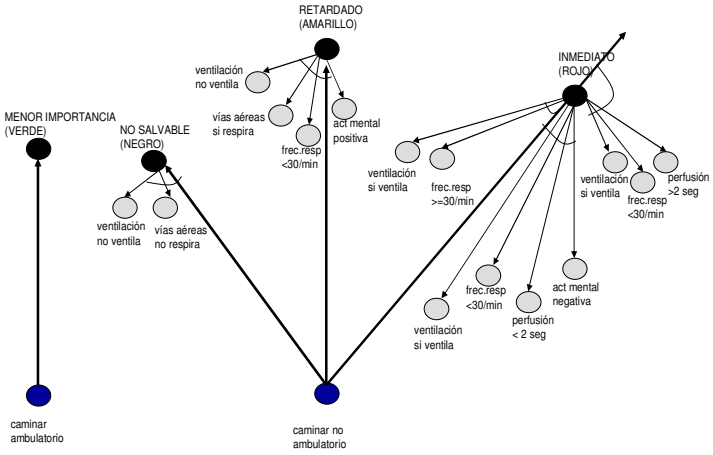


Figura 45. Grafo que muestra el caso de un diagnóstico en desastres

Caso 3

El tipo de diagnóstico es el **diagnóstico de becas**, en el cual la entidad a diagnosticar es un candidato a obtener una beca, y el resultado del diagnóstico es la decisión de si se le debe otorgar la beca a dicho candidato. Las propiedades de las entidades son las mismas, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

Este caso de estudio cuenta con un caso de uso (realizar decisión de otorgamiento) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son los requisitos que debe cumplir el candidato para ser beneficiado con una beca, y corresponden al nivel 0 (p.e. promedio de calificaciones).

La hipótesis es la indicación de si debe o no debe otorgarse la beca al candidato, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 1 (único nivel de reglas en este caso de estudio).

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico de becas es el siguiente:
Con valores que el usuario final ingresa al sistema sobre los requisitos que debe cumplir un candidato a beca, es inferida la hipótesis, deduciendo así la decisión de otorgamiento de beca

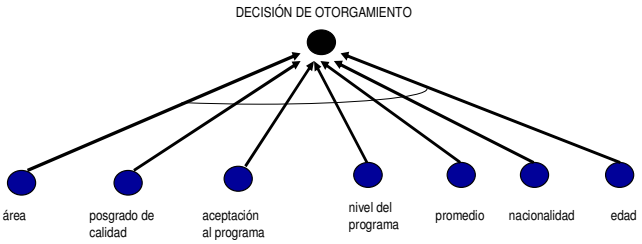


Figura 46. Grafo que muestra el caso de un diagnóstico de becas

Caso 4

El tipo de diagnóstico es el **diagnóstico educativo**, en el cual la entidad a diagnosticar es un programa educativo de postgrado, y el resultado del diagnóstico es la etapa de desarrollo que tiene el programa educativo. Las propiedades de las entidades son las mismas, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

En este caso, se ha considerado un caso de uso (obtener etapa de desarrollo) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son los rubros y subrubros que debe se contemplan en la evaluación de un programa educativo.

Las propiedades del nivel 0 son los subrubros cuyo valor es ingresado al sistema por el usuario final (p.e. control de calidad de alumnos).

Las propiedades del nivel 1 son los rubros cuyo valor es inferido por el sistema aplicando las reglas del nivel 1 (p.e. plan de estudios).

La hipótesis es la calificación que se le otorga al plan de estudio por medio de su etapa de desarrollo, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 2.

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico educativo es el siguiente:
Con valores que el usuario final ingresa al sistema sobre los subrubros involucrados en la evaluación del programa educativo, es inferido por deducción el valor de los rubros y posteriormente es inferida la hipótesis, deduciendo así la etapa de desarrollo del programa educativo.

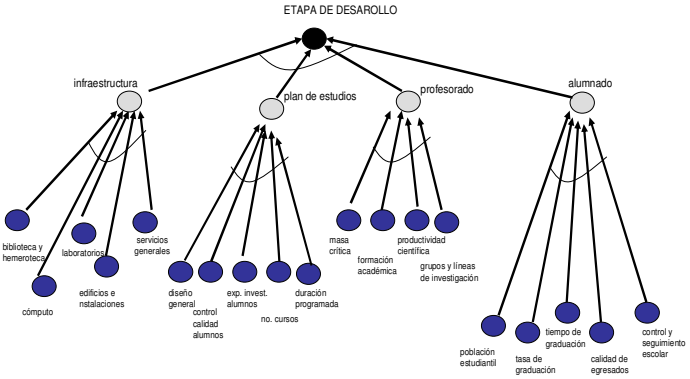


Figura 47. Grafo que muestra el caso de un diagnóstico educativo

Caso 5

El tipo de diagnóstico es el **diagnóstico televisivo**, en el cual la entidad a diagnosticar es un video, y el resultado del diagnóstico es el estado del video con el fin de tomar la decisión de si debe o no transmitirse al aire dicho video. Las propiedades de las entidades son las mismas, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

En este caso de estudio, se ha contemplado un caso de uso (obtener decisión de transmisión) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son las características del video.

Las propiedades del nivel 0 son las subcaracterísticas del video, cuyo valor es ingresado al sistema por el usuario final (p.e. imágenes).

Las propiedades del nivel 1 son los rubros cuyo valor es inferido por el sistema aplicando las reglas del nivel 1 (p.e. producción).

La hipótesis es el estado del video en el que se encuentra, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 2.

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico televisivo es el siguiente:

Los valores de las subcaracterísticas del video son ingresadas al sistema por el usuario final. Con lo cual es inferido por deducción el valor de las características del video y finalmente es inferida la hipótesis, deduciendo así el estado del video.

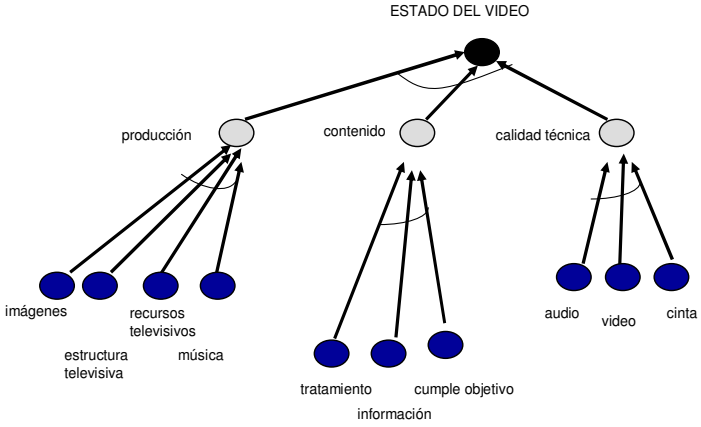


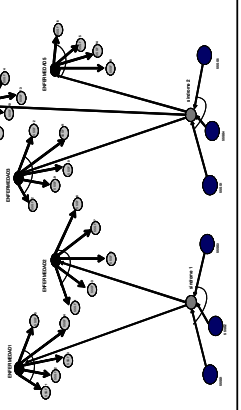
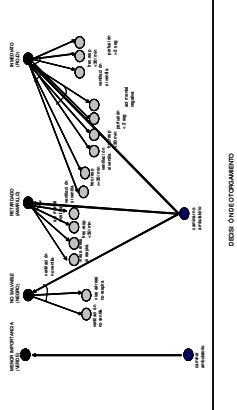
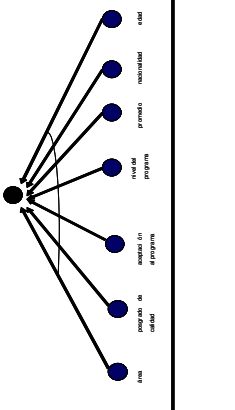
Figura 48. Grafo que muestra el caso de un diagnóstico televisivo

La siguiente tabla muestra un concentrado de las diferentes características de cada uno de los casos de estudio realizados:

	DOMINIO entidad/diagnóstico	Tipo propiedades (cambian/ mismas)	No. Hipótesis	No. Niveles granularidad de propiedades	Tipo Razonamiento	No. Casos de uso	No. usuarios	No. Roles por suario
1	DIAG. MÉDICO paciente/enfermedad	cambian propiedades	5	3	Diferencial	3	2	1 2
2	DIAG. EN DESASTRES víctima/etiqueta	cambian propiedades	4	1	Diferencial	1	1	1
3	DIAG. DE CANDIDATO A BECA candidato/otorgamiento de beca	mismas propiedades	1	1	Deductivo	1	1	1
4	DIAG. EDUCATIVO prog.educativo/etapa de desarrollo	mismas propiedades	1	2	Deductivo	1	1	1
5	DIAG. TELEVISIVO video/transmisión	mismas propiedades	1	2	Deductivo	1	1	1

Tabla 2. Características de los casos de estudio

Para una mejor comprensión de la variabilidad de nuestra LPSD, en la tabla de abajo se presenta un concentrado de los casos de estudio realizado, indicando las características del dominio específico y la representación de algunas de ellas (nivel de propiedades, tipo de razonamientos, número de hipótesis) a través de un grafo.

TIPO DE DIAGNÓSTICO *ENTIDAD/RESULTADO DIAG.*	*CARACTERÍSTICAS* *(VARIABILIDAD)*	*GRAFOS*
Diag. Médico: *paciente/enfermedad*	Cambian propiedades Razonamiento diferencial 5 hipótesis 3 niveles de propiedades 3 casos de uso 2 usuarios 2 roles	
Diag. Desastres: *víctima/etiqueta*	Cambian propiedades Razonamiento diferencial 4 hipótesis 2 niveles de propiedades 1 caso de uso 1 usuario 1 rol	
Diag. Becas: *candidato/otorgar beca*	Mismas propiedades Razonamiento deductivo 1 hipótesis 1 nivel de propiedades 1 caso de uso 1 role	

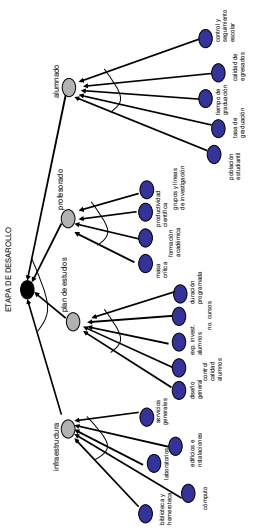
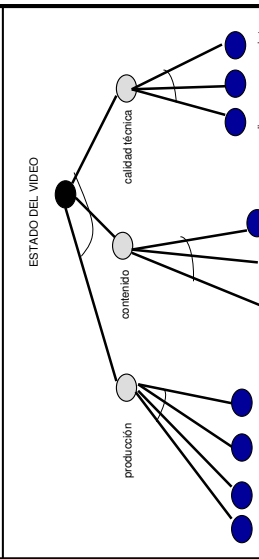
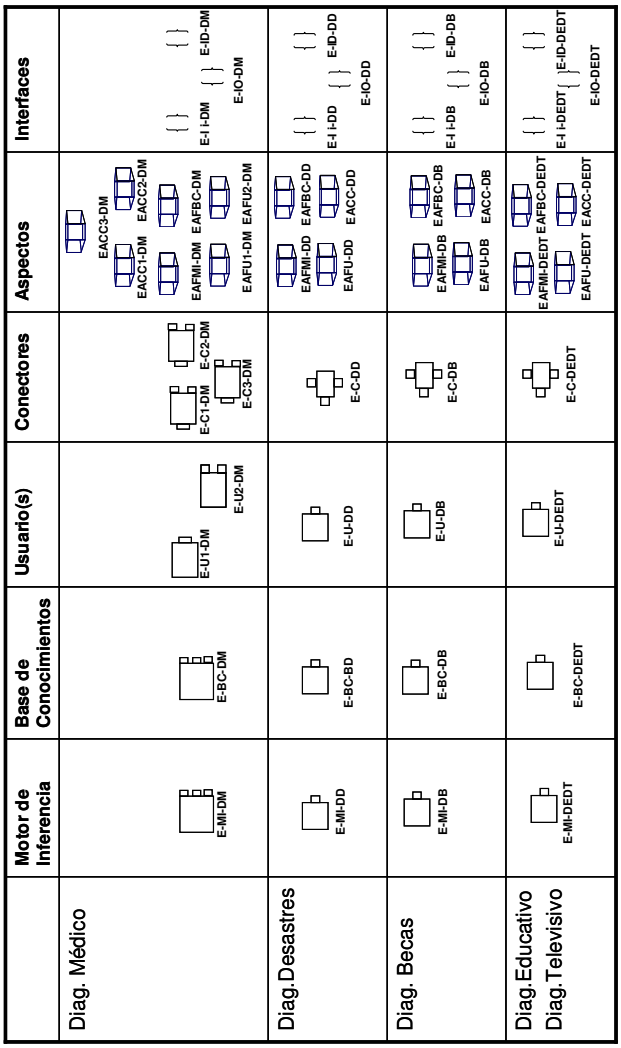
Diag. Educativo: *prog. educativo/etapa de desarrollo*	Mismas propiedades Razonamiento deductivo 1 hipótesis 2 niveles de propiedades 1 caso de uso 1 usuario 1 rol	
Diag. Televisivo: *video/transmitir video*	Mismas propiedades Razonamiento deductivo 1 hipótesis 2 niveles de propiedades 1 caso de uso 1 usuario 1 rol	

Tabla 3 Grafo que muestra la variabilidad del nivel de propiedades de las entidades, el tipo de razonamiento y el número de hipótesis de los casos de estudio

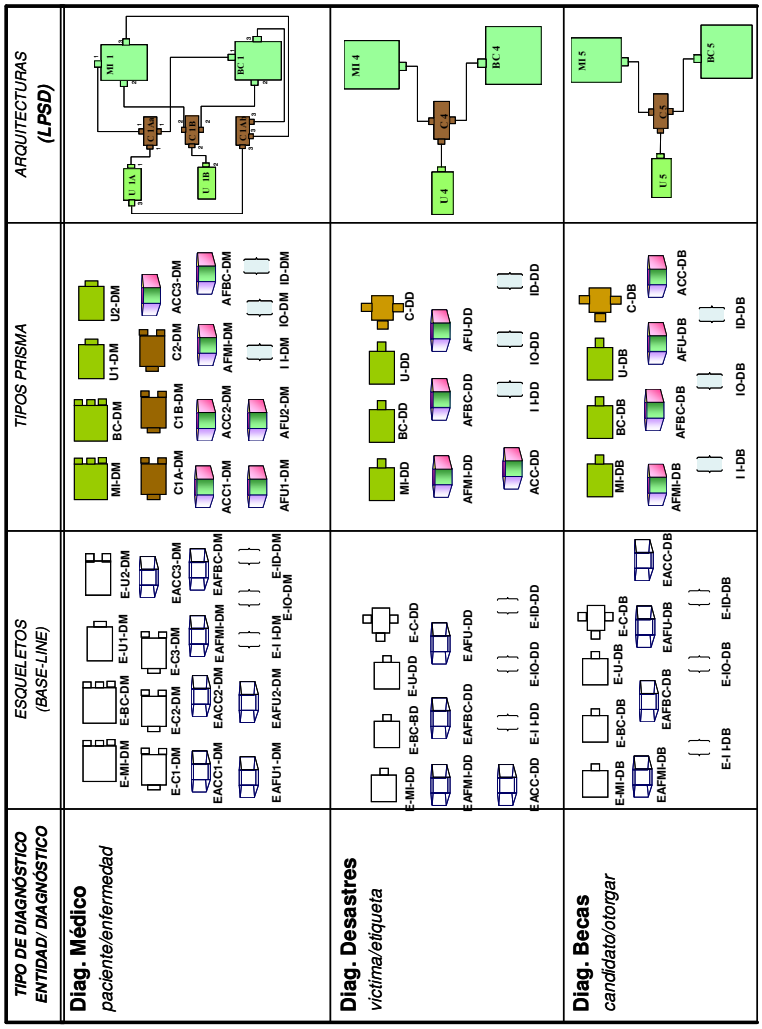
La siguiente tabla muestra una metáfora visual de los esqueletos de los casos de estudio.

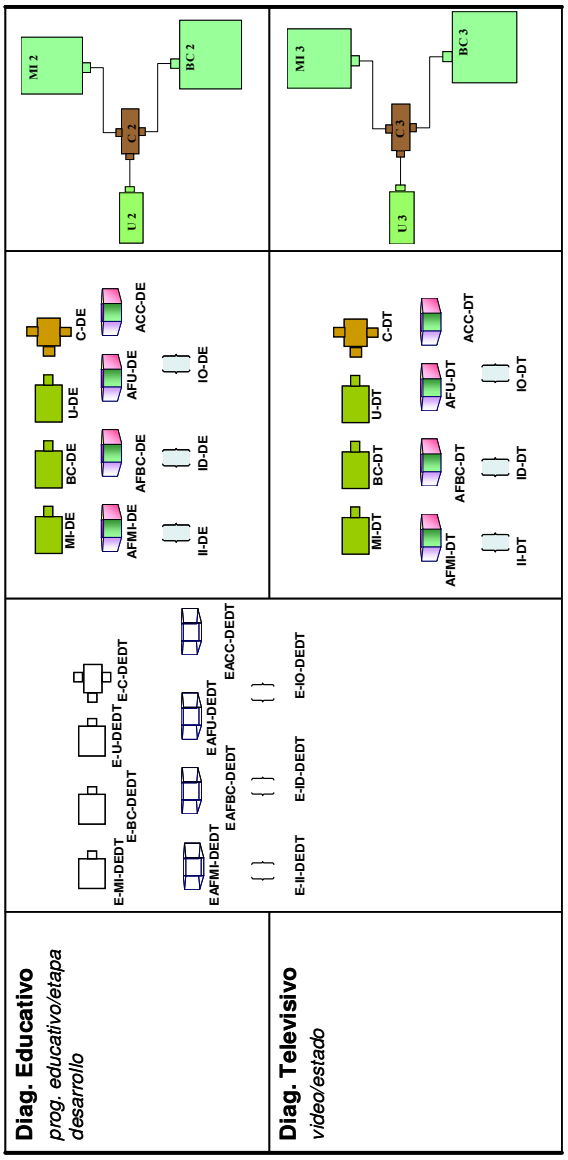


NOMENCLATURA: **E**= Esqueleto o plantilla, **MI**=Componente Motor de Inferencia, **BC**=Componente Base de Conocimientos, **U**=Componente Usuario, **C**=Conector Coordinador, **AF**=Aspecto Funcional, **AC**=Aspecto de Coordinaciòn, **I**=Interfaces (**I**=Inference, **D**=Domain, **O**=Operator),
DM= Diag. Médico, **DD**=Diag. Desastres, **DB**=Diag. Becas, **DE**=Diag. Educativo, **DT**=Diag.Televisivo

Tabla 4. Esqueletos de los casos de estudio

Con el fin de aclarar lo que es la LPSD desarrollada en este trabajo, se muestra en la siguiente tabla, para cada uno de los casos de estudio, una metáfora visual de los esqueletos con sus correspondientes tipos y la configuración de su arquitectura.





NOMENCLATURA: **E**= Esqueleto o plantilla, **MI**=Componente Motor de Inferencia, **BC**=Componente Base de Conocimientos, **U**=Componente Usuario, **C**=Conector Coordinador, **A**=Aspectos, **I**=Interfaces,
DM= Diag. Médico, **DD**=Diag. Desastres, **DB**=Diag. Becas, **DE**=Diag. Educativo, **DT**=Diag.Televisivo

Tabla 5 Esqueletos, tipos y modelos arquitectónicos de los casos de estudio

Finalmente, para una mayor comprensión, el caso de estudio del diagnóstico televisivo es presentado con más detalle en el apéndice C de esta tesis.

Apéndice C

UN CASO DE ESTUDIO: DIAGNÓSTICO TELEVISIVO

Se ha seleccionado un caso de estudio: el diagnóstico televisivo para ser presentado en este apéndice, con la finalidad de realizar un panorama sobre las características de un producto específico de la LPSD. Explícitamente se presenta:

- la información del dominio;
- las propiedades, las reglas, la hipótesis y el tipo de razonamiento;
- el modelo de características, el árbol de decisión y el modelo conceptual del dominio de aplicación;
- la metáfora gráfica de los esqueletos, los tipos y la arquitectura del sistema final;
- la especificación en el LDA de PRISMA de cada uno de los esqueletos y tipos correspondientes a este caso de estudio.

El sistema DiagTV es un producto de nuestra línea de producto que provee capacidades para seleccionar videos y tomar la decisión de transmitirlo o no transmitirlo al aire, a través del diagnóstico realizado al estado en el que se encuentra dicho video.

La información del dominio de este caso de estudio consiste en:

```
a) propiedades del nivel 0:
      imagenes: string,
      estructura televisiva: string,
      recursos televisivos: string,
      musicalización: string,
      tratamiento: string,
      información:string,
      objetivo: string,
      audio: string,
      video: string,
     cinta: string,

b) propiedades del nivel 1.
      produccion: string,
      contenido: string,
     calidad tecnica: string,

c) hipótesis:
     estado del video: string,

 d) reglas del nivel 1 (sólo se muestran algunas):
     IF (imágenes = ”adecuadas” and estructura
televisiva= “autorizada” and recursos televisivos=
“buenos” and musicalizacion= “buena”) THEN
produccion:= “buena”
```

IF (tratamiento= “didactico” and información= correcta” and objetivo= ”si lo cumple”) THEN contenido:=bueno”

IF (audio= “aceptable” and video= “aceptable” and cinta= ”buen estado”) THEN calidad tecnica:= “buena”

e) reglas del nivel 2 (sòlo se muestran algunas):
IF (producción:= “buena” and contenido:= “bueno” and calidad tecnica:= “buena”) THEN estado del video:= “video OK”

f) razonamiento:
deductivo

g) casos de uso:
Realizar diagnóstico televisivo

h) usuarios:
un solo usuario final: el técnico

i) casos de uso por usuario:
un solo usuario participa en el caso de uso: realizar diagnóstico televisivo

El siguiente grafo muestra las propiedades, las reglas, la hipótesis y el tipo de razonamiento de este caso de estudio

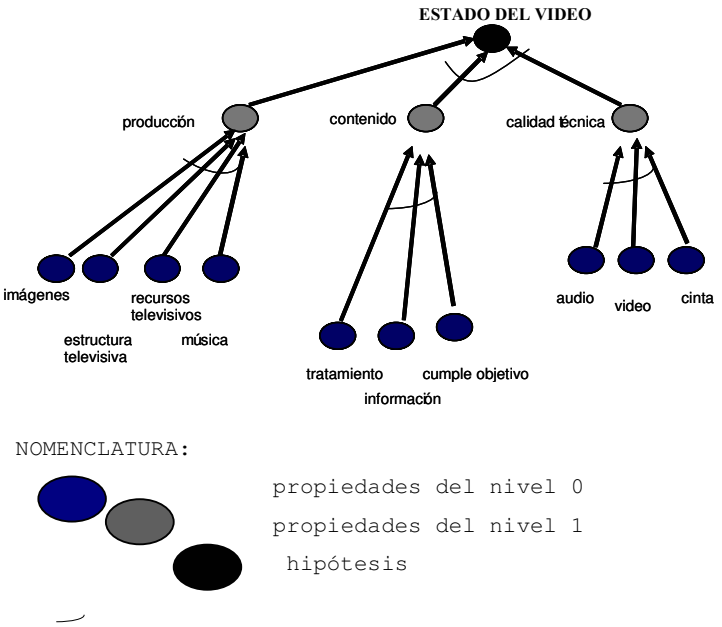
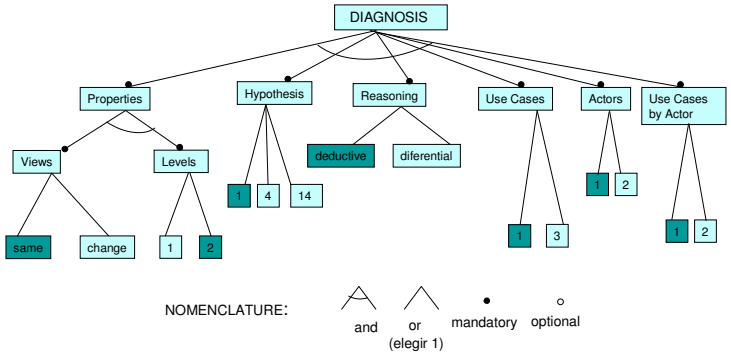


Figura 49. Grafo que muestra las propiedades, las reglas, la hipótesis y el tipo de razonamiento del diagnóstico televisivo

En este dominio específico las características del modelo de características representadas en el árbol de decisión, contemplan la selección indicada en la figura con el color azul resaltado.



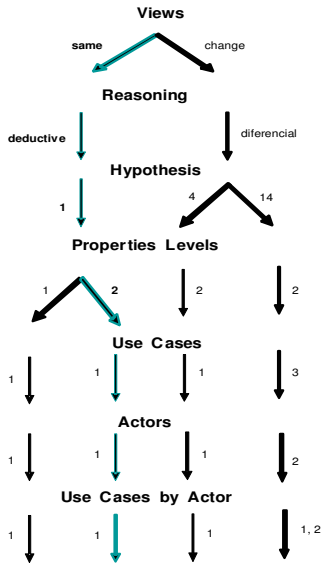
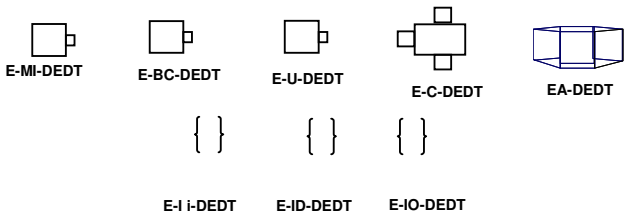


Figura 50. Selección en el modelo de características y en el árbol de decisión para el caso de estudio: diagnóstico televisivo

Con esta información de variabilidad es seleccionado los esqueletos de la "Base-Line", y que son mostrados en la figura x::



NOMENCLATURA: E= Esqueleto o plantilla, MI=Componente Motor de Inferencia, BC=Componente Base de Conocimientos, U=Componente Usuario, C=Conector Coordinador, A=Aspectos, I=Interfaces (I=Inference, D=Domain, O=Operator), DT=Diag.Televisivo

Figura 51. Esqueletos PRISMA del caso de estudio: diagnóstico televisivo

Estos esqueletos serán rellenados con la información específica del dominio contenida en el modelo conceptual del dominio de aplicación de la figura 52, para conformar los tipos de los elementos arquitectónicos, y que son mostrados en la figura 52:

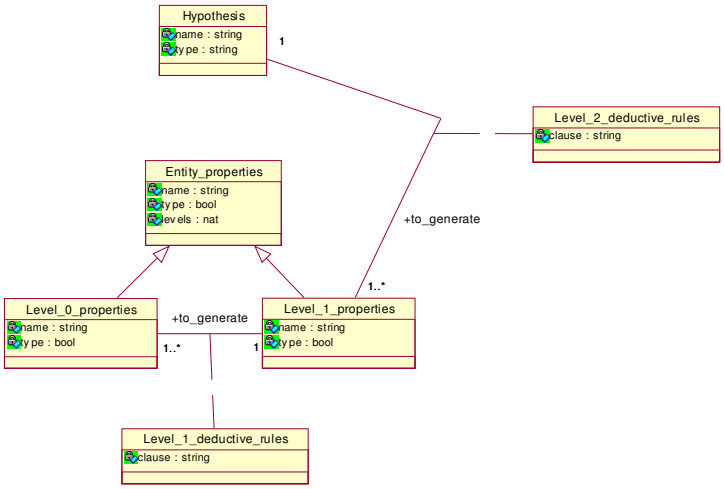


Figura 52. Modelo conceptual del dominio de aplicación del caso de estudio: diagnóstico televisivo

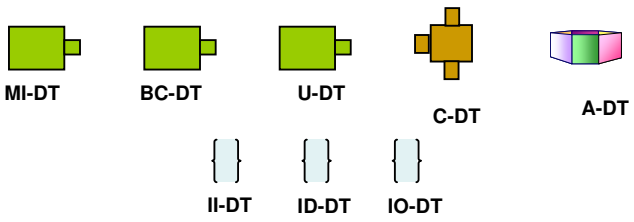


Figura 53. Tipos PRISMA del caso de estudio: diagnóstico televisivo

Finalmente al configurar el sistema, se instancian estos tipos de elementos arquitectónicos para conformar la arquitectura software del producto específico de nuestra línea de productos, y que a continuación se muestra:

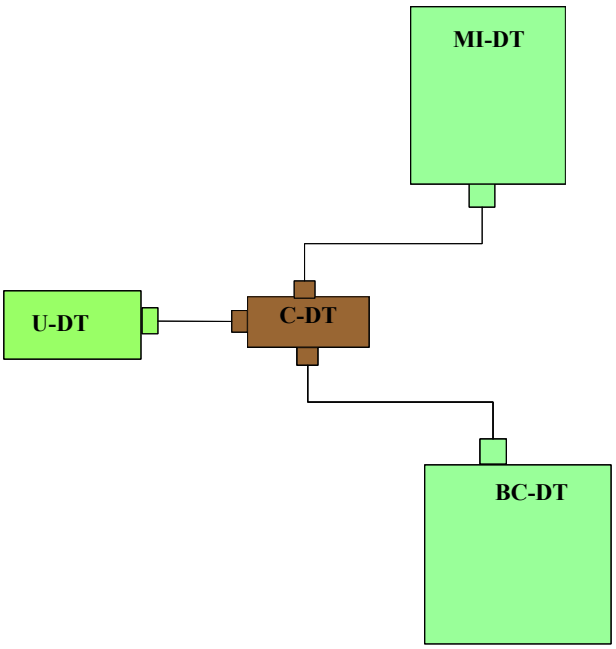


Figura 54. Modelo arquitectónico del caso de estudio: diagnóstico televisivo

El modelo arquitectónico del sistema DiagTV presenta la metáfora visual de la figura 55, sobre la herramienta PRISMA-CASE .

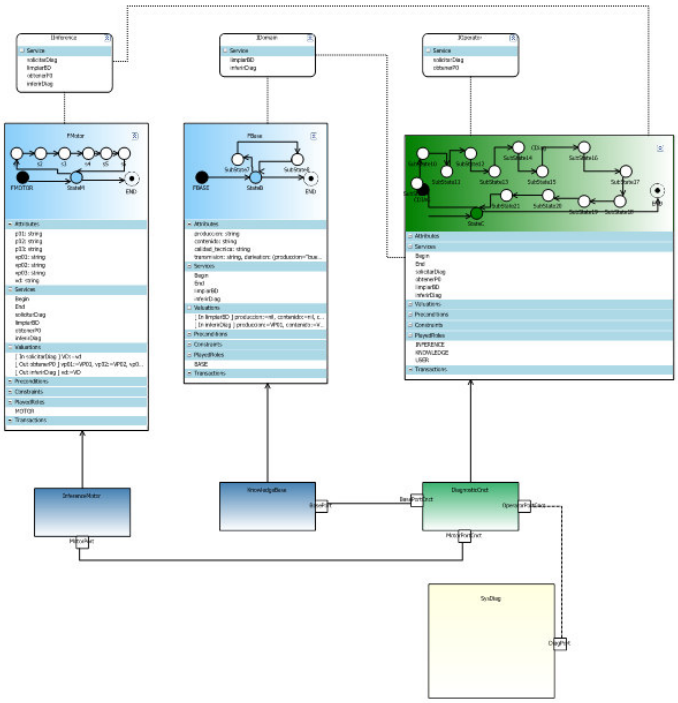


Figura 55. Metáfora visual de la herramienta PRISMA-CASE de la arquitectura del sistema de diagnóstico televisivo

En la ejecución del sistema, el técnico televisivo (usuario final) ingresa los valores de las propiedades del video para obtener el resultado del diagnóstico, que en este caso es la indicación de si el video es correcto o no para ser transmitido al aire.

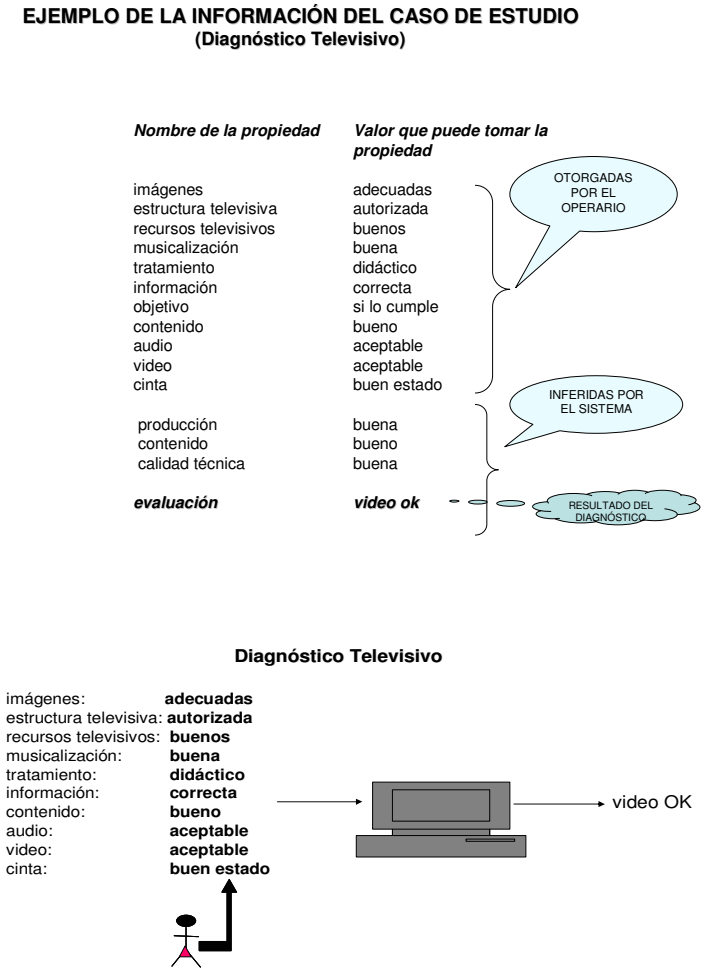


Figura 56. Ejemplo de información de entrada y salida del sistema de diagnóstico televisivo

Especificación del sistema DiagTV en el LDA de PRISMA

(Son considerados solamente los aspectos funcionales de los componentes y el aspecto de coordinación del conector)

1.- Esqueletos:

1.1.- Esqueleto de la Interfaz IDomainDEDT

```
Interface  IDomainDEDT
  limpiarBD ()
  inferirValorPropiedadesN1 (VALOR_PROPIEDADNO,
VALOR_PROPIEDADNI)
  inferirValorDiagnóstico (VALOR_PROPIEDADN1,
VALOR_DIAGNOSTICO)
End_Interface  IDomainDEDT
```

1.2.- Esqueleto de la Interfaz IInferenceDEDT

```
Interface IInferenceDEDT
     limpiarBD (..);
     obtenerDiagnostico ( output
        VALOR_DIAGNOSTICO: string)
     obtenerValorPropiedades0  (input PROPIEDADES0:
        string  list [1..*],
           output VALOR_PROPIEDADES0: string  list
        [1..*])
     inferirValorPropiedades1  (input
        VALOR_PROPIEDADES0: string  list [1..*],
        output VALOR_PROPIEDADES1: string  list
        [1..*])
     inferirValorDiagnostico (input
        VALOR_PROPIEDADES1: string,  list [1..*],
             output VALOR_ DIAGNOSTICO: string)
End_ Interface IInferenceDEDT
```

1.3.- Esqueleto de la Interfaz IOperatorDEDT

```
Interface IOperatorDEDT
     obtenerDiagnostico ( output
       VALOR_DIAGNOSTICO: string)
     obtenerValorPropiedades0  (input PROPIEDADES0:
       string  list [1..*],
          output VALOR_PROPIEDADES0: string  list
       [1..*])
End_ Interface IOperatorDEDT
```

1.4.- Esqueleto del Aspecto Funcional de la Base de Conocimientos

```
Functional Aspect < Hueco nombre aspecto funcional
> using IDomainDEDT
  Attributes
   Variable
    (<Hueco Feature FP.0.n >:<Hueco Feature
FP.0.t>)*;

   Derived
    (<Hueco Feature FP.1.n>:<Hueco Feature
FP.1.t)>)*;
    derivation
    ({<Hueco Feature FR.1>

    (< Hueco Feature FH.n >:< Hueco Feature FH.t
>)*;
    derivation
    {(< Hueco Feature FH.n >=<Hueco Feature FH.v)>
<"and">* }* <Hueco Feature FP.1.n>:= <Hueco Feature
FP.1.v>

Services
   begin;
   in  limpiarBD (..);
       Valuations
       [in  limpiarBD ( )] (<Hueco FP.0.n> := nil
)*;

   in  inferirValorPropiedadesN1  (input
VALOR_PROPIEDADN0:=list , output
VALOR_PROPIEDADN1:list)
       Valuations
       [in  inferirValorPropiedadesN1
(VALOR_PROPIEDADN0, VALOR_PROPIEDADNI)]
       (< Hueco FP.0.n > := VALOR_PROPIEDADN0,
       VALOR_PROPIEDADN1:= <Hueco FP.1.n >)* ;

   in  inferirDiagnostico  (input
VALOR_PROPIEDADN1:=list, output
VALOR_DIAGNOSTICO:string)
       Valuations
       [in  inferirDiagnostico (VALOR_PROPIEDADN1,
VALOR_DIAGNOSTICO)]
       (< Hueco FP.1.n > := VALOR_PROPIEDADN1,
                VALOR_DIAGNOSTICO:= <Hueco FH.n>)*
;

Played_Roles
   KNOWLEDGE  for  IDomainDEDT ::=
       limpiarBD ? (  ):1
       →
       inferirValorPropiedadesN1 ?
(VALOR_PROPIEDADN0, VALOR_PROPIEDADN1)
```

```
      →
      inferirValorPropiedadesN1 !
(VALOR_PROPIEDADN0, VALOR_PROPIEDADN1)
      →
      inferirValorDiagnostico ?
(VALOR_PROPIEDADN1, VALOR_DIAGNOSTICO)
      →
      inferirValorDiagnostico !
(VALOR_PROPIEDADN1, VALOR_DIAGNOSTICO) ;

Protocols
      FBASE ::= begin ( ):1 → P0

      P0 ::=  KNOWLEDGE_limpiarBD ? (  ):1 → P1

      P1          ::=                              (
KNOWLEDGE_inferirValorPropiedadesN1               ?
(VALOR_PROPIEDADN0, VALOR_PROPIEDADN1)
      →
      KNOWLEDGE_inferirValorPropiedadesN1         !
(VALOR_PROPIEDADN0, VALOR_PROPIEDADN1)  ) → P2

      P2::=   (KNOWLEDGE_inferirValorDiagnostico  ?
(VALOR_PROPIEDADN1, VALOR_DIAGNOSTICO)
      →
       KNOWLEDGE_inferirValorDiagnostico          !
(VALOR_PROPIEDADN1,  VALOR_DIAGNOSTICO)  ) →  FIN

      FIN:: = end ( ):1;

End_Functional  Aspect  < Hueco nombre aspecto
funcional>
```

1.5.- Esqueleto del Aspecto Funcional del Motor de Inferencia

```
   Functional  Aspect  < Hueco nombre aspecto
   funcional > using  IInferenceDEDT
     Attributes
     Variables
    (<Hueco Feature FP.0.n>:<Hueco Feature
FP.0.t>)*;
    (<Hueco Feature FP.0.v>:<Hueco Feature
FP.1.t)>)*;
    (<Hueco Feature FP.1.v>:<Hueco Feature
FP.1.t)>)*;
    (<Hueco Feature FH.v>:<Hueco Feature FH.t>)*;

     Services
     begin  ()

     out  limpiarBD (..);
```

```
in  obtenerDiagnostico ( output
   VALOR_DIAGNOSTICO: string)
  Valuations
    [in  obtenerDiagnostico (VALOR_DIAGNOSTICO)
    ]
          VALOR_DIAGNOSTICO:=  <Hueco  Feature
          FH.v>,

out  obtenerValorPropiedades0  (input
   PROPIEDADES0: string  list [1..*],
      output VALOR_PROPIEDADES0: string  list
   [1..*])
  Valuations
    [out  obtenerValorPropiedades0
    (propiedades0, VALOR_PROPIEDADES0) ]
          <Hueco       Feature       FP.0.v>:=
          VALOR_PROPIEDADES0,

out  inferirValorPropiedades1  (input
   VALOR_PROPIEDADES0: string  list [1..*],
   output VALOR_PROPIEDADES1: string  list
   [1..*])
  Valuations
    [out  inferirValorPropiedades1
    (valorPropiedades0, VALOR_PROPIEDADES1) ]
          <Hueco       Feature       FP.1.v>:=
          VALOR_PROPIEDADES1;

out  inferirValorDiagnostico (input
   VALOR_PROPIEDADES1: string,  list [1..*],
   output VALOR_ DIAGNOSTICO: string)
  Valuations
    [out inferirValorDiagnostico
    (valorPropiedades1, VALOR_DIAGNOSTICO)]
          <Hueco       Feature       FH.v>:=
          VALOR_DIAGNOSTICO,

end;

Played_Roles
   INFERENCE=
        IInferenceDEDT.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.LimpiarBD ! ( )
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0  !( PROPIEDADES0,
        VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0 ? (PROPIEDADES0, VALOR_PROPIEDADES0)
        →
```

```
        IInferenceDEDT.InferirValorPropiedades
        1  ! (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorPropiedades
        1 ? (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ! (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ? (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.ObtenerDiagnostico !
        (VALOR_DIAGNOSTICO).

Protocols
   FMOTOR = begin ( ).P0

   P0 =
        INFERENCE.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO).P1

   P1 =
        INFERENCE.LimpiarBD ! ( ).P2

   P2 =
        ( INFERENCE.ObtenerValorPropiedades0
        !( PROPIEDADES0, VALOR_PROPIEDADES0)
        →
        INFERENCE ObtenerValorPropiedades0 ?
        (PROPIEDADES0, VALOR_PROPIEDADES0)
        ).P3

   P3 =
        (INFERENCE.InferirValorPropiedades1  !
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        INFERENCE.InferirValorPropiedades1 ?
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)  ).P4

   P4 =
        (INFERENCE.InferirValorDiagnostico !
        (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        INFERENCE.InferirValorDiagnostico ?
        (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO) ).P5
```

```
        P5 =
             INFERENCE.ObtenerDiagnostico !
             (VALOR_DIAGNOSTICO).FIN

        FIN = end;

   End_Functional Aspect < Hueco nombre aspecto
   funcional >
```

1.6.- Esqueleto del Aspecto Funcional del Usuario

```
Functional Aspect <Hueco nombre aspecto funcional>
using IOperatorDT
   Attributes
     Variables
    (<Hueco Feature FP.0.n>:<Hueco Feature
FP.0.t>)*;
    (<Hueco Feature FP.0.v>:<Hueco Feature
FP.1.t)>)*;
    (<Hueco Feature FH.v>:<Hueco Feature FH.t>)*;

   Services
    begin  ()

    out  obtenerDiagnostico ( output
       VALOR_DIAGNOSTICO: string)
     Valuations
       [out  obtenerValorDiagnostico
       (VALOR_DIAGNOSTICO) ]
             <Hueco        Feature        FH.v>:=
             VALOR_DIAGNOSTICO,

    in  obtenerValorPropiedades0  (input
       PROPIEDADES0: string  list [1..*],
          output VALOR_PROPIEDADES0: string  list
       [1..*])
     Valuations
       [in  obtenerValorPropiedades0
       (PROPIEDADES0, VALOR_PROPIEDADES0) ]
             <Hueco       Feature       FP.0.n>:=
             PROPIEDADES0,
             VALOR_PROPIEDADES0:=  <Hueco  Feature
             FP.0.v>;

    end;

    Played_Roles
       OPERATOR =
            IOperatorDEDT ObtenerDiagnostico !
            (VALOR_DIAGNOSTICO)
            →
            IOperatorDEDT.ObtenerValorPropiedades0
            ? (PROPIEDADES0, VALOR_PROPIEDADES0)
```

```
            →
            IOperatorDEDT ObtenerValorPropiedades0
            ! (PROPIEDADES0, VALOR_PROPIEDADES0)
            →
            IOperatorDEDT ObtenerDiagnostico ?
            (VALOR_DIAGNOSTICO)

    Protocols
       FUSER = begin ( ).P0

       P0 =
             OPERATOR.ObtenerDiagnostico !
            (VALOR_DIAGNOSTICO).P1

       P1 =
            (OPERATOR.ObtenerValorPropiedades0 ?
            (PROPIEDADES0, VALOR_PROPIEDADES0)
            →
            OPERATOR.ObtenerValorSubpropiedades0 !
            (PROPIEDADES0, VALOR_PROPIEDADES0) ).
            P2

       P2 =
            OPERATOR.ObtenerDiagnostico ?
            (VALOR_DIAGNOSTICO).FIN

       FIN = end;

End_Functional Aspect <Hueco nombre aspecto
funcional>
```

1.7.- Esqueleto del Aspecto de Coordinación del Coordinador

```
Coordination Aspect <Hueco nombre aspecto de
coordinación> using IInferenceDEDT,
IDomainDEDT, IOperatorDEDT

  Services
  begin  ( ),

  in/out  limpiarBD (..);

  in/out  obtenerDiagnostico ( output
     VALOR_DIAGNOSTICO: string)

  in/out  obtenerValorPropiedades0  (input
     PROPIEDADES0: string,  list [1..*],
     output VALOR_PROPIEDADES0: string  list
     [1..*])
```

```
in/out  inferirValorPropiedades1  (input
   VALOR_PROPIEDADES0: string  list [1..*],
   output VALOR_PROPIEDADES1: string  list
   [1..*])

in/out  inferirValorDiagnostico (input
   VALOR_PROPIEDADES1: string,  list [1..*],
   output VALOR_ DIAGNOSTICO: string)

end;

Played_Roles
   INFERENCE=
        IInferenceDEDT.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.LimpiarBD ! ( )
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0  !( PROPIEDADES0,
        VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0 ? (PROPIEDADES0, VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.InferirValorPropiedades
        1  ! (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorPropiedades
        1 ? (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ! (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ? (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.ObtenerDiagnostico !
        (VALOR_DIAGNOSTICO).

   KNOWLEDGE=
        IDomainDEDT.LimpiarDB ? ( )
        →
        IDomainDEDT.InferirValorPropiedades0 ?
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IDomainDEDT.InferirValorPropiedades0 !
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
```

```
          →
          IDomainDEDT.InferirValorDiagnostico ?
          (VALOR_PROPIEDADES1,
          VALOR_DIAGNOSTICO)
          →
          IDomainDEDT.InferirValorDiagnostico !
          (VALOR_PROPIEDADES1,
          VALOR_DIAGNOSTICO).

     OPERATOR =
          IOperatorDEDT .ObtenerDiagnostico !
          (VALOR_DIAGNOSTICO)
          →
          IOperatorDEDT
          .ObtenerValorPropiedades0 ?
          (PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          IOperatorDEDT
          .ObtenerValorPropiedades0 !
          (PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          IOperatorDEDT .ObtenerDiagnostico ?
          (VALOR_DIAGNOSTICO)

Protocols
     FMOTOR = begin ( ).P0

     P0 = (
          OPERATOR.ObtenerDiagnostico ?
          (VALOR_DIAGNOSTICO)
          →
          INFERENCE.ObtenerDiagnostico !
          (VALOR_DIAGNOSTICO)
          →
          INFERENCE.LimpiarBD ? ( )
          →
          KNOWLEDGE.LimpiarBD ! ( )
          →
          INFERENCE.ObtenerValorPropiedades0 ? (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          OPERATOR.ObtenerValorPropiedades0 ! (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          OPERATOR.ObtenerValorPropiedades0 ? (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          INFERENCE.ObtenerValorPropiedades0 ! (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          INFERENCE.InferirValorPropiedades1  ?
          (VALOR_PROPIEDADES0,
          VALOR_PROPIEDADES1)
          →
```

```
                KNOWLEDGE.InferirValorPropiedades1 !
                (VALOR_PROPIEDADES0,
                VALOR_PROPIEDADES1)
                →
                KNOWLEDGE.InferirValorPropiedades1 ?
                (VALOR_PROPIEDADES0,
                VALOR_PROPIEDADES1)
                →
                INFERENCE.InferirValorPropiedades1 !
                (VALOR_PROPIEDADES0,
                VALOR_PROPIEDADES1)
                →
                INFERENCE.InferirValorDiagnostico ?
                (VALOR_PROPIEDADES1,
                VALOR_DIAGNOSTICO)
                →
                KNOWLEDGE.InferirValorDiagnostico !
                (VALOR_PROPIEDADES1,
                VALOR_DIAGNOSTICO)
                →
                KNOWLEDGE.InferirValorDiagnostico ?
                (VALOR_PROPIEDADES1,
                VALOR_DIAGNOSTICO)
                →
                INFERENCE.InferirValorDiagnostico !
                (VALOR_PROPIEDADES1,
                VALOR_DIAGNOSTICO)
                →
                INFERENCE.ObtenerDiagnostico ?
                (VALOR_DIAGNOSTICO)
                →
                OPERATOR.ObtenerDiagnostico !
                (VALOR_DIAGNOSTICO)
                ).FIN

        FIN = end;

    End_Coordination Aspect <Hueco nombre aspecto de
    coordinación>
```

1.8.- Esqueleto del Componente Base de Conocimientos

```
Component_Type <Hueco nombre componente >
  Ports
    KnowledgePort: IDomainDEDT,
    Played_Role <Hueco nombre aspecto
funcional>.KNOWLEDGE;
  End_Ports;

  Functional Aspect  import <Hueco nombre aspecto
funcional>;
```

```
  Initialize
     new ( )
     {<Hueco nombre aspecto funcional>.begin ();
     }
  End_Initialize;

  Destruction
     destroy ()
     {<Hueco nombre aspecto funcional>.end ();
     }
  End_Destruction;
End_Component_Type <Hueco nombre componente>;
```

1.9.- Esqueleto del Componente Motor de Inferencia

```
    Component_Type <Hueco nombre componente>
       Ports
         InferencePort: IInferenceDEDT,
         Played_Role  <Hueco nombre aspecto
     funcional>.INFERENCE;
       End_Ports;

       Function Aspect import  <Hueco nombre
     aspecto funcional>;

       Initialize
           new ( )
           {  <Hueco nombre aspecto
     funcional>.begin ();
            }
       End_Initialize;

       Destruction
           destroy ()
           {  <Hueco nombre aspecto funcional>.end
     ();
            }
       End_Destruction;

     End_Connector_Type  <Hueco nombre componente>;
```

1.10.- Esqueleto del Componente Usuario

```
     Component_Type <Hueco nombre componente>
       Ports
         OperatorPort: IOperatorDEDT
         Played_Role  <Hueco nombre aspecto
     funcional>.OPERATOR
       End_Ports;

       Functional Aspect import  <Hueco nombre
     aspecto funcional>;
```

```
  Initialize
     new ( )
     {  <Hueco nombre aspecto funcional>.begin
();
      }
  End_Initialize;

  Destruction
     destroy ()
     {  <Hueco nombre aspecto funcional>.end
();
      }
  End_Destruction;

End_Connector_Type  <Hueco nombre componente>;
```

1.11.- Esqueleto del Conector Coordinador

```
Connector_Type <Hueco nombre conector>
  Ports
    InferencePortCnct: IInferenceDEDT,
    Played_Role  <Hueco nombre aspecto de
coordinación>.INFERENCE;
    KnowledgePortCnct: IDomainDEDT,
    Played_Role  <Hueco nombre aspecto de
coordinación>.KNOWLEDGE;
    OperatorPortCnct: IOperatorDEDT,
    Played_Role  <Hueco nombre aspecto de
coordinación>.OPERATOR;
  End_Ports;

  Coordination Aspect import  <Hueco nombre
aspecto de coordinación>;

  Initialize
     new ( )
     {  <Hueco nombre aspecto de
coordinación>.begin ();
     }
     End_Initialize;

  Destruction
     destroy ()
     {  <Hueco nombre aspecto de
coordinación>.end ();
     }
  End_Destruction;

End_Connector_Type  <Hueco nombre conector>;
```

2.- Tipos

2.1 Tipo de la Interface IDomainDEDT

```
Interface  IDomainDEDT
  limpiarBD ()
  inferirValorPropiedadesN1 (VALOR_PROPIEDADNO,
VALOR_PROPIEDADNI)
  inferirValorDiagnóstico (VALOR_PROPIEDADN1,
VALOR_DIAGNOSTICO)
End_Interface  IDomainDEDT
```

2.2 Tipo de la Interface IInferenceDEDT

```
Interface IInferenceDEDT
     limpiarBD (..);
     obtenerDiagnostico ( output
        VALOR_DIAGNOSTICO: string)
     obtenerValorPropiedades0  (input PROPIEDADES0:
        string  list [1..*],
           output VALOR_PROPIEDADES0: string  list
        [1..*])
     inferirValorPropiedades1  (input
        VALOR_PROPIEDADES0: string  list [1..*],
        output VALOR_PROPIEDADES1: string  list
        [1..*])
     inferirValorDiagnostico (input
        VALOR_PROPIEDADES1: string,  list [1..*],
             output VALOR_ DIAGNOSTICO: string)
End_ Interface IInferenceDEDT
```

2.3 Tipo de la Interface IOperatorDEDT

```
Interface IOperatorDEDT
     obtenerDiagnostico ( output
       VALOR_DIAGNOSTICO: string)
     obtenerValorPropiedades0  (input PROPIEDADES0:
       string  list [1..*],
          output VALOR_PROPIEDADES0: string  list
       [1..*])
End_ Interface IOperatorDEDT
```

2.4 Tipo del Aspecto Funcional del Componente Base de Conocimientos DT

```
   Functional  Aspect  FBaseDT  using  IDomainDEDT
     Attributes
       Variables
       imagenes: string,
       estructura televisiva: string,
```

```
  recursos televisivos: string,
  musica: string,
  tratamiento: string,
  informacion:string,
  objetivo: string,
  audio: string,
  video: string,
  cinta: string,

  Deriveds
  produccion: string,
  contenido: string,
  calidad tecnica: string,
  estado del video: string;

Derivations
{ (imágenes = ”adecuadas” and estructura
televisiva= “autorizada” and recursos
televisivos= “buenos” and musica= “buena”) }
produccion:= “buena”

{ (imágenes = ”adecuadas” and estructura
televisiva= “diferente” and recursos
televisivos= “malos” and musica= “mala”) }
produccion:= “mala”

{ (tratamiento= “didactico” and información=
“correcta” and objetivo= ”si lo cumple”) }
contenido:= “bueno”

{ (tratamiento= “confuso” and información=
“incorrecta” and objetivo= ”no lo cumple”) }
contenido:= “malo”

{ (audio= “sucio” and video= “sucio” and
cinta= ”deteriorada”) } calidad tecnica:=
“mala”

{ (produccion:= “buena” and contenido:=
“bueno” and calidad tecnica:= “buena” ) }
estado del video:= “video OK”

{ (produccion:= “mala” and contenido:= “malo”
and calidad tecnica:= “mala” ) } estado del
video:= “video MALO”

Services
begin ()

in limpiarBD (..);
  Valuations
     [in limpiarBD ( )]
          imagenes:= nil,
          estructura televisiva:= nil,
          recursos televisivos:= nil,
```

```
          musica:= nil,
          tratamiento:= nil,
          informacion:= nil,
          objetivo:= nil,
          audio:= nil,
          video:= nil,
          cinta:= nil,
          produccion:= nil,
          contenido:= nil,
          calidad tecnica:= nil,
          decision de transmitir:= nil;

in  inferirValorPropiedades1  (input
   VALOR_PROPIEDADES0: string   list [1..*],
   output VALOR_PROPIEDADES1: string  list
   [1..*])
  Valuations
   [in  inferirValorPropiedades1
   (VALOR_PROPIEDADES0, VALOR_PROPIEDADES1) ]
          imagenes:= VALOR_PROPIEDADES0 [1],
          estructura televisiva:=
          VALOR_PROPIEDADES0 [2],
          recursos televisivos:=
          VALOR_PROPIEDADES0 [3],
          musica:= VALOR_PROPIEDADES0 [4],
          tratamiento:= VALOR_PROPIEDADES0
          [5],
          información:= VALOR_PROPIEDADES0
          [6],
          objetivo:= VALOR_PROPIEDADES0 [7],
          audio:= VALOR_PROPIEDADES0 [8],
          video:= VALOR_PROPIEDADES0 [9],
          cinta:= VALOR_PROPIEDADES0 [10],
          VALOR_PROPIEDADES1 [1]:= producción,
          VALOR_PROPIEDADES1 [2]:= contenido,
          VALOR_PROPIEDADES1   [3]:=   calidad
          tecnica;

in  inferirValorDiagnostico (input
   VALOR_PROPIEDADES1: string  list [1..*],
        output VALOR_DIAGNOSTICO: string )
  Valuations
   [in  inferirValorDiagnostico
   (VALOR_PROPIEDADES1, VALOR_DIAGNOSTICO) ]
          producción:= VALOR_PROPIEDADES1 [1],
          contenido:= VALOR_PROPIEDADES1 [2],
          calidad tecnica:= VALOR_PROPIEDADES1
          [3],
          VALOR_DIAGNOSTICO:= estado del
          video;

end;

Played_Roles
   KNOWLEDGE=
```

```
            IDomainDEDT.LimpiarDB ? ( )
            →
            IDomaiDEnDT.InferirValorPropiedades0 ?
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            IDomainDEDT.InferirValorPropiedades0 !
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            IDomainDEDT.InferirValorDiagnostico ?
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            IDomainDEDT.InferirValorDiagnostico !
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO).

      Protocols
         FBASE = begin ( )..P0

         P0 =
               KNOWLEDGE.LimpiarBD ? (  ). P1

         P1 =
            ( KNOWLEDGE.InferirValorPropiedades0 ?
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            KNOWLEDGE.InferirValorPropiedades0 ?
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)  ).P2

         P2 =
            ( KNOWLEDGE.InferirValorDiagnostico ?
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            KNOWLEDGE.InferirValorDiagnostico !
            (VALOR_PROPIEDADES1,
            AVALOR_DIAGNOSTICO)  ).FIN

         FIN = end;

   End_Functional Aspect FBaseDT
```

2.5 Tipo del Aspecto Funcional del Componente Motor de Inferencia DT

```
   Functional  Aspect  FMotorDT  using
   IInferenceDEDT
     Attributes
```

```
Variables
propiedades0: string    list [1..*]
valorPropiedades0: string    list [1..*]
valorPropiedades1: string    list [1..*]
valorDiagnostico: string;

Services
begin  ()

out  limpiarBD (..);

in  obtenerDiagnostico ( output
   VALOR_DIAGNOSTICO: string)
  Valuations
    [in  obtenerDiagnostico (VALOR_DIAGNOSTICO)
    ]
          VALOR_DIAGNOSTICO:=valorDiagnostico,

out  obtenerValorPropiedades0  (input
   PROPIEDADES0: string  list [1..*],
      output VALOR_PROPIEDADES0: string  list
   [1..*])
  Valuations
    [out  obtenerValorPropiedades0
    (propiedades0, VALOR_PROPIEDADES0) ]
          valorPropiedades0:=
          VALOR_PROPIEDADES0,

out  inferirValorPropiedades1  (input
   VALOR_PROPIEDADES0: string  list [1..*],
   output VALOR_PROPIEDADES1: string  list
   [1..*])
  Valuations
    [out  inferirValorPropiedades1
    (valorPropiedades0, VALOR_PROPIEDADES1) ]
          valorPropiedades1:=
          VALOR_PROPIEDADES1;

out  inferirValorDiagnostico (input
   VALOR_PROPIEDADES1: string,  list [1..*],
   output VALOR_ DIAGNOSTICO: string)
  Valuations
    [out inferirValorDiagnostico
    (valorPropiedades1, VALOR_DIAGNOSTICO)]
          valorDiagnostico:=
          VALOR_DIAGNOSTICO,

end;

Played_Roles
   INFERENCE=
        IInferenceDEDT.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.LimpiarBD ! ( )
```

```
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0  !( PROPIEDADES0,
        VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0 ? (PROPIEDADES0, VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.InferirValorPropiedades
        1  ! (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorPropiedades
        1 ? (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ! (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ? (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.ObtenerDiagnostico !
        (VALOR_DIAGNOSTICO).

Protocols
   FMOTOR = begin ( ).P0

   P0 =
        INFERENCE.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO).P1

   P1 =
        INFERENCE.LimpiarBD ! ( ).P2

   P2 =
        ( INFERENCE.ObtenerValorPropiedades0
        !( PROPIEDADES0, VALOR_PROPIEDADES0)
        →
        INFERENCE ObtenerValorPropiedades0 ?
        (PROPIEDADES0, VALOR_PROPIEDADES0)
        ).P3

   P3 =
        (INFERENCE.InferirValorPropiedades1  !
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        INFERENCE.InferirValorPropiedades1 ?
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)  ).P4
```

```
        P4 =
             (INFERENCE.InferirValorDiagnostico !
             (VALOR_PROPIEDADES1,
             VALOR_DIAGNOSTICO)
             →
             INFERENCE.InferirValorDiagnostico ?
             (VALOR_PROPIEDADES1,
             VALOR_DIAGNOSTICO) ).P5

        P5 =
             INFERENCE.ObtenerDiagnostico !
             (VALOR_DIAGNOSTICO).FIN

        FIN = end;

    End_Functional Aspect FMotorDT
```

2.6.- Tipo del Aspecto Funcional del Componente Usuario DT

```
Functional Aspect FUserDT  using IOperatorDT
    Attributes
      Variables
      propiedades0: string  list [1..*]
      valorPropiedades0: string    list [1..*]
      valorDiagnostico: string;

    Services
      begin  ()

      out  obtenerDiagnostico ( output
         VALOR_DIAGNOSTICO: string)
        Valuations
          [out  obtenerValorDiagnostico
          (VALOR_DIAGNOSTICO) ]
                valorDiagnostico:=
                VALOR_DIAGNOSTICO,

      in  obtenerValorPropiedades0  (input
         PROPIEDADES0: string  list [1..*],
            output VALOR_PROPIEDADES0: string  list
         [1..*])
        Valuations
          [in  obtenerValorPropiedades0
          (PROPIEDADES0, VALOR_PROPIEDADES0) ]
                propiedades0:= PROPIEDADES0,
                VALOR_PROPIEDADES0:=
                valorPropiedades0;

      end;

      Played_Roles
         OPERATOR =
              IOperatorDEDT ObtenerDiagnostico !
              (VALOR_DIAGNOSTICO)
```

```
            →
            IOperatorDEDT.ObtenerValorPropiedades0
            ? (PROPIEDADES0, VALOR_PROPIEDADES0)
            →
            IOperatorDEDT ObtenerValorPropiedades0
            ! (PROPIEDADES0, VALOR_PROPIEDADES0)
            →
            IOperatorDEDT ObtenerDiagnostico ?
            (VALOR_DIAGNOSTICO)

    Protocols
       FUSER = begin ( ).P0

       P0 =
             OPERATOR.ObtenerDiagnostico !
            (VALOR_DIAGNOSTICO).P1

       P1 =
            (OPERATOR.ObtenerValorPropiedades0 ?
            (PROPIEDADES0, VALOR_PROPIEDADES0)
            →
            OPERATOR.ObtenerValorSubpropiedades0 !
            (PROPIEDADES0, VALOR_PROPIEDADES0) ).
            P2

       P2 =
            OPERATOR.ObtenerDiagnostico ?
            (VALOR_DIAGNOSTICO).FIN

       FIN = end;

End_Functional Aspect FUserDT
```

2.7.- Tipo del Aspecto de Coordinación del Conector Coordinador DT

```
Coordination Aspect CDiagDT using
IInferenceDEDT, IDomainDEDT, IOperatorDEDT

  Services
  begin  ( ),

  in/out  limpiarBD (..);

  in/out  obtenerDiagnostico ( output
     VALOR_DIAGNOSTICO: string)

  in/out  obtenerValorPropiedades0  (input
     PROPIEDADES0: string,  list [1..*],
     output VALOR_PROPIEDADES0: string  list
     [1..*])
```

```
in/out  inferirValorPropiedades1  (input
   VALOR_PROPIEDADES0: string  list [1..*],
   output VALOR_PROPIEDADES1: string  list
   [1..*])

in/out  inferirValorDiagnostico (input
   VALOR_PROPIEDADES1: string,  list [1..*],
   output VALOR_ DIAGNOSTICO: string)

end;

Played_Roles
   INFERENCE=
        IInferenceDEDT.ObtenerDiagnostico ?
        (VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.LimpiarBD ! ( )
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0  !( PROPIEDADES0,
        VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.ObtenerValorPropiedades
        0 ? (PROPIEDADES0, VALOR_PROPIEDADES0)
        →
        IInferenceDEDT.InferirValorPropiedades
        1  ! (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorPropiedades
        1 ? (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ! (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.InferirValorDiagnostico
        ? (VALOR_PROPIEDADES1,
        VALOR_DIAGNOSTICO)
        →
        IInferenceDEDT.ObtenerDiagnostico !
        (VALOR_DIAGNOSTICO).

   KNOWLEDGE=
        IDomainDEDT.LimpiarDB ? ( )
        →
        IDomainDEDT.InferirValorPropiedades0 ?
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
        →
        IDomainDEDT.InferirValorPropiedades0 !
        (VALOR_PROPIEDADES0,
        VALOR_PROPIEDADES1)
```

```
          →
          IDomainDEDT.InferirValorDiagnostico ?
          (VALOR_PROPIEDADES1,
          VALOR_DIAGNOSTICO)
          →
          IDomainDEDT.InferirValorDiagnostico !
          (VALOR_PROPIEDADES1,
          VALOR_DIAGNOSTICO).

    OPERATOR =
          IOperatorDEDT .ObtenerDiagnostico !
          (VALOR_DIAGNOSTICO)
          →
          IOperatorDEDT
          .ObtenerValorPropiedades0 ?
          (PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          IOperatorDEDT
          .ObtenerValorPropiedades0 !
          (PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          IOperatorDEDT .ObtenerDiagnostico ?
          (VALOR_DIAGNOSTICO)

Protocols
    FMOTOR = begin ( ).P0

    P0 = (
          OPERATOR.ObtenerDiagnostico ?
          (VALOR_DIAGNOSTICO)
          →
          INFERENCE.ObtenerDiagnostico !
          (VALOR_DIAGNOSTICO)
          →
          INFERENCE.LimpiarBD ? ( )
          →
          KNOWLEDGE.LimpiarBD ! ( )
          →
          INFERENCE.ObtenerValorPropiedades0 ? (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          OPERATOR.ObtenerValorPropiedades0 ! (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          OPERATOR.ObtenerValorPropiedades0 ? (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          INFERENCE.ObtenerValorPropiedades0 ! (
          PROPIEDADES0, VALOR_PROPIEDADES0)
          →
          INFERENCE.InferirValorPropiedades1  ?
          (VALOR_PROPIEDADES0,
          VALOR_PROPIEDADES1)
          →
```

```
            KNOWLEDGE.InferirValorPropiedades1 !
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            KNOWLEDGE.InferirValorPropiedades1 ?
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            INFERENCE.InferirValorPropiedades1 !
            (VALOR_PROPIEDADES0,
            VALOR_PROPIEDADES1)
            →
            INFERENCE.InferirValorDiagnostico ?
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            KNOWLEDGE.InferirValorDiagnostico !
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            KNOWLEDGE.InferirValorDiagnostico ?
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            INFERENCE.InferirValorDiagnostico !
            (VALOR_PROPIEDADES1,
            VALOR_DIAGNOSTICO)
            →
            INFERENCE.ObtenerDiagnostico ?
            (VALOR_DIAGNOSTICO)
            →
            OPERATOR.ObtenerDiagnostico !
            (VALOR_DIAGNOSTICO)
            ).FIN

        FIN = end;

End_Coordination Aspect CDiagDT
```

2.8.- Tipo del Componente Base de Conocimientos DT

```
Component_Type KnowledgeBaseDT
  Ports
    KnowledgePort: IDomainDEDT,
      Played_Role  FBaseDT.KNOWLEDGE;
  End_Ports;

  Function Aspect import  FBaseDT;

  Initialize
      new ( )
      {  FBaseDT.begin ();
```

```
        }
    End_Initialize;

    Destruction
        destroy ()
        {  FBaseDT.end();
         }
    End_Destruction;

End_Connector_Type  KnowledgeBaseDT;
```

2.9.- Tipo del Componente Motor de Inferencia DT

```
Component_Type InferenceMotorDT
   Ports
     InferencePort: IInferenceDEDT,
     Played_Role  FMotorDT.INFERENCE;
   End_Ports;

   Function Aspect import  FMotorDT;

   Initialize
       new ( )
       {  FMotorDT.begin ();
        }
   End_Initialize;

   Destruction
       destroy ()
       {  FMotorDT.end ();
        }
   End_Destruction;

End_Connector_Type  InferenceMotorDT;
```

2.10.- Tipo del Componente Usuario DT

```
Component_Type UserDT
  Ports
    OperatorPort: IOperatorDEDT
    Played_Role  FUserDT.OPERATOR
  End_Ports;

  Functional Aspect import  FUserDT;

  Initialize
     new ( )
     {  FUserDT.begin ();
      }
  End_Initialize;

  Destruction
     destroy ()
```

```
      {  FUserDT.end ();
       }
    End_Destruction;

  End_Connector_Type  UserDT;
```

2.11.- Tipo del Conector Coordinator DT

```
  Connector_Type CoordinatorDT
    Ports
      InferencePortCnct: IInferenceDEDT,
      Played_Role  CDiagDT.INFERENCE;
      KnowledgePortCnct: IDomainDEDT,
      Played_Role  CDiagDT.KNOWLEDGE;
      OperatorPortCnct: IOperatorDEDT,
      Played_Role  CDiagDT.OPERATOR;
    End_Ports;

    Coordination Aspect import  CDiagDT;

    Initialize
       new ( )
       {  CDiagDT.begin ();
       }
       End_Initialize;

    Destruction
       destroy ()
       {  CDiagDT.end ();
       }
    End_Destruction;

  End_Connector_Type  CoordinatorDT;
```

3- Modelo Arquitectónico del Sistema de Diagnóstico Televisivo

```
Architectural_Model  DiagTV

  Variables
    VarMotorInferencia: InferenceMotorDT;
    VarBaseConocimientos: KnowledgeBaseDT;
    VarUsuario: UserDT;
    VarConector: CoordinatorDT;
  End_Variables;

  Attachments
    VarConector.InferencePortCnct   ↔
  VarMotorInferencia.InferencePort;
```

```
    VarConector.KnowledgePortCnct   ↔
  VarBaseConocimientos.KnowledgePort;
    VarConector.OperatorPortCnct   ↔
  VarUsuario.OperatorPort;
  End_Attachments;

  Initialize
    new ( )
    {
     VarMotorInferencia= new InferenceMotorDT;
     VarBaseConocimientos= new KnowledgeBaseDT;
     VarUsuario= new UserDT;
     VarConector= new CoordinatorDT;
    }
  End_Initialize;

  Destruction
     destroy ()
     {
    VarMotorInferencia.destroy ( );
    VarBaseConocimientos.destroy ( );
    VarUsuario.destroy ( );
    VarConector.destroy ( );
     }
  End_Destruction;

End_Architectural_Model  DiagTV;
```