

UNIVERSIDAD POLITÉCNICA DE
VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y
COMPUTACIÓN

TESIS DOCTORAL



Las vistas arquitectónicas de software y sus correspondencias mediante la gestión de modelos.

Rogelio Noé Limón Cordero

Memoria para optar al grado de Doctor en
Informática
bajo la dirección de

Dr. Isidro Ramos Salavert

Valencia, España

CONTENIDO

Capítulo 1	1
Introducción	1
1.1 Planteamiento del problema.....	3
1.2 Objetivos de la tesis.....	6
1.3 Estructura de la tesis.....	7
 Capítulo 2	 9
La Arquitectura de Software y sus Asuntos de Interés (<i>concerns</i>).	9
2.1 La arquitectura de software en el desarrollo de sistemas de software.....	9
2.2 Los <i>concerns</i> arquitectónicos.....	16
2.3 La separación de concerns arquitectónicos.....	28
2.4 El modelo PRISMA en la separación de concerns	38
 Capítulo 3	 48
Modelos de vistas arquitectónicas de software existentes y la propuesta de su modelado	48

3.1	Precisando el concepto de vista arquitectónica	48
3.2	Los modelos propuestos de vistas en la arquitectura	55
3.3	Anomalías detectadas en los modelos de las vistas, y propuesta del marco de trabajo para el modelado de las vistas.....	67
3.4	Bases para la especificación de los modelos de las vistas-arquitectónicas.....	72
3.5	Especificación del modelo de la vista modular.....	75
3.6	Especificación del modelo de la vista componentes-y-conectores.	83
 Capítulo 4.....		104
Desarrollo Dirigido por Modelo en las Vistas Arquitectónicas		104
4.1	Desarrollo dirigido por modelos	105
4.2	Elementos de modelado en MDA	110
4.3	Transformación entre modelos	116
4.4	Propuesta de modelado de vistas arquitectónicas de software basándose en MDA.....	123
4.5	Propuesta del proceso para el modelado de las vistas mediante MDA.....	129
 Capítulo 5		135
Meta-modelos de las Vistas Arquitectónicas y sus Correspondencias.....		135

5.1 Diseñando el meta-modelo de la vista modular.	136
5.2 Diseñado el meta-modelo de la vista componentes-y-conectores	142
5.3 Meta-modelo para el comportamiento de componentes	152
5.4 Correspondencias entre la vista modular y la vista de componentes-y-conectores	157
5.5 Estableciendo relaciones entre interacciones y conectores	169
5.6 Generando modelos de la vista de componentes-y-conectores con modelos de la vista de modular.	171
Capítulo 6	180
Aplicando la Transformación de Modelos en las Vistas Arquitectónicas	180
6.1 Caso de estudio	180
6.2 Iniciando la arquitectura con el modelo de la vista modular	190
6.3 Obteniendo el modelo arquitectónico de la vista de componentes-conectores	200
6.4 Refinamiento de relaciones.	209
6.5 Gestión de los modelos de las vistas	214
6.6 Análisis de las relaciones en el modelo de la vista modular.	221
Capítulo 7	232
Conclusiones y trabajos futuros.	232

7.1 Conclusiones.....	232
7.2 Aportaciones	237
7.3 Publicaciones	239
7.4 Trabajos Futuros.....	240
 APENDICE A.....	 244
Implementación de los meta-modelos de las vistas	244
APENDICE B	251
Codificación de las relaciones de correspondencias en QVT-Relaciones.....	251
APENDICE C	257
Codificación de las transformaciones entre modelos de las vistas en QVT-Operacional	257
REFERENCIAS.....	263

LISTA DE FIGURAS

Figura 2.1. La arquitectura de software en el ciclo de vida del software del modelo de espiral de Boehm (Boehm, 1988)	11
Figura 2.2. Interacción de la AS con los actores involucrados en el ciclo de requerimientos. (Tomado de (McBride 2007)).	13
Figura 2.3. Modelo de la arquitectura para sistemas intensivos de acuerdo al estándar 1471 de la IEEE	17
Figura. 2.4 Elementos modulares con un concern que presentan crosscutting	30
Figura 2.6. Los aspectos en el modelo PRISMA, tomado de (Perez 2006)	39
Figura 2.7. Representación gráfica de una especificación de una arquitectura en PRISMA, adaptación de (Perez 2006)	40
Figura 2.8. Los cinco pasos en el modelado de una arquitectura con PRISMA, tomado de (Perez, 2006)	42
Figura 2.9. Actividades y artefactos de ATRIUM, tomado de (Navarro, 2007).....	44
Figura 2.10. Flujo de trabajo para especificar objetivos y requerimientos, tomada de (Navarro, 2007).....	45
Figura 3.1. Relación entre <i>vista</i> , punto-de-vista y <i>concerns</i> , tomado de (Kande y Strohmeier, 2000)	52
Figura. 3.2 El modelo de vistas 4+1, tomado de (Kruchten, 1995)....	58
Figura 3.3 Relación entre las cuatro vistas de Siemens, (adaptación hecha de (Hofmeister, et al.,2000, pag. 20))	63
Figura 3.4. Propuesta del modelado de las vistas y sus relaciones, usando el enfoque de DDM	71
Figura 3.5 Representación esquemática de una relación de usa-capacidad	79
Figura. 3.6 Representación esquemática de la relación de <i>descomposición</i>	80
Figura 3.7 Representación esquemática de la relación de filtro-tubería	96
Figura 3.9 Representación esquemática de la relación <i>igualdad-de-servicios</i>	98
Figura 3.10 Representación esquemática de la relación <i>escritor-lector</i>	99

Figura 3.11 Representación esquemática de la relación datos-compartidos	100
Figura 3.12 Representación esquemática de la relación comunicación-de-procesos	101
Figura 3.13 Representación en notación UML 2.0 de las relaciones: (a) Filtro-tubería, con dos componentes; (b) filtro-tubería, con un solo componente; (c) cliente-servidor combinado con la relación datos-compartidos; (d) escritor-lector	103
Figura 4.1. La correspondencia entre un modelo y un sistema, tomado de (Gašević et al., 2006)	110
Figura 4.2. Arquitectura de capas del enfoque DDM	113
Figura 4.3. Transformación de un PIM a un PSM, adaptación hecha de (Gašević et al., 2006)	116
Figura 4.4. Clasificación de tipos de transformación, tomada de (Metzger, 2005)	118
Figura 4.5. Estrategia para transformación de modelos de vistas arquitectónicas de software	126
Figura 4.6 Proceso propuesto para modelar vistas arquitectónicas mediante MDA	130
Figura 5.1 Meta-Modelo de la Vista Modular	139
Figura 5.2 Meta-modelo base de la vista de Componentes-y-Conectores	145
Figura 5.3 Meta-modelo vista C-C con estilo Filtro-Tubería	147
Figura 5.4 Meta-modelo de la relación Cliente-Servidor	148
Figura 5.5 Meta-modelo de la relación igualdad-de-servicios	149
Figura 5.6 Meta-modelo de la relación escritor-lector	150
Figura 5.7 Meta-modelo de la relación acceso-a-datos	151
Figura 5.8 Meta-modelo de la relación comunicación-de-procesos	152
Figura 5.9 Ejemplo de un modelo de interacciones entre componentes	154
Figura 5.10 Meta-modelo de interacción	155
Figura 5.9 Especificación de la relación moduloAcomponente	161
Figura 5.10 Especificación de la relación funcionApropiedades	163
Figura 5.11 Especificación de la relación ResponsabilidadesAservicios	163

Figura 5.12 Especificación de la relación subsistemaAcompoCom .	165
Figura 5.13 Especificación de la relación capaAnivel	166
Figura 5.14 Especificación de la relación usoAconector.....	167
Figura 5.15 Especificación de la relación usoSubsistemaAconector	168
Figura 5.16 Especificación de la relación <i>generalizaAConector</i> . (b) functionToservice, (c) rUseModToConnector, (d) rCompositionModToComp.....	169
Figura 5.17 Especificación de la relación iteraccionAconector , (b) functionToservice, (c) rUseModToConnector, (d) rCompositionModToComp.....	170
Figura 5.18 Transformación de un modelo de vista modular con relación de generalización a un modelo componentes-y-conectores	174
Figura 5.19 Transformación de un modelo de vista modular a un modelo de vista de componentes-y-conectores con refinamiento .	178
Figura 5.19 (continuación) Transformación de un modelo de vista modular a un modelo de vista de componentes-y-conectores con refinamiento	179
Figura 6.1 Trabajo colaborativo para telemedicina, tomado de (Stevenson et al., 2008).....	183
Figura 6.2 Ciclo de la arquitectura aplicado al sistema de la red de servicios médicos públicos	188
Figura 6.3 Método para la especificación del modelo de la vista modular	191
Figura 6.4 (a) Requisitos funcionales de un módulo (b) Relación de composición. (c) Reestructurando los módulos con un requisito de calidad.	192
Figura 6.5 Primer arquitectura de la vista modular del sistema red- atención-médica	194
Figura 6.6. Descomposición del módulo <i>AsistenciaMedicaRemota</i> y las relaciones identificadas entre los módulos derivados	195
Figura 6.7 Descomposición del módulo <i>GestionConsultas</i> con sus relaciones identificadas.....	197
Figura 6.8 Relaciones de uso entre los módulos LocalizarCentroAtencion y SuministroMedicaMaterial con FormarRutas	198

Figura 6.9 Modelo final de la vista de la arquitectura modular para el sistema -red de atención médica-	199
Figura 6.10 Primera fase de transformación entre el modelo del vista modular y el de la vista componentes-y-conectores	202
Figura 6.11 Transformando una relación <i>usa</i> del modelo de la vista modular al modelo de la vista componentes-y-conectores.....	205
Figura 6.12 Transformación de una relación generalización del modelo de la vista modular al modelo de la vista componentes-y-conectores	207
Figura 6.13. Modelo de la vista de componentes y conectores resultante de la transformación.....	208
Figura 6.14 Modelo de interacciones entre componentes	210
Figura 6.15 Refinamiento del modelo de componentes-y-conectores, usando la iteración <i>GestionHistorial</i>	212
Figura 6.16 Resultado de aplicar una comparación entre dos versiones del modelo de la vista modular: (a) detectando un elemento nuevo , y (b) detectando un cambio en una propiedad.	216
Figura 6.17. Transformaciones requeridas por cambios en el modelo (a) de interacciones, (b) de la vista modular , y (c) de la vista de componentes –y-conectores	217
Figura 6.18 Transformación con un cambio en el modelo de interacción	218
Figura 6.19 Transformación con un cambio en el modelo de la vista modular, parcialmente.	219
Figura 6.20. Matriz de relaciones básica entre dos elementos	222
Figura 6.21. Tres casos de relaciones con $R=2$	223
Figura 6.22. Complemento de los casos para $R=3$	224
Figura 6.23. Relación de descomposición: (a) en forma general, (b) un ejemplo que incluye una relación, (c) ocultando los elementos. ..	225
Figura 6.24. Representación matricial de la relación <i>usa</i> , (a) en forma básica, (b) combinándola con una relación de descomposición	226
Figura 6.25 Representación matricial de la relación de capa	226
Figura 6.26 Representación matricial del modelo de la vista modular del sistema red-médica	227

LISTA DE TABLAS

Tabla 2.1. Diversas taxonomías sobre los atributos de calidad que debe cumplir el software	23
Tabla 3.1 Notación en UML 2.0, de los tipos de elementos de la vista de módulo.	81
Tabla 3.2 Notación en UML 2.0, de las relaciones entre elementos de la vista de módulo.	82
Tabla 3.3 Tipos de componentes, sus funciones computacionales y atributos adicionales.....	88
Tabla 3.4 Tipos de conectores con sus componentes asociados, sus roles y atributos.....	94
Tabla 3.5 Notación en UML 2.0, de los elementos de la vista de componente-conector.	102
Tabla 5.1 Concerns considerados en las vistas V_m y V_{cc} , de acuerdo a los criterios empleados	159
Tabla 5.2 Relaciones identificadas en la correspondencia entre V_m y V_{cc} base.	160
Tabla 6.1 Indicadores para medir el nivel de dependencia	228

RESUMEN

La arquitectura de software es un área relativamente joven de la ingeniería de software. Iniciada a mediados de los años 90's, ha llegado a consolidarse actualmente como una pieza fundamental para el desarrollo de grandes y complejos sistemas de software, debido a que permite tomar decisiones tempranas que son trascendentes en las siguientes etapas del desarrollo de estos sistemas.

Las vistas arquitectónicas constituyen las estructuras fundamentales de dicha arquitectura, y hacen posible enfrentar su diseño desde diversas perspectivas reduciendo la complejidad presente en su desarrollo. El diseño de las vistas arquitectónicas implica establecer una sincronización entre sus elementos para conformar una arquitectura integrada, y de poder ser capaz de mantener la consistencia entre sus elementos.

Por otro lado, la arquitectura dirigida por modelos (MDA), ofrece la posibilidad de crear modelos independientes a las plataformas tecnológicas, y a la vez establece vínculos entre ellos que logran una trazabilidad entre sus elementos que los integran.

El trabajo que aquí se presenta va en dos sentidos, uno es la creación de un marco de trabajo para el diseño de vistas arquitectónicas que respondan a los requerimientos de modelado de los sistemas a diseñar, y por el otro lado en gestionar estas vistas mediante una estrategia que potencialice las ventajas del MDA.

Capítulo 1

Introducción

La arquitectura de software ha llegado a constituirse en una actividad vital para el desarrollo de software a gran escala ya que es usada como referencia en la fase del diseño de un sistema de software.

Dentro de una arquitectura de software se conjunta a los requisitos funcionales y a los requisitos de calidad, ya que en su modelado se toman en cuenta los intereses del espacio del problema y algunos de los intereses del espacio de la solución.

Los intereses de dichos espacios se modelan mediante diversos artefactos que toman en cuenta los diferentes aspectos arquitectónicos. Estos artefactos son representados mediante las vistas arquitectónicas usando meta-modelos que establecen la notación, el lenguaje y las reglas para especificarlas.

Cada vista agrupa a elementos que pertenecen a diferentes intereses, no obstante ello, se llegan a establecer relaciones de correspondencias entre los elementos de distintas vistas puesto que al formar parte de un mismo sistema de alguna manera se llegan a relacionar. Estas relaciones de correspondencia constituyen el adhesivo que mantiene enlace a las vistas que integran a la arquitectura del software.

Esta tesis está enfocada a identificar, mantener y gestionar las relaciones de correspondencia usando la estrategia que presenta la ingeniería dirigida por modelos, específicamente

la presentada por la arquitectura dirigida por modelos (MDA¹), la cual permite relacionar modelos (a nivel de meta-modelos), y basándose en estas relaciones es capaz de generar modelos a partir de otros modelos, ya sea del mismo nivel de abstracción o de niveles más concretos mediante técnicas de transformación.

La estrategia del MDA es aplicada para llegar a relacionar los meta-modelos de las vistas arquitectónicas. Por medio de la transformación entre los modelos de éstas se hace posible mantener su consistencia entre vistas distintas, e incluso llegar a generar una vista a partir de otra.

Se parte del análisis sobre el papel que desempeñan las vistas en la separación de los diversos asuntos de interés arquitectónicos, y la manera en que las propuestas existentes llegan a usar a las vistas para este propósito. Cada propuesta analizada sugiere meta-modelos que representan a los asuntos de interés que deben ser incluidos en las vistas. Entre todos estos meta-modelos se elige el que mejor representa a los asuntos de interés de las diversas estructuras arquitectónicas.

Para poder llegar a formar los meta-modelo que le corresponde a cada una las vistas, se analizan las relaciones que se forman entre los elementos de cada una de ellas, de este análisis se deriva también la manera de incluir en estos meta-modelos a los diferentes estilos arquitectónicos que se detectan en cada vista.

Para el diseño de los meta-modelos de cada vista se usa la propuesta de OMG que se tiene para el MDA, porque con ella se obtiene el soporte necesario tanto para el modelado como para la expresividad. Esta misma propuesta también

¹ MDA siglas en Inglés de Model Driven Architecture

es utilizada para modelar las relaciones entre estos meta-modelos y para llevar a cabo la transformación entre los modelos de las vistas, estableciendo un marco de trabajo congruente entre estas tareas.

1.1 Planteamiento del problema

A medida que la Arquitectura de Software se ha consolidado como un área de conocimiento dentro de la Ingeniería de Software, se han abierto más posibilidades de su aplicación, pero al mismo tiempo esto ha implicado abordar varios retos que se han presentado en este ámbito. En esta tesis los retos que se abordan están dirigidos hacia el manejo de las relaciones entre cada modelo que forma a una arquitectura, y de las relaciones que entre ellos se establecen.

Se abarca desde de la determinación de las estructuras arquitectónicas que intervienen en el desarrollo de un sistema de software, hasta establecer y mantener los vínculos que se forman entre estas estructuras. A ello se agrega el reto de realizar todo ello a través de la arquitectura dirigida por modelos (MDA).

La necesidad de contar con varias estructuras arquitectónicas parte del hecho de que en el modelado de la arquitectura de un sistema intervienen varios tipos de actores (ingenieros de requisitos, diseñadores y los arquitectos), los cuales tienen diversos intereses que deben ser modelados en forma separada (no implicando que sean totalmente independientes). Esto trae como consecuencia que los asuntos de interés sean identificados y separados para formar varios modelos.

La identificación y separación de estos intereses impone el primer problema, que consiste en determinar cuáles son los

criterios a tomarse en cuenta para realizar esta separación. Durante sus primeros años de incursión de la arquitectura de software el criterio que imperó fue el de considerar sólo a elementos que intervienen al tiempo de ejecución de un sistema de software (componentes), ignorando a otros elementos como por ejemplo aquéllos que intervienen en el diseño y en la asignación de recursos (los cuales son también determinantes en el desarrollo y funcionamiento del sistema), dando por resultado que únicamente se considerara a un sólo modelo, conocido como de componentes y conectores.

Sin embargo, dado que estos intereses arquitectónicos surgen de dos ambientes diferentes (del dominio del problema y del dominio de la solución), se debe considerar más de un criterio para su separación y por lo tanto se debe contar con más de un modelo.

Se han hecho varias propuestas de los modelos a considerar, haciendo agrupaciones en vistas y en estilos arquitectónicos. Es aquí donde surgen otros problemas, ¿cuáles modelos deben ser considerados como importantes?, ¿Cuáles elementos deben considerarse en cada modelo y cuáles son las relaciones que entre ellos se establecen? También hacia estos problemas se dirige este trabajo.

Por otra parte, el contar con varios modelos se corre el riesgo de realizar su modelado en forma separada, provocando que no se establezcan nexos que debe haber entre una vista y otra. Los nexos se forman por varios motivos, ya sea porque cubren etapas consecutivas del desarrollo del sistema, o porque comprenden elementos semejantes a diferentes niveles de abstracción o simplemente porque cubren aspectos que están interrelacionados.

Cualquiera que sea el caso, se debe tener presente la correspondencia entre una y otra vista por el sólo hecho de que son parte de un mismo sistema. Dicha correspondencia hace que los elementos de una vista se relacionen con los de otra de alguna manera.

Las relaciones se pueden establecer ya sea en forma directa o a través de otro modelo como es el caso de la propuesta de Kruchten (Krutchten 1995), donde la vista de escenarios se usa para vincular a las otras cuatro vistas de su propuesta. Sin embargo, cuando estos vínculos no se han establecido en forma explícita es difícil mantener la consistencia, puesto que ocasiona esfuerzo adicional para realizar los cambios pertinentes al no mantener la traza entre los elementos de una vista con otra. La arquitectura se vuelve inconsistente cuando una vista se llega a modificar (por razones de mantenimiento o evolución), y esos cambios no se reflejan en las demás vistas relacionadas.

Los problemas de inconsistencia son debidos a la ausencia de mecanismos efectivos para afrontar los cambios entre las vistas, y por no conservar en forma persistente las relaciones que se establecen entre ellas a lo largo del modelado de la arquitectura de software.

Otro problema que se ha detectado por no contar con una sistematización de las relaciones entre las vistas, es el aumento de esfuerzo al diseñar una vista cuando ésta tiene relación de dependencia con otra, es decir, cuando los elementos a diseñar en una vista requieren de los elementos de otra. El aumento de esfuerzo se manifiesta cuando se realiza una verificación de qué elementos diseñados existen en la vista para que puedan ser usados por otra vista que los requiera. También ese esfuerzo se puede dar por la incorporación de nuevos elementos en la vista que se está diseñando y que no hayan sido considerados en ella,

teniendo que realizar trabajos de re-diseño en la segunda vista. Un ejemplo de relación de dependencia entre vistas se tiene en el modelo de *Kruchten*, donde una de sus vistas (la de procesos) usa elementos de otra vista (la lógica).

La solución de estos problemas mediante la gestión de modelos permite un diseño de las estructuras arquitectónicas en forma fiables y con capacidad para responder a los cambios impuestos por la evolución del sistema.

1.2 Objetivos de la tesis

- Establecer un marco de trabajo para diseñar y analizar arquitecturas de software, tomando como base las relaciones que se establecen entre los diferentes elementos de cada vista arquitectónica.
- Analizar y establecer relaciones de correspondencias que pueden existir entre los modelos de las vistas arquitectónicas.
- Vincular las vistas arquitectónicas mediante los modelos que las conforman a través del meta-modelado usando la propuesta de OMG de modelado dirigido por la arquitectura.
- Aplicar la estrategia de la transformación de modelos para generar el modelo de una vista arquitectónica tomando otro como referencia.
- Aplicar un análisis de dependencias de elementos en los modelos diseñados de las vistas, y en los modelos generados en una transformación.

- Mantener la consistencia entre las vistas cuando alguna de ellas cambie por razones de mantenimiento y evolución mediante la gestión de modelos.

1.3 Estructura de la tesis

Esta memoria de tesis se encuentra organizada de la siguiente manera:

El **capítulo 2** presenta a la arquitectura de software, que es la base que sobre la cual se sustentan las vistas arquitectónicas. Primero se ubica en el contexto del proceso de desarrollo de un sistema de software, destacando el papel de ésta dentro de dicho proceso. En seguida se presentan los asuntos de interés (llamados *concerns*) que se relacionan con ella, identificando los atributos de calidad que estos concerns involucran. Luego se analiza los problemas que se presentan para identificar y separar los concerns arquitectónicos. Concluyendo el capítulo con la exposición de cómo las propuestas existentes han enfrentado esos problemas.

El **capítulo 3** está dedicado a las vistas arquitectónicas. Se analizan las diferentes propuestas desarrolladas para su modelado. Como resultado de este análisis se realiza una propuesta para llevar a cabo el modelado de las vistas que cubren a los modelos expuestos. Se hace su especificación formal y se plantea la notación usada para los elementos y relaciones de las dos vistas que serán las empleadas en el resto de este trabajo.

El **capítulo 4** establece cómo vincular las vistas arquitectónicas mediante una estrategia basada en la arquitectura dirigida por modelos (MDA). Se inicia por

ubicar al MDA dentro del contexto de la arquitectura de software, y se elabora una propuesta de cómo aplicarla para vincular a dos vistas arquitectónicas. Especificando el proceso a seguir y las tareas que implica el proceso.

El **capítulo 5** constituye la médula de este trabajo, porque en él se incluye todo lo necesario para llevar a cabo el proceso de modelado de las vistas y de su gestión usando MDA. Ahí se presentan los meta-modelos, la relaciones y transformaciones que hacen posible el modelado y gestión de las vistas arquitectónicas.

En el **capítulo 6** se prueba la propuesta de este trabajo mediante su aplicación en un caso de estudio. Además se incorporan elementos que permiten analizar algunos resultados del proceso propuesto.

Por último, en el **capítulo 7** se presentan las conclusiones, aportes y trabajos futuros a desarrollar derivados de esta investigación.

Capítulo 2

La Arquitectura de Software y sus Asuntos de Interés (*concerns*).

Conforme los sistemas de software han sido más complejos la necesidad de anticiparse a la especificación detallada del sistema y de tener un mayor aprovechamiento de los recursos se hace más evidente, estos aspectos entre otros son capturados por la arquitectura de software (AS).

El análisis de las estructuras (vistas) que aquí se hace contribuye a mejorar esas decisiones tempranas del diseño, lo cual se materializa en la gestión de los modelos de las estructuras arquitectónicas.

En este capítulo se comienza por situar a la AS en el proceso del desarrollo de un sistema, destacando las contribuciones más importantes y las líneas de investigación que están en fase de maduración. Luego se analizan los asuntos de interés (*concerns*) que intervienen en la arquitectura, los problemas que estos presentan y cómo esto ha sido abordado a la fecha. Se termina con el análisis de modelo *PRISMA* en cuanto la contribución que este modelo hace en el tratamiento de los concerns arquitectónicos.

2.1 La arquitectura de software en el desarrollo de sistemas de software

La AS está ubicada en una fase previa al diseño, debido a que sus tareas son el fundamento del diseño de un sistema, como declara Booch (Booch 2007):

“... una implementación de un sistema es la manifestación de su arquitectura y una arquitectura de un sistema modela su implementación”.

Esto remarca la dualidad de la AS dentro de un sistema de software, porque por un lado la arquitectura es el referente para diseñarlo, pero a la vez sirve como el instrumento de modelado del sistema.

En (Bass et al 2003) se señala cómo la AS interviene en las decisiones tempranas del desarrollo de un sistema destacando la importancia de sus tareas antes del diseño, entre las tareas que se mencionan están: i) el establecimiento de las restricciones de los elementos que serán construidos en el diseño; ii) la determinación de la estructura del sistema y de las estructuras de las organizaciones que se requieren para darle soporte a la misma AS; iii) las tareas que posibilitan la adecuada aplicación de los atributos de calidad que mejor respondan a los requerimientos; iv) el usar a la AS como un medio de evaluación para hacer predicciones de calidad acerca del sistema; v) tareas para identificar y evaluar el nivel del impacto de los posibles cambios que ocurran en el ciclo de vida del sistema; vi) el uso de la AS como un prototipo para que a partir de ella se refine el diseño o bien hacer pruebas para analizar el comportamiento del sistema previo a su diseño. Como se apunta, la AS juega el rol de la planificación y guía de un sistema de software.

Para ubicar a estas tareas dentro del proceso de desarrollo de software se usa al modelo de espiral propuesto por *Boehm* (Boehm, 1988), el cual es ampliamente utilizado como base en diferentes metodologías de desarrollo. A través de él se describen las fases y las tareas que le corresponden a la arquitectura de software, como se ilustra en la Figura 2.1.

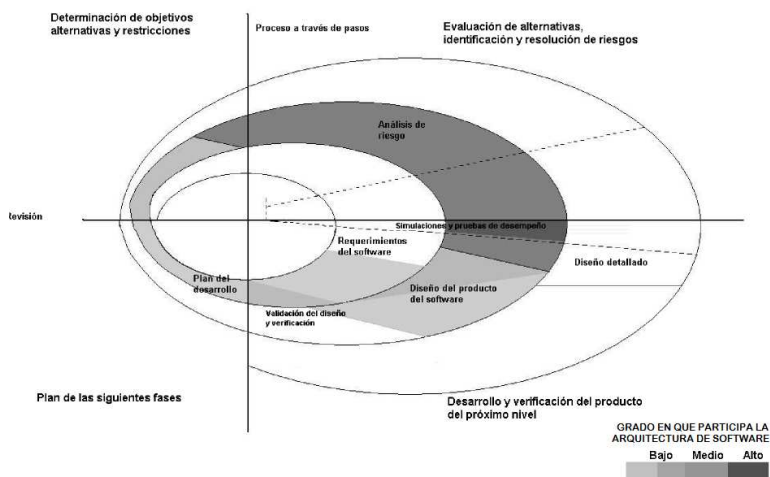


Figura 2.1. La arquitectura de software en el ciclo de vida del software del modelo de espiral de Boehm (Boehm, 1988)

En la referida Figura 2.1 se pueden distinguir cuatro fases o etapas delimitadas por los ejes de cada cuadrante formado, las cuales se describen a continuación: la fase donde se determinan los objetivos, se fijan alternativas y se imponen restricciones (situada en el cuadrante superior izquierdo); la fase donde se evalúan alternativas, se identifican y se resuelven riesgos, es decir es una fase analítica y de valoración de prototipos (cuadrante superior derecho); la fase de desarrollo y verificación de los productos del próximo nivel, esta es la fase más constructiva ya que en ella se desarrollan diferentes productos que van conformando al sistema en sí; la fase donde se planean la siguientes fases, donde también se consolidan los elementos de la fase previa

mediante una revisión por parte de los actores de la organización involucrados con el producto para asegurar que las partes concernientes cumplen las expectativas que requiere la fase siguiente.

Una espiral que cubre a estas cuatro fases es un ciclo, cada uno indica el nivel de desarrollo que se tiene del sistema. Suponiendo que se quiere desarrollar un sistema completamente nuevo, el punto de partida de estos ciclos lo conforman el plan para su consecución y el plan de requerimientos, pero si se quiere modificar uno ya existente entonces se puede partir del nivel y fase que corresponda al nivel de profundidad de los cambios requeridos. La intervención de la AS dentro de este proceso está marcada con la parte sombreada de la Figura 2.1. La intensidad del sombreado indica el grado de intensidad de las tareas arquitectónicas y el grado en que ésta se involucra en el proceso de desarrollo del software.

La AS empieza a ser formada desde que inicia el ciclo de la especificación de los requerimientos de software, cuando se establecen los puntos de referencias del dominio del problema para su entendimiento el arquitecto debe interpretar, conciliar y proponer alternativas de solución en función del conocimiento de los expertos del dominio y de los planificadores del proyecto tal como señala (McBride 2007) que ilustra esquemáticamente la interacción entre estos actores, mostrado en la Figura 2.2. El propósito de esta interacción es llegar a establecer una integridad conceptual del modelo a desarrollar expresada mediante la arquitectura, esta integridad conceptual es la clave para el desarrollo de un sistema tal como se señala en (Clements 2003, pag. 13) con alusión al libro del *“The mitical Man-Month”* de Fred:

“La integridad conceptual es la clave para el diseño de sistemas con éxito y tal integridad conceptual puede ser

tenida por un pequeño número de mentes llegando juntos a diseñar la arquitectura del sistema”

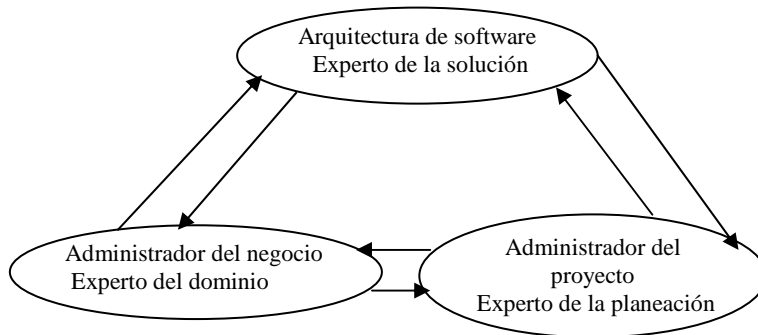


Figura 2.2. Interacción de la AS con los actores involucrados en el ciclo de requerimientos. (Tomado de (McBride 2007)).

Al avanzar al siguiente ciclo donde se realiza el desarrollo del plan, se identifican los requerimientos significativos arquitectónicamente que son aquellos que tienen un impacto esencial sobre la arquitectura (Jazayeri et al 2000), esto consolida la integridad conceptual antes mencionada y refina la primera arquitectura que expresa principalmente a elementos genéricos que representan a los requerimientos funcionales.

En el siguiente ciclo correspondiente a la fase de determinación de objetivos, alternativas y restricciones, se toman decisiones arquitectónicas que involucran a los actores del diseño del sistema, decisiones como las que se citan en (Tyree Akeman 2005) que tienen que ver con cuestiones de cómo implementar los requisitos y con situaciones de riesgos, costos y viabilidades de las posibles alternativas que se tienen en la organización dónde el sistema se desarrolla, por ejemplo si se usa un sistema legado, si se construye uno nuevo, o se usan elementos(componentes) de software prefabricados

disponibles en el mercado llamados *COTS*¹. Conforme se avanza por este ciclo los requerimientos se van incorporando en forma iterativa a la arquitectura hilándose entre sí, como se describe en (Nuseibeh 2001), donde se presenta una adaptación del *modelo de Boehm* destacando cómo los requisitos están al mismo nivel que la arquitectura, -de ahí el nombre que le dieron a su modelo *Twin Peaks*- enfatizando la igualdad en importancia entre la arquitectura y los requisitos, en el sentido de que un entendimiento oportuno de los requerimientos y una elección adecuada de la arquitectura son claves para la gestión de proyectos de gran escala.

Los requisitos que son derivados de las necesidades de los usuarios son conocidos como funcionales o también como características operacionales del sistema, y son útiles para formar los elementos básicos de cada estructura que integra la AS, mientras que los requisitos que capturan la mayoría de las facetas de cómo estos requisitos funcionales deben llevarse a cabo se les conoce como requisitos no-funcionales, los cuales pueden ser expresados en términos de requisitos de calidad que tienen su origen en los servicios de calidad dentro del ámbito de las comunicaciones cómo se describe en (Manola Frank 1999), estos requisitos de calidad son los que tienen que ver con la AS.

Los requisitos de calidad están conformados por un conjunto de atributos de calidad, los más comunes de estos son escalabilidad, seguridad, desempeño y fiabilidad, estos sirven para dirigir el diseño de una AS, como se muestra en el método de diseño propuesto por Bass (Bass et al 2002), llamado "*diseño dirigido por atributos*", conocido técnicamente por sus siglas en inglés como ADD (*Attribute Driven Design*). Mediante estos atributos de calidad se llega a refinar las estructuras diseñadas con los requisitos

¹ Por sus siglas en inglés: Commercial Off-The-Shelf software

funcionales llegando a producir la especificación de la arquitectura, que constituye el producto más importante generado en este ciclo de desarrollo del software.

En el siguiente ciclo correspondiente al análisis de riesgos y elaboración de prototipos, la AS tiene una doble función, la primera está encaminada hacia ella misma en el sentido de que es analizada para comprobar si la AS soporta los requerimientos de calidad planteados, y la segunda función es el usarla como un prototipo para analizar el comportamiento de algunos aspectos del sistema antes de proceder a su diseño. Para el primer caso pueden usarse alguno de los métodos de análisis arquitectónico como el ATAM, CBAM o SAAM ¹ (Clements et al 2002-b), ALMA² (Bengtsson et al 2002). En el caso de los prototipos la AS tiene por lo menos dos formas de llevarlos a cabo, una es mediante la creación de *middlewares* para hacer la conversión de los elementos arquitectónicos a código ejecutable como PRISMA(Pérez 2007), o bien usar un lenguaje de descripción arquitectónica como *arjava*, o *acme* con el cual se pueda ejecutar el comportamiento a evaluar.

En el siguiente ciclo donde se construye el diseño de los productos de software y se realiza la validación y la verificación del mismo, se ejecutan los prototipos de la AS contruidos en la fase anterior y se hacen las adecuaciones pertinentes en caso necesario, después de esto la AS es usada para construir el diseño del sistema. Una vez más se puede adoptar la estrategia de usar un *middleware* para hacer el puente entre los elementos arquitectónicos con los del diseño, o bien hacer un refinamiento sobre los

¹ Siglas correspondientes a sus nombres en ingles: Architectural Trade-off Analysis Method (ATAM), Cost Benefit Analysis Method (CBAM) y Software Architecture Analysis Method (SAAM)

² Siglas correspondientes a su nombre en ingles de: Architecture Level Modifiability Analysis

elementos de la AS mediante técnicas de transformación de modelos. En cualquier caso se debe establecer un medio para conservar la traza entre estos elementos de *grano grueso*, que son los elementos arquitectónicos, con los elementos de *grano fino* que corresponden al diseño para asegurar que la arquitectura sea quien gobierne de manera efectiva las siguientes tareas del proceso.

Para concluir esta sección, hay que señalar que en cada fase del desarrollo las estructuras arquitectónicas van teniendo una diferente representación dependiendo del tipo de elementos que se vayan usando en cada fase del desarrollo, como más adelante se verá pueden ir desde módulos generales hasta estructuras a nivel de código. Por lo cual resulta importante precisar las características de los elementos que intervienen en la AS y cuáles son los asuntos de interés que en ella intervienen.

2.2 Los *concerns* arquitectónicos

Los *concerns*¹ de acuerdo a su definición dada por el estándar 1471 de la IEEE descrita en la sección 1.5 son considerados como aspectos críticos o que de alguna manera resultan importantes por los actores involucrados.

En esta definición hay dos términos que se tienen que precisar, lo que se considera como crítico y lo que puede ser importante. El primer término aquí se usa con la acepción dada por el diccionario de la Real Academia Española² *crítico*: “perteneciente a crisis”. De manera tal que los aspectos críticos son aquellos que deben atenderse para no producir

¹ El término *concerns* es habitualmente empleado tal cual en la ingeniería de software porque tiene un significado más amplio que su simple traducción (asuntos de interés) . Aquí se usa de acuerdo a la definición dada en el estándar 1471 de la IEEE descrita en la sección de referencias.

² url: <http://buscon.rae.es/draeI/>: 2.Adj. Crítico: Perteneciente o relativo a la crisis.

un estado de crisis en el desarrollo del sistema pues de no atenderlos se llegaría a afectar al tiempo, recursos y actividades programadas o bien al diseño y comportamiento del mismo sistema obtenido. Con respecto a lo que se considera como importante se toma lo expuesto por (Filman et al 2005, pags. 3-4) en donde los intereses importantes son aquellos donde se involucran cosas que tienen un comportamiento sistemático, es decir los que tienen trascendencia en el desarrollo del sistema.

Como se aprecia en su definición los *concerns* tienen la característica de ser relativos ya que son dependientes de los actores. La dependencia entre concerns y actores está representada dentro del modelo conceptual propuesto por el estándar 1471 de la IEEE (IEEE Std 1471- 2000, pag. 5) descrito en la Figura 2.3.

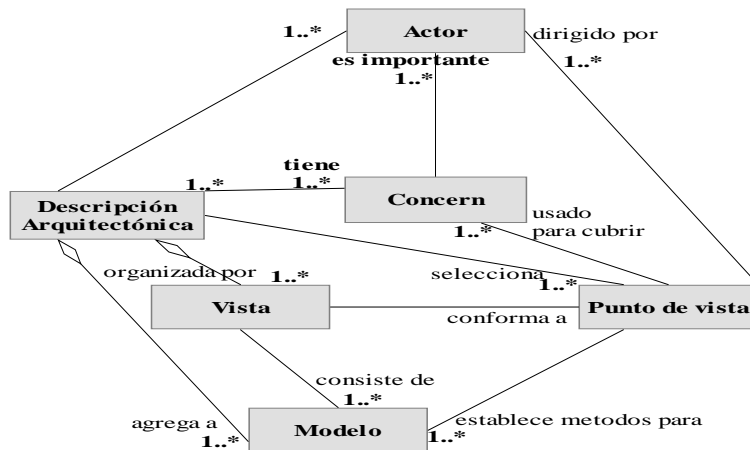


Figura 2.3. Modelo de la arquitectura para sistemas intensivos de acuerdo al estándar 1471 de la IEEE

En esta Figura 2.3 se representan al *concern* y al *actor* como clases enlazadas entre sí mediante una relación de asociación con una correspondencia (*cardinalidad*) de n a n . El rol en la asociación indica que un actor tiene (o está involucrado) con uno o más *concerns*, y que un *concern* es importante para uno o más actores. También en este modelo se muestra cómo un *punto-de-vista* que está dirigido por uno o más actores es usado para abarcar a uno o más *concerns*. Un *punto-de-vista* contiene las técnicas y lenguajes utilizados para producir resultados relevantes para los *concerns* que los actores dirigen a través de él.


Los *concerns* no solo están presentes a nivel arquitectónico, sino que estos van surgiendo conforme se avanza en cada ciclo de desarrollo del sistema (Filman et al 2005, pags. 481) y se van documentado de acuerdo al producto que se va creando en cada ciclo, así por ejemplo un *concern* relativo a seguridad puede surgir en el ciclo de los requisitos y se documenta mediante una especificación del mismo, a nivel arquitectónico la seguridad se representa en otros *concerns* como la autenticación y encriptación, y al nivel de implementación surgen otros más en la misma línea, como por ejemplo los mecanismos utilizados para la autenticación y encriptación.

A los *concerns* que se presentan al inicio del ciclo de desarrollo se les conoce como *tempranos* (Early-Aspects 2004) y abarcan a los requisitos y a la arquitectura, a los que se presentan en el ciclo del análisis y diseño se les llama *intermedios* y a los que se presentan a nivel de implementación se les designa como *concerns tardíos*. Esta es una manera de designarlos para ubicarlos dentro de proceso de desarrollo. Aquí concentraremos la atención en los *concerns* tempranos específicamente a los que son del ámbito de la arquitectura.

Como ya se mencionó en la sección anterior, dentro del ámbito de la arquitectura los atributos de calidad son los que forman los principales *concerns arquitectónicos* debido a que dirigen el desarrollo de su diseño. Estos son parte de los requerimientos no-funcionales de una aplicación, es decir provienen de las características que deben cumplir el diseño, la implementación y el desarrollo.

Dentro de la ingeniería de software se han detectado propiedades de calidad que deben cumplir los sistema en mayor o menor medida llamadas informalmente como *ilities*¹ o también designados como *atributos*² de calidad.

En la Tabla 2.1 se presentan distintas taxonomías sobre los atributos de calidad que se han identificado: en la primer columna se listan los atributos de calidad que son analizados generalmente en los sistemas de software como se cita en (Tzilla 2005, pag. 372).


En la segunda columna se listan los atributos de calidad considerados para la arquitectura los cuales fueron mencionadas en OMG-DARPA Workshop on Compositional Software Architectures (Tyson 1998), en esta columna se aprecia cómo mas de la mitad de estos atributos (los primeros nueve remarcados con el fondo gris ) coinciden con los atributos considerados en forma general para los sistemas, los que no coinciden fueron considerados más apropiados para la arquitectura que para los sistemas en general.

En la tercera columna se listan los atributos que de acuerdo al modelo de calidad del ISO 9126 deben tener los productos

¹ Esto es por la terminación de su nombre en ingles que tienen la mayoría de estos, como por ejemplo *reliability* y *availability*

² Atributo: Una propiedad medible física o abstracta de una entidad (ISO 9126 1991, pag. 16).

de software, los cuales están clasificados en seis categorías (fiabilidad, funcionalidad, usabilidad, eficiencia, mantenibilidad y portabilidad) que a su vez son sub-divididos en otros más específicos como puede apreciarse, las coincidencias con los atributos de las columnas anteriores también han sido remarcadas con fondo gris.

Finalmente, en las dos últimas columnas se despliegan los atributos de calidad que les corresponde tratar a la arquitectura de software de acuerdo a Gorton y a Clements autores de sus respectivos libros (Gorton 2006) y (Clements et al 2003), ellos coinciden en cuatro atributos (remarcados con el fondo gris ). En cada libro se les dan enfoques diferentes, por ejemplo el primero se enfoca a la definición de los atributos y a la dificultad para poder medirlos, y en el segundo caso la atención a los atributos se enfoca a la forma de cómo poder usarlos como guías para dirigir el diseño de la arquitectura, por otro lado en cada libro se destacan atributos que en el otro no los trata, sin que ello signifique que sean más o menos relevantes en la arquitectura, simplemente son apreciaciones diferentes. Dichos atributos discutidos por ambos autores son descritos a continuación puesto que constituyen la base de los principales *concerns* que intervienen en la arquitectura.

Desempeño. Tiene que ver con el tiempo en que se llevan a cabo las tareas en dos principales aspectos que no son excluyentes, uno es la cantidad de trabajo que debe ser realizado en un cierto tiempo y otro es el límite de tiempo conveniente para la correcta operación de las tareas. En el primer caso se refiere a la capacidad de procesamiento y se mide por las transacciones por segundo o mensajes procesados por segundo.

El segundo caso abarca a dos requerimientos más específicos, una se refiere al tiempo de respuesta o latencia

en que una aplicación debe responder a alguna de las entradas que se tenga, la cual puede ser especificada como una garantía en que siempre se tendrá como máximo un cierto valor del tiempo de respuesta o que en promedio los tiempos de respuesta debe cumplir con un cierto valor, y el segundo requerimiento se refiere al tiempo límite (*deadline*) en que una tarea debe llegar a cumplirse, puesto que al excederlo hace inútil su resultado.

El desempeño ha sido ampliamente tratado dentro de la ingeniería de software (hay libros completos que lo estudian), lo importante en la arquitectura es analizar el posible comportamiento de los elementos de software en función de este tipo de requerimientos para diseñar las estructuras que mejor respondan a ellos.

Disponibilidad. De manera general puede definirse como la probabilidad de que un sistema sea operativo cuando este sea necesitado (Clements et al 2003, pag. 79), o viéndolo de otra manera es la probabilidad que el sistema no falle cuando sea requerido.

Un sistema falla por algún defecto¹ del software que se presente en un momento determinado el cual no puede ser corregido por él mismo haciendo que deje de ser operativo mientras se repara el fallo. El que el sistema esté disponible operativamente significa que el software debe ser capaz de estar en un estado tal que permita cumplir una función requerida en un punto dado en el tiempo bajo condiciones establecidas de uso, tal como se especifica en (ISO 9126 1991, pag. 11).

La disponibilidad puede verse también como una combinación de otros atributos más específicos como la

¹ Un defecto no es observable por los usuarios del sistema hasta que el sistema deja de ser operativo, es decir el defecto se convierte en fallo (Clements et al 2003, pag. 79)

madurez, tolerancia a fallos y capacidad de recuperación. Estos atributos son definidos en el modelo de calidad de ISO 9126 antes referido de la siguiente manera: la madurez es la capacidad del producto de software para evitar fallar como consecuencia de errores en el software; la tolerancia a fallos es la capacidad del software para mantener un nivel específico de desempeño en casos de defectos del software o de la violación de su interfaz especificada; por último la capacidad de recuperación es la capacidad del software para restablecer su nivel de desempeño y recobrar los datos directamente afectados en caso de un fallo.

Tabla 2.1. Diversas taxonomías sobre los atributos de calidad que debe cumplir el software

Para los sistemas en forma general de acuerdo a (Tzila 2005, pag. 372)	Mencionadas en OMG-DARPA (Thompson 1998)	Consideradas en el modelo de calidad del ISO 9126 (ISO 9126 1991)	Consideradas para la arquitectura de software.	
			(Gorton 2006)	(Clements et al 2003)
<ul style="list-style-type: none"> fiabilidad seguridad interoperabilidad poderle dar mantenimiento adaptabilidad poder hacerse modular poder ser portado evolutivo ser entendible reutilizable propiedad de ser correcto eficiencia poder ser predecible tolerancia a fallos poderse recuperar ser capaz de aprender analizable robustez poder ser probado verificable consistencia poder seguir las trazas poder ser medido 	<ul style="list-style-type: none"> fiabilidad seguridad interoperabilidad poderle dar mantenimiento adaptabilidad poder ser compuesto poder ser portado evolutivo ser entendible poderse adecuar extensibilidad poder subsistir (o sobrevivir) escalabilidad ser accesible financieramente desempeño calidad de servicio ser movable 	<ul style="list-style-type: none"> fiabilidad madurez tolerancia a fallos capacidad de recuperación funcionalidad poderse adecuar precisión interoperatividad seguridad usabilidad entendible capacidad de aprender operatividad ser atractivo para el usuario eficiencia comportamiento de los tiempos de respuesta y proceso utilización de recursos poderle dar mantenimiento ser analizable posibilidad de cambio ser estable posibilidad de probarlo poderse portar adaptabilidad ser instalable poder coexistir ser reemplazable 	<ul style="list-style-type: none"> desempeño disponibilidad capacidad de ser modificado seguridad escalabilidad 	<ul style="list-style-type: none"> desempeño disponibilidad capacidad de ser modificado seguridad capacidad para ser probado usabilidad

Modificabilidad. Este atributo sirve para medir que tan fácil puede ser cambiada una aplicación para atender a nuevos requerimientos funcionales y no-funcionales ya que los cambios en el software son parte de su ciclo de vida. En enfoques fuera de la arquitectura esto se trata como la facilidad para darle mantenimiento pero en esos casos no se consideran a los requerimientos no-funcionales, es decir en el caso de la arquitectura los cambios se tratan en un sentido más amplio puesto que una de las principales tareas de la arquitectura es facilitar los cambios anticipándose a ellos.

Seguridad. En términos generales la seguridad es definida de acuerdo a (ISO 9126, 1991) como la capacidad del software para proteger información y datos de tal manera que personas o sistemas no autorizados no puedan leer o modificar a estos, en cambio personas o sistemas autorizados tienen acceso a ellos.

A nivel arquitectónico los requerimientos de seguridad se centran en cinco tópicos, estos son: autenticación, las aplicaciones pueden verificar la identidad de sus usuarios y otras aplicaciones con las cuales ellas se comunican; autorización, los usuarios y aplicaciones autenticadas tienen definidos los derechos de acceso al sistema.; encriptación, los mensajes enviados de o hacia la aplicación son encriptados; integridad, esto significa asegurar que los contenidos de un mensaje no son alterados en su tránsito; y por último el no-desconocimiento, referido al hecho de que quien envía un mensaje tiene prueba de su entrega y de quien lo recibe tiene seguridad de la identidad de quien lo envió.

Escalabilidad. En términos generales escalabilidad es definida así: *“qué tan bien una solución para algún problema podrá trabajar cuando el tamaño del problema se*

*incrementa”*¹. Para el caso de la arquitectura esto es aplicable para saber que tanto el diseño de un sistema puede afrontar algún aspecto de los requerimientos de la aplicación que incremente en tamaño. Ejemplos de estos aspectos pueden ser: aumento en la carga de solicitudes (demanda), incremento en las conexiones simultaneas (conurrencia), y aumento en el volumen de datos.

Capacidad para poder ser probado. Esto se refiere a la facilidad con la cual el software puede ser hecho para poder detectar sus defectos a través de pruebas. Las pruebas se sustentan en el cálculo de la probabilidad de que el sistema falle en la próxima ejecución debido a un posible defecto, estas pruebas son hechas a lo largo del ciclo de vida del software por los diferentes actores que intervienen en ella. Una de las tareas de la arquitectura es encontrar la mejor manera de llevarlas a cabo para reducir los costos en su aplicación.

Usabilidad. Como su nombre lo sugiere son las facilidades que presenta el software a los usuarios para llevar cabo una tarea deseada y el soporte al usuario que el sistema le brinda. Este atributo puede ser sub-dividido en cinco propiedades mas especificas, las cuales son: características de aprendizaje del sistema para asistir a los usuarios en caso de que estos no conozcan alguna de sus características operativas; la posibilidad de usar un sistema eficientemente, esto es qué puede hacer el sistema para hacer más eficiente su operación; la capacidad de minimizar el impacto de errores, lo que sistema provee para que un error del usuario tenga el mínimo impacto; la adaptación del sistema a la necesidades del usuario; y por último lo que el sistema pueda hacer para aumentar la confidencialidad y satisfacción de los usuarios.

¹ Tomada de www.hyperdictionary.com

El uso de las definiciones de estos atributos de calidad nos ayudan como referencia para la especificación de los requisitos no-funcionales, el uso de ellos tal como se describen resulta inoperativo como lo señala (Clements et al 2003). Por ejemplo al especificar un requisito de calidad para un sistema como *“debe ser escalable”* tal especificación no ayuda en nada puesto que un sistema puede hacerse escalable en alguno de varios aspectos (volumen de datos, conexiones simultaneas o carga de solicitudes) pero no en otros, lo mismo sucede para los demás atributos de calidad.

Una solución es incluir en forma textual una especificación con información precisa, como se ilustra en el caso expuesto por (Gorton 2006, pag. 23) sobre la especificación de un atributo de escalabilidad para un cierto sistema: - *“debe ser posible escalar el desarrollo desde cien usuarios en ordenadores personales dispersos geográficamente planteados inicialmente hasta 10 mil usuarios sin un incremento en esfuerzo y costo para la instalación y configuración”* -, en este ejemplo se especifica que son las conexiones simultaneas las que se escalaran indicando también la magnitud, sin embargo no en todos los requisitos siempre es posible hacer una descripción a este nivel, y aún en los casos en que esto fuera posible hacerlo, la parte importante de esto es determinar cómo la arquitectura puede hacer que dichas especificaciones se lleven a cabo y de qué manera.

La intención de incorporar a los requisitos de calidad en la arquitectura es hacer que a través de ella se llegue a establecer los fundamentos de la calidad mediante la especificación de sus elementos y sus estructuras que servirán de base para los demás elementos de software que se diseñen e implementen en las siguientes fases de

desarrollo, ya que la arquitectura por sí misma es incapaz de llevarlos a cabo (Clements et al 2003, pag. 73).

Se han desarrollado varias propuestas para hacer operativos los requisitos de calidad dentro de la arquitectura de software, entre los más recientes se encuentran los trabajos de (Xavier Franch and Botella, 1998), (Bosch, 2000) y (Bass et al 2001).

En el primer trabajo Xavier Franch y Botella, se propone una notación llamada *NoFun* para expresar los requisitos no-funcionales, sin embargo no se hace alusión cómo los atributos pueden llegar a conformar la arquitectura para lograr el cometido de los requisitos planteados, ya que su atención se centra sólo en la manera de expresar a este tipo de requisitos mediante los elementos básicos arquitectónicos (componentes y conectores).

Por su parte Bosch presenta un método iterativo para diseñar la arquitectura en donde usa los requisitos funcionales como puntos medulares en la conformación de este diseño y en cada iteración se van introduciendo los requisitos no-funcionales los cuales hacen que el diseño se vaya modificando, no obstante de ser una buena aproximación de cómo llevar a cabo el diseño de la arquitectura no aborda con profundidad la manera de representar los atributos de calidad.

En la propuesta de Bass la atención se centra en los atributos de calidad considerados como los principales elementos que dirigen el diseño de la arquitectura, a diferencia de las anteriores propuestas que los dejan en un rol secundario. Esto se hace manifiesto desde el nombre de su método para definir a una arquitectura software llamado *diseño dirigido por atributos* conocido como ADD (el cual fue ya referido en la sección anterior). Este método emplea

colecciones de componentes y conectores llamados primitivas de atributos para cumplir las metas impuestas por los atributos de calidad, como ejemplo de estas primitivas se citan las siguientes: balance de carga, prioridad de asignación, maquina virtual, interprete (todas a nivel de componentes).

La dificultad en este método radica en la forma y maneras de identificar y elegir a las primitivas para llegar a cumplir algún requisito de calidad debido a que una primitiva puede ser útil para más de un requisito lo cual ocasiona efectos colaterales no deseados, además en el método no se precisa el cómo incorporar estas primitivas a la estructuras arquitectónicas que al fin de cuentas es lo que se pretende al definir la arquitectura.

Las situaciones de dificultad que se destacan en las propuestas examinadas en los párrafos anteriores son parte de los problemas que los requisitos no-funcionales generan a todo lo largo del desarrollo del software cuando se trata de hacerlos operativos e implementarlos.

Este tipo de problemas son generalizados como problemas de “*separación de concerns*” los cuales son abordados en el desarrollo orientado a aspectos (Filman et al 2004). En este trabajo se abordará la separación de los llamados *concerns tempranos* (*early concerns*) que son los que competen a la arquitectura de software.

2.3 La separación de concerns arquitectónicos

La separación de concerns ha sido una estrategia aplicada en el software para reducir la complejidad inherente que este presenta, puesto que al separar los diferentes asuntos de interés que intervienen en el software se logra una mejor comprensión de sus elementos y de las relaciones que se


establecen entre ellos, se reduce el esfuerzo que impone su análisis y diseño y se facilitan las tareas de mantenimiento y evolución. Parnas (Parnas 1972) fue uno de los primeros en aplicar la separación de concerns al usar módulos como unidades para separar (y a la vez agrupar) las responsabilidades que un sistema debe cumplir.

Actualmente existen varias maneras de realizar esta separación las cuales dependen de la metodología de desarrollo que se aplique, ya que cada una impone sus criterios sobre la forma en la que el software debe ser dividido.

Por ejemplo las metodologías de desarrollo estructuralista y la orientada a objetos dividen al software de maneras muy diferentes, la estructuralista realiza una separación modular dividiendo al sistema en funciones representadas por módulos que se relacionan entre sí en forma jerárquica. En cambio la metodología orientada a objetos hace una descomposición del sistema en clases/objetos, donde cada clase representa una abstracción y estas se pueden llegar a relacionar en diversas formas (como *es-parte-de*, *es-un*, y *asociación*)¹. En estos casos cada metodología aplica un criterio dominante para efectuar la descomposición de concerns llamado por Tarr como “tiranía de la descomposición dominante” (Tarr et al 1999), en el caso de la metodología estructuralista es la función lo que domina la descomposición, y en el caso de la orientación a objetos es la clase.

El problema que se presenta con la descomposición dominante es que no todos los *concerns* pueden llegar a ser agrupados (o descompuestos) en las unidades modulares

¹ Una comparación entre diferentes métodos de estas metodologías y los criterios en como aplican la descomposición del software es abordada en (Wieringa 1998).

que impone esa descomposición (en clases o módulos por ejemplo), como es el caso de los concerns no-funcionales que por su naturaleza de estar relacionados con más de un *concern* tienen que ser esparcidos en varias unidades modulares haciendo que se generen los llamados *crosscutting-concerns*, cuya descripción esquemática se muestra en la Figura 2.4. En esta Figura se muestra como un *concern* no-funcional (esquemático como ) atraviesa la modularidad donde se encuentra cada elemento, es decir produce un *crosscutting-concern*.

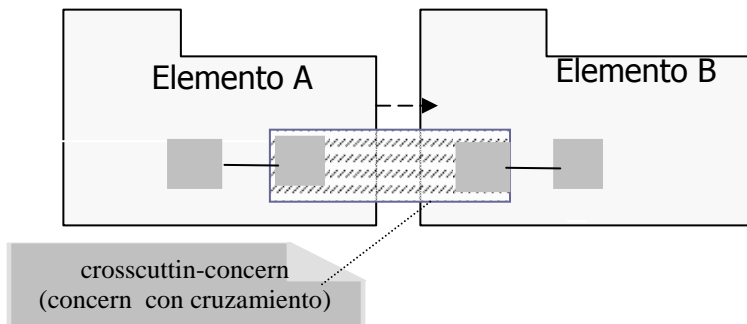


Figura. 2.4 Elementos modulares con un concern que presentan crosscutting

Los *crosscutting-concerns* fueron primero encontrados a nivel de programación lo que dio origen a la programación orientada a aspectos (Kiczales et al 1997), posteriormente fueron también detectados en fases anteriores a la programación (Filman et al 2005, pags. 414-421), desde la fases de requisitos (Rashid et al 2002), (Moreira et al 2002), (Brito and Moreira 2004), (Baniassad and Clarke 2004) y arquitectura (Perez et al 2003), (Cuesta et al 2004) hasta el análisis y diseño (Baniassad and Clarke 2004). Actualmente el tratamiento de estos es considerado dentro del desarrollo orientado a aspectos cuyo principal objetivo es precisamente mejorar la separación de *concerns* en todas las fases del desarrollo (Filman et al 2005) para evitar el código

enmarañado que se genera con los *crosscutting-concerns*. Los llamados *crosscutting-concerns tempranos* (Tekinerdoğan et al 2004) son los que tienen que ver con los requisitos y con la arquitectura, a ellos está enfocado este trabajo específicamente los relacionados con la arquitectura.

Para entender la forma en que se generan los *crosscutting-concerns* a nivel arquitectónico, se muestra en forma esquemática en la Figura 2.5 el proceso simplificado de la separación de concerns para la conformación de la arquitectura.

El proceso parte del dominio del problema de donde se obtiene la especificación de los requisitos, dicha especificación sirve como entrada para iniciar con la descomposición en los primeros concerns arquitectónicos usando un solo criterio dominante para dicha descomposición, usualmente se usa el criterio de las funciones que debe realizar del sistema (Kishi and Noda, 2001),(Clements et al pag. 157), (Videira and Bajracharya 2005), como resultado de esta descomposición se obtienen los primeros elementos arquitectónicos constituidos por unidades modulares o componentes, representados en la figura mediante los cuadros etiquetados con las letras *a,b,c,d*, estas unidades modulares constituyen los concerns base de la arquitectura.

Por otro lado, desde la especificación de requisitos se llegan a detectar *concerns* que pueden causar *crosscutting* (indicado en la figura con el signo X), que también son tomados en cuenta para la conformación de la arquitectura.

Algunos trabajos sobre requisitos detectan los *crosscutting concerns* a este nivel como el de (Rashid et al 2002), otros además de detectarlos les dan un tratamiento para evitar sus efectos como (Brito and Moreira 2004) que crea reglas

de composición para deshacer a los *crosscutting*, otros incorporan los requisitos no-funcionales para formar la especificación de requisitos aspectual como las propuesta de (Rashid et al 2003) que identifica y resuelve los conflictos de cruzamientos de requisitos, o los trabajos de (Moreira et al 2002), (Sousa et al 2004), (France, et al, 2004) y (Araújo et al 2004) que hacen adaptaciones al análisis del proceso unificado de desarrollo (Kruchten 2000) modelando los requisitos no-funcionales (o atributos de calidad) desde la obtención requisitos.

Para representar la interacción entre los requisitos no-funcionales y la especificación de requisitos se usa una línea punteada entre ellos. Lo importante de estas propuestas es que los *crosscutting* concerns son manejados desde la especificación de requisitos, lo cual repercute en los requisitos que son significantes arquitectónicamente, es decir en aquellos requerimientos que tienen un impacto esencial sobre la arquitectura (Jazayeri et al 2000).

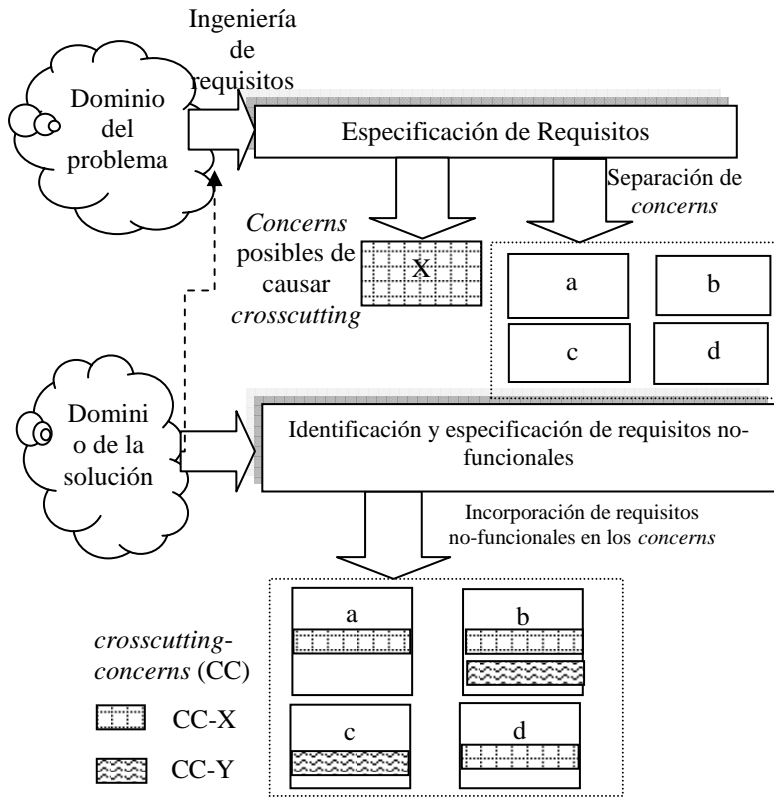


Figura 2.5. Proceso de separación e identificación de *concerns* y *crosscutting-concerns* en la fase de la arquitectura de software.

Por su parte el proceso a nivel arquitectónico continúa con la identificación y especificación de requisitos no-funcionales usando como fuente todos los *concerns* derivados de la especificación de requisitos más los requisitos de calidad que se llegan a detectar en el dominio de la solución como se representa en la Figura 2.5, de tal manera que todos los requisitos no-funcionales (los transmitidos desde la fase de requisitos más los detectados por la arquitectura) se incorporan en los *concerns* base esparciéndose en las unidades modulares existentes, trayendo como

consecuencia que se produzcan los *crosscutting-concerns* (CC). En la referida Figura 2.5, esto es ilustrado mediante el requisito no-funcional *X* que está esparcido en las unidades modulares *a, b* y *d* provocando el *crosscutting concerns* *CC-X*, y el requisito no-funcional *Y* que involucra a los módulos *c* y *b* generando el *crosscutting* *CC-Y*.

Durante la ilustración de este proceso para separar *concerns* a nivel arquitectónico (aunque el nivel de requisitos es incluido también aunque sea indirectamente), se detectan las siguientes cuestiones:

- (a) ¿Cómo lograr una adecuada separación de *concerns* arquitectónicos, cuando existen más de una dimensión que interviene en ellos? . Esto es porque la arquitectura está formada por varias estructuras y cada una es en realidad una dimensión.
- (b) ¿Cómo se detectan los *crosscutting concerns*? . Pues aunque se sabe que los requisitos no-funcionales son por naturaleza *crosscutting*, falta ver la manera en que estos se llegan a incorporar a los *concerns* provocando *crosscutting*.
- (c) ¿Cómo lograr resolver el problema que se causa con los *crosscutting concerns*?. Esto implica evitar que se tengan esparcidos *concerns* en varias unidades modulares.

Como se mencionó anteriormente la cuestión (a) está vinculada con el hecho de tener una dimensión dominante para separar los *concerns*, puesto que no todos los *concerns* pueden ser descompuestos adecuadamente aplicando un solo criterio y por supuesto esto repercute en la formación de los *crosscutting*. La cuestión (b) tiene que ver con la manera de incorporar los requisitos de calidad, y la cuestión

(c) tiene que ver con la manera de modelar a los crosscutting.

Aunque este trabajo está dirigido principalmente a buscar la manera de formar y vincular las diferentes estructuras arquitectónicas, lo cual está estrechamente relacionado con la cuestión (a), las propuestas que se revisan a continuación también abordan las otras cuestiones por la estrecha relación que hay entre ellas.

Entre las propuestas más relevantes enfocadas a resolver los problemas que se producen en la separación concerns por emplear una dimensión dominante están las de (Tarr et al 1999), (Filman et al 2004, pages 225-440),(Baniassad and Clarke 2004),(Kandé and Strohmeier 2000),(Kandé 2003),(Navasa, et al, 2005),(Perez, 2006) y (Navarro, 2007). Tarr propone la descomposición multidimensional como la alternativa para manejar la variedad de concerns que se tienen a nivel de programación mediante lo que él designa como *hiperlices*, de tal manera que una *hiperlice* agrupa a cada una de las variedades de concerns que se encuentren en un programa, donde el conjunto de todas ellas representan al sistema.

Para poder establecer diferentes relaciones entre las *hiperlices* se utiliza una composición de éstas en unidades modulares llamadas *hiper-modulos*, esto se lleva a cabo mediante reglas de composición que indican cómo deben ser relacionadas las *hiperlices*, aunque esta estrategia es buena carece de una notación formal además de que solo se ha aplicado a nivel de programación.

En cambio en (Filman et al 2004, pages 225-440) Clarke y Wlaker, proponen un mecanismo de separación de *concerns* a nivel de requisitos agrupadas en unidades llamados temas expresados en la notación de UML, de ahí el nombre dado a su propuesta Theme/UML. Un tema es una clase

estereotipada de UML <<*theme*>> con la característica de ser una clase con parámetros que sirve de *plantilla* (template). Mediante los parámetros se maneja el comportamiento de los crosscutting *concerns* ya que hacen que cada clase <<*theme*>> pueda ser adecuada a diferentes tipos de operaciones.

Además estos parámetros también son usados para hacer una composición de relaciones entre los temas. Una de las carencias que tenía este trabajo es que hasta el momento en que se publicó se carecía de una metodología para llegar a formar los temas, dicha debilidad fue cubierta con una mejora que se le hizo en la propuesta de (Baniassad and Clarke, 2004), en este trabajo se introduce una notación para especificar requisitos en forma textual denominadas vistas, y un mecanismo mediante el cual el desarrollador refina estas vistas con la intención de llegar a encontrar cuales son las funcionalidades del sistema que llegan a producir crosscutting.

Una vez detectado los crosscutting se lleva a cabo una planificación para crear las clases <<*theme*>>. Estas dos últimas propuestas resultan muy útiles para el análisis y diseño de la separación de *crosscutting-concerns* tempranos, pero su enfoque es solo hacia los requisitos sin incursionar al ámbito arquitectónico.

Siguiendo con la estrategia de la separación multidimensional de *concerns* pero en este caso enfocada específicamente a la arquitectura, la propuesta de (Kandé and Strohmeier 2000) introduce tres mecanismos de abstracción para llevar a cabo esta separación: los puntos de vista arquitectónicos, los espacios de concerns arquitectónicos y las vistas arquitectónicas. El primer mecanismo está especificado en el estándar de la IEEE1471 (IEEE Std 1471-2000) abordado en el punto 2.1 de este

trabajo, sirve para expresar las reglas y notación para representar los concerns desde la perspectiva de los actores, las vistas por su parte se usan para expresar estos concerns, y por último los espacios de concerns agrupan a las vistas para poder llevar a cabo con ellas diferentes tratamientos.

Esta propuesta se profundiza y complementa en la tesis de (Kandé 2003), donde estos mecanismos se modelan usando UML, además de introducir otros elementos a nivel de componentes para llevar a cabo la interrelación entre las vistas, sin embargo ahí no se trata la manera de cómo detectar los *crosscutting concerns*.

Algunas propuestas en lugar de detectar *crosscutting* y enfrentar sus problemas, evitan que estos se produzcan. Por ejemplo en (Navasa, et al, 2005) se propone el diseño arquitectónico creando primero un modelo básico de diseño, para luego en forma iterativa ir agregando en otra estructura a los elementos posibles de causar *crosscutting*, esto se logra estableciendo dos niveles arquitectónicos, uno integrado por elementos (componentes) del diseño arquitectónico y el otro nivel llamado de aspectos que agrupa a todos los elementos que son comunes a los elementos del otro nivel, y se usa un mecanismo para interrelacionar estos niveles evitando con ello el *crosscutting*. Para la especificación arquitectónica usa al lenguaje LEDA¹ el cual al no ser específico para la describir una arquitectura tiene que ser adaptado para este propósito.

Una de las propuestas más recientes dirigida a evitar la formación de *crosscutting* mediante la agrupación de *concerns* arquitectónicos (requisitos no-funcionales) en

¹ Es un lenguaje de programación multi-paradigma, es decir que soporta programación imperativa, orientada a objetos, programación lógica y programación funcional. Una referencia completa de LEDA puede verse en el libro: *Multiparadigm Programming in Leda*, de Timothy A. Budd, Addison-Wesley, 1995

unidades llamadas *aspectos* es la que se realiza en (Perez 2006), no solo resuelve este problema sino que también cuenta con el soporte formal y tecnológico que permite obtener una especificación arquitectónica *aspectual*. Esta propuesta se fundamenta en un lenguaje llamado PRISMA diseñado ex profeso para el manejo de aspectos usando componentes, que además de modelar los diferentes *concerns* funcionales y no-funcionales se llega hasta el nivel de código (en C#). Esta propuesta es complementada con la de (Navarro 2007) que permite obtener desde los requisitos una especificación arquitectónica de PRISMA manteniendo la trazabilidad entre el nivel de requisitos y el arquitectónico.

Dado que uno de los objetivos esta tesis es el contribuir a mejorar el modelo de PRISMA con la incorporación de vistas es necesario conocer a fondo este modelo para ubicar los puntos de esta contribución.

2.4 El modelo PRISMA en la separación de concerns

La estrategia que usa el modelo PRISMA para separar *concerns* se fundamenta en el hecho de que especifica todos los *concerns* que intervienen en la arquitectura dentro de unidades *aspectuales* de tal manera que tanto los requisitos no-funcionales cómo la funcionalidad del sistema son especificados como *aspectos* (Perez, 2006, Pag 130). Los *aspectos* son incluidos dentro de cada elemento básico arquitectónico (componente,conector...) como se muestra en la Figura 2.6. Ahí se ejemplifican cinco *aspectos* (presentación, distribución, funcional, seguridad, replicación) que se relacionan entre sí mediante un tejido (weaving) para mantener su cohesión dentro de un componente, estos aspectos a su vez utilizan a los puertos (Port1, Port2) como interfaces para comunicarse con el

exterior, es decir para ofrecer y recibir servicios de otros componentes.

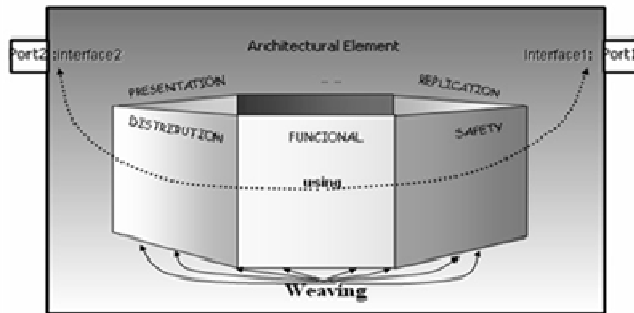
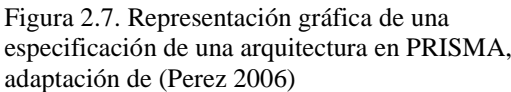


Figura 2.6. Los aspectos en el modelo PRISMA, tomado de (Perez 2006)

El *aspecto* como los demás elementos arquitectónicos son considerados como ciudadanos de primer orden, lo cual significa que no se realiza adecuación alguna para su modelado e implementación. Los demás elementos fundamentales que usa el modelo para especificar a una arquitectura son: componentes, conectores y sistemas. Los componentes sirven para expresar la funcionalidad del sistema software, los conectores se usan para intercomunicar a los componentes desempeñando el rol de coordinadores de esta intercomunicación, y los sistemas son usados para posibilitar la agrupación de varios componentes con lo que se consigue formar componentes complejos.

Un ejemplo de una especificación gráfica de una arquitectura con estos elementos se muestra en la Figura 2.7, en ella se aprecia dos tipos de conexiones entre los elementos: una es la que se establece entre un componente y un conector, lo cual se consigue conectando los puertos de dichos elementos a través de un canal llamado *attachment*; y la otra conexión es la que se establece entre un elemento arquitectónico que está dentro de un sistema y la interfaz

Dentro los componentes (en su vista interna) los aspectos también están enlazados entre sí para poder establecer secuencias de acciones entre los servicios que brinda cada aspecto. La manera en que un aspecto activa los servicios de otro aspecto se especifica por medio de un mecanismo denominado *weaving*. Un *weaving* sirve para coordinar los diferentes aspectos que los elementos arquitectónicos importan, por lo tanto están definidos dentro de los elementos arquitectónicos pero fuera de los aspectos.



40

Fase 1. Detección de los elementos arquitectónicos y aspectos. Los elementos arquitectónicos son derivados de la especificación de los requisitos. -por ejemplo para el caso del modelado de un robot que se analiza en la tesis de (Pérez 2006), se usa el manual del robot como fuente para derivar los elementos arquitectónicos-. A partir de los requisitos se identifican los componentes y conectores, y de ellos los servicios que prestan lo cual implica también la identificación de sus interfaces que corresponden a los puertos de entrada y salida. Para el caso de los crosscutting concerns se utilizan los requisitos no-funcionales para su especificación, donde cada uno de estos requisitos se le asocia el aspecto correspondiente.

Fase 2. Modelando la Arquitectura Software. Esta es la fase más elaborada de la metodología analizada puesto que en ella se pone de manifiesto el lenguaje PRISMA desarrollado para este fin y la herramienta de modelado e implementación llamada PRISMA-CASE. El lenguaje es utilizado para dos tareas, la primera consiste en la especificación de cada tipo de elemento arquitectónico que será usado y la segunda en la configuración de la arquitectura donde los tipos de elementos especificados son instanciados.

Esta fase se compone de cinco pasos que se ilustran en la Figura 2.8. En los primeros cuatro se especifican los diferentes tipos de elementos arquitectónicos y en el último se realiza la configuración de la arquitectura. En el primer paso se definen las interfaces que se identificaron en la fase previa, ya que al no depender de ningún otro elemento es lo primero en ser especificado modelando los servicios y firmas de dichas interfaces. En el segundo paso se modelan los aspectos que especifican la semántica de los servicios de las interfaces, tanto los aspectos como las interfaces son almacenadas en un repositorio para que

puedan ser reutilizados. En el tercer paso se definen los tipos que podrán ser usados por diferentes AS, que corresponden a elementos arquitectónicos simples, con esto se logra el propósito de la reutilización a nivel arquitectónico. En el cuarto paso se definen los sistemas mediante la composición de los componentes y conectores especificados anteriormente. Una vez que todos los elementos han sido definidos se procede a la configuración arquitectónica que es el último paso donde se instancian los elementos que fueron definidos previamente, para llegar a obtener la conformación de una arquitectura específica.

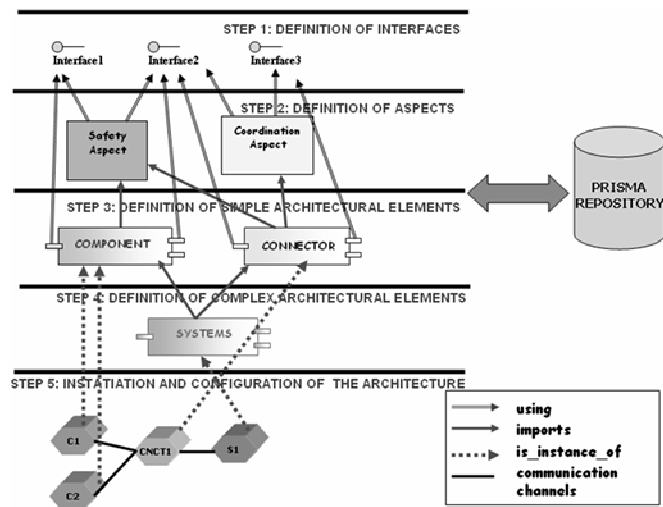


Figura 2.8. Los cinco pasos en el modelado de una arquitectura con PRISMA, tomado de (Perez, 2006)

Fase 3. Generación de Código y Ejecución. Al contar con la configuración arquitectónica de un sistema de software específico, lo que hace falta es llegar a conseguir que esta configuración sea ejecutable, esto se logra por medio de la herramienta PRISMA-CASE que se encarga de generar el código en C# en forma automática, de tal manera que este

código se convierta en la arquitectura ejecutable del sistema software modelado.

Como se aprecia, el modelo PRISMA permite modelar y hacer ejecutable una arquitectura software orientada a aspectos, y al agrupar todas las funciones de la arquitectura en los aspectos evita que se genere *crosscutting*.

PRISMA ha sido el punto de ignición para más proyectos de desarrollo algunas ya han concluido en otras tesis, por ejemplo la de (Navarro2007) la de (Ali 2008), y la presente propuesta por supuesto. En este caso se hará una revisión del trabajo realizado en (Navarro2007) con relación al tema de separación de concerns dentro de PRISMA.

En (Navarro 2007) se propone la obtención de los concerns arquitectónicos a partir de los requisitos, con el fin de llegar a generar una arquitectura software en el lenguaje PRISMA, su propuesta llamada ATRIUM¹ es una metodología que parte de un conjunto de necesidades de los usuarios o sistemas (*goals*) para que mediante un proceso iterativo se llegue hasta la instanciación de la arquitectura. Este proceso está compuesto por tres actividades principales: definición de objetivos, definición de escenarios, y la actividad compuesta por síntesis y transformación. Estas actividades y los artefactos empleados en cada una son mostrados en la Figura 2.9. Los artefactos que están en la parte superior de la Figura antes referida (encima de las actividades) son las entradas y los artefactos que están debajo de cada actividad son sus salidas, nótese que el producto final de todo esto es la proto-arquitectura de PRISMA instanciada.

¹ ATRIUM, es el acrónimo en ingles de Architecture Traced from Requirements applying an Unified Methodology que traducido al español equivaldría a: *traza de la arquitectura desde los requisitos aplicando una metodología unificada*

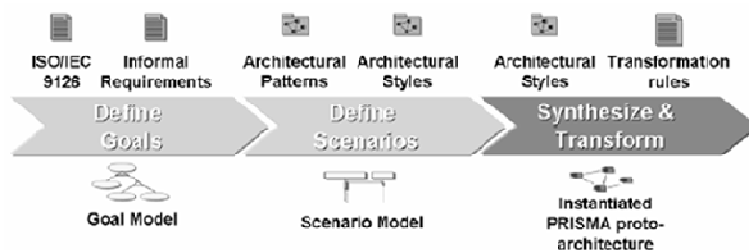


Figura 2.9. Actividades y artefactos de ATRIUM, tomado de (Navarro. 2007)

La cuestión aquí es revisar cómo se realiza la separación de concerns en la actividad de definición de objetivos, por lo que las demás actividades no serán detalladas en esta sección. En términos generales la actividad para definir metas parte de los requerimientos expresados en lenguaje natural, y usa los atributos de calidad definidos por el ISO-9126 como marco de referencia para llegar a obtener el modelo de objetivos. Este modelo de objetivos es generado mediante un proceso que comprende las siguientes actividades con sus respectivos productos:

- a) *Obtención y especificación de objetivos.*** Aquí se obtiene como producto la primera versión del modelo de objetivos lo cual constituye la entrada para la siguiente actividad.
- b) *Análisis de lo obtenido en (a).*** Aquí se generan dos productos que son: el modelo de objetivos analizado y los resultados de la evaluación, el segundo producto es usado en (a) en forma iterativa hasta conseguir la versión final del modelo analizado.
- c) *Validación del modelo de objetivos de ATRIUM.*** Aquí se producen los resultados de la evaluación que son retroalimentados en forma iterativa a la

actividad (a) hasta la obtención final del modelo ATRIUM validado.

La actividad sobre la obtención y especificación de metas está dirigida a la detección y separación de concerns desde el espacio del dominio del problema. Esta actividad está compuesta de cinco tareas cuyo esquema y secuencia se muestran en la Figura 2.10, también se representan las fuentes de información (del lado izquierdo) y el producto final que es el Modelo de metas ATRIUM.

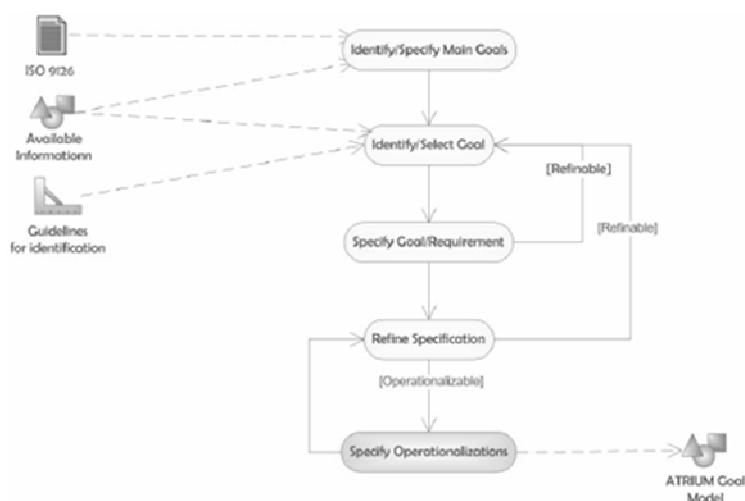


Figura 2.10. Flujo de trabajo para especificar objetivos y requerimientos, tomada de (Navarro, 2007)

Como se aprecia en el esquema de la Figura 2.10 los primeros *concerns* que se identifican corresponden a los objetivos principales, éstos son generados mediante un análisis que referencia la información disponible del dominio del problema con cada uno de los atributos de calidad que estén relacionados con la aplicación que se pretende modelar, por ejemplo para el caso que se ilustra en (Navarro

2007) se inicia usando como referencia el atributo de la funcionalidad. Los objetivos principales sirven para catalogar los nuevos objetivos que se van seleccionando en el siguiente paso donde son especificados en una plantilla que contiene los atributos que las definen y califican, aquí es donde también sus relaciones se van refinando.

Para que un objetivo pueda ser verificable se va descomponiendo en sub-objetivos, un objetivo llega a ser verificable cuando un proceso que debe llevar a cabo el sistema puede ser descrito, si esto se consigue entonces un requerimiento llega a ser especificado en lugar de un objetivo. Una vez que los objetivos y requisitos se han establecido se proceden a refinarlos, es aquí donde se establecen relaciones de dependencias entre los requerimientos/objetivos especificados con otros requerimientos/objetivos del modelo. También en este punto se llegan a especificar los crosscutting aunque la tarea de su identificación se la deja a los analistas.

El último paso en la especificación de los requisitos/objetivos consiste en hacer que ésta llegue a ser operacional, esto significa que se deben describir las soluciones posibles que permita al sistema llevar a cabo los objetivos, esto es proceso exhaustivo puesto que cada una de los objetivos/requisitos se someten a un análisis profundo para llegar a determinar cómo pueden ser operacional en el sistema.

Lo importante de la propuesta que se analiza es hacer notar su contribución respecto a la separación de *concerns*, de la cual se puede destacar que: a) Está enfocada a nivel de requisitos, por lo tanto sus principales esfuerzos están en el análisis del espacio del dominio para llegar a generar la especificación de los requisitos/objetivos; b) Su proceso se centra en el análisis de los objetivos; c) Se involucran los

requisitos no-funcionales, teniendo como principal referencia al modelo de calidad del ISO9126; d) Se usan herramientas de análisis que permiten ir refinando a los concerns (objetivos/requisitos) hasta un nivel que permite hacerlas operacionales por el sistema para llegar a la detección de los componentes que formaran la arquitectura en PRISMA.

Como se puede apreciar, el modelo PRISMA tiene un fuerte soporte para el análisis de concerns a nivel de requisitos. Todo lo realizado hasta el momento para ese modelo, va dirigido al diseño arquitectónico usando *componentes* y *conectores* como sus elementos principales de modelado e implementación. Considerando que esta es la única estructura que forma a una arquitectura. Sin embargo, se han detectado otras estructuras derivadas de propia definición de lo que es una arquitectura software (ésta formada por varias estructuras). Es decir hay otras estructuras que introducen otro tipo de elementos y por lo tanto involucran otros tipos de *concerns*. Esto será abordado en el siguiente Capítulo.

Capítulo 3

Modelos de vistas arquitectónicas de software existentes y la propuesta de su modelado

Este capítulo comienza por precisar el concepto de lo que aquí se considera como vista arquitectónica y del concepto puntos-de-vista que está muy relacionado con las vistas, lo cual permitirá tener una precisión de los términos aquí empleados. Luego se hace un análisis de las propuestas más importantes de los modelos de las vistas en la arquitectura de software, en función de los criterios considerados para separar los asuntos de interés (*concerns*) arquitectónicos. Del resultado de este análisis, se detectan los principales problemas y se presenta la propuesta para su solución que se hace en esta tesis. Como primer paso para iniciar con la propuesta se definen las bases del modelo de vistas que aquí se usa, especificando tanto los elementos y relaciones que lo constituyen. A continuación se realiza la especificación del modelo para cada una de las vistas básica aquí consideradas, la cual comprende a sus elementos, relaciones y notación correspondiente.

3.1 Precisando el concepto de vista arquitectónica

Se han hecho algunas analogías para entender lo que una vista es, aunque no todas son del todo apropiadas. Por ejemplo (Perry y Wolf, 1995) compara las vistas con los diferentes planos arquitectónicos de un edificio, específicamente con los planos de planta y elevación (proyección isométrica), cada uno equiparable a una vista,

pero esto refleja una idea de dependencia más que de separación de elementos diferentes, puesto que para crear el plano de elevación se requiere tener primero el de planta (el plano de elevación es en parte una proyección del de planta), siendo que lo que se pretende con las vistas es no hacerlas tan dependientes una de la otra (aunque estén relacionadas).

Otra analogía descrita en (Clements et al., 2002, pag. 13), compara a una vista con las diferentes interpretaciones o 'vistas' que tiene el ala de un ave: plumas, esqueleto, venas, músculo, etc, esta analogía resalta que la unión de las vistas forman a una arquitectura, cuando se declara que: todas estas 'vistas' forman la arquitectura de un ave. Una analogía semejante, hecha también por Clements en (Clements et al., 2003, pag 35), es sobre la ilustración de lo que es una vista sobre el estudio de cuerpo humano, comparando las diferentes percepciones que tienen los especialistas médicos sobre el estudio del cuerpo (traumatólogo, otorrinolaringólogo, etc.), cada vista en esta analogía toma en cuenta solo ciertas propiedades olvidándose de las demás. Esto último resalta el sentido de abstracción que tiene una vista, entendiéndose por abstracción a: 'una descripción simplificada o especializada de un sistema que enfatiza algunos detalles o propiedades del mismo mientras suprime otros' (Shaw 1984).

A veces se corre el riesgo de limitar este sentido de abstracción de una vista sólo como diferentes niveles de abstracción cuando en realidad las abstracciones, aunque diferentes, pueden ser del mismo nivel.

En este trabajo se usará como fundamento básico la definición de vista presentada por el estándar 1471 de la IEEE (IEEE Std 1471- 2000), y se hace un breve análisis de

esta definición para conducir hacia el concepto que aquí se maneja sobre una vista arquitectónica.

Una vista en términos básicos, de acuerdo a la definición del estándar 1471 de la IEEE, es lo siguiente:

‘Una vista es una representación de un sistema completo desde la perspectiva de un conjunto de concerns relacionados’ (IEEE Std 1471- 2000, pag. 3).

Esta definición es complementada con la especificación del modelo para una arquitectura de un sistema de software, que también se hace en (IEEE Std 1471- 2000), este modelo es mostrado en la Figura 2.3, donde se aprecia las clases relacionadas con la vista (descripción arquitectónica, punto-de-vista y modelo). Tomando como referencia al modelo de la Figura 2.3, se profundizará en la definición dada sobre una vista, en lo concerniente a: i) la característica de que una vista es una representación de un sistema; y ii) lo que se entiende por una perspectiva. Lo cual se detalla a continuación:

- i) Con respecto a que la vista es una presentación de un sistema, hay que resaltar que ésta abarca a todo el sistema, no solo a una parte de él, así que independientemente de qué elementos use una vista, con ellos se llega a modelar a todo el sistema. Esta representación se sustenta en un modelo que aporta las reglas de cómo pueden relacionarse sus elementos, y el método a seguir para su diseño.

La relación entre *vista* y modelo se aprecia en la Figura 2.3, donde se muestra también cómo una vista puede consistir de más de un modelo, y un modelo puede participar en más de una vista.

- ii) Con relación a la *perspectiva* bajo la cual se proyecta la vista, primero hay que hacer notar que la perspectiva viene de la percepción que tienen los actores involucrados en el sistema, es decir es un punto de vista de un actor (grupo o individuo). Con respecto a la perspectiva arquitectónica, esta dirige a los atributos de calidad que serán aplicados en la arquitectura como señala (Woods y Rozanski, 2005), donde también se hace alusión que una perspectiva en realidad corresponde a un punto de vista. Lo anterior se reafirma en el modelo mostrado en la Figura 2.3 donde en lugar de usar el término perspectiva se usa una clase llamada punto-de-vista.

Un punto-de-vista es definido como:

‘Una especificación de una convención para construir y usar a una vista. Un patrón o plantilla desde las cuales desarrollar vistas individuales, estableciendo los propósitos y audiencia para una vista, y las técnicas para su creación y análisis’. (IEEE Std 1471-2000, pag. 4)

Al definir un *punto-de-vista* como patrón de la *vista*, equivale a establecer una relación análoga a la que hay entre clase y objeto, es decir donde el punto-de-vista es equivalente a la clase, la cual contiene la especificación que se capta de los actores, y la *vista*, equivalente al objeto, sería una instancia que materializa dicha especificación mediante elementos arquitectónicos y sus relaciones.

En un *punto-de-vista* también se definen a las características de los concerns que serán incluidos en las vistas, creándose grupos de concerns relacionados, lo cual se indica en el modelo de la

(IEEE 1471, 2000) con la asociación entre estas dos clases (concern y punto-de-vista).

La relación entre vistas, puntos-de-vista y concerns es ilustrada esquemáticamente por (Kande y Strohmeier, 2000), lo cual se muestra en la Figura 3.1, apreciándose cómo cada uno de los dos puntos-de-vista ilustrados (el estructural y del análisis de la arquitectura), establece un grupo de concerns, cuyas reglas para definirlos están contenidos en lo que ahí se llama espacio-de-concerns, de dicho espacio también se llegan a generar las vistas que modelan y representan a estos concerns.

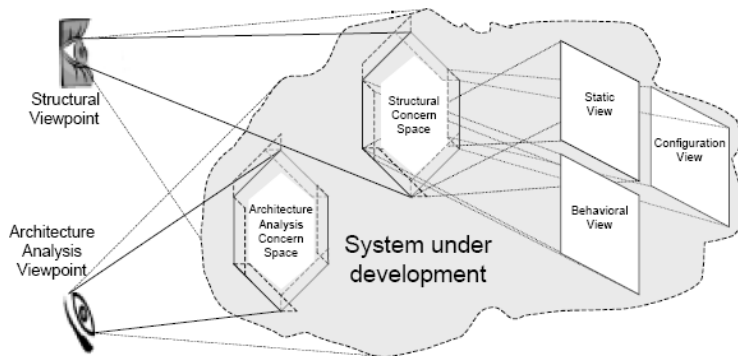


Figura 3.1. Relación entre *vista*, punto-de-vista y *concerns*, tomado de (Kande y Strohmeier, 2000)

Existe otra definición sobre punto-de-vista para los sistemas de software distribuidos que está dada por la norma ISO Reference Model of Open Distributed Processing (RM-ODP) (ISO/IEC DIS 10746-3, 1993). Ahí un punto-de-vista es considerado sólo como una forma de abstracción para seleccionar a las construcciones arquitectónicas con el fin de enfocarlas a concerns particulares dentro de un sistema, lo

cual se enlaza con el concepto de abstracción dado por (Shaw 1984), descrito en párrafos anteriores.

La norma de RM-ODP define cinco puntos-de-vista sobre un sistema distribuido y su ambiente, los cuales son generales y complementarios. Generales porque pueden ser aplicables a cualquier sistema de software aunque no sea distribuido, y complementarios porque al hacer una partición del sistema, cada uno de esos punto-de-vista comprende solo una parte de él, y entre todos se logra una completa descripción del sistema. Ellos son denominados como puntos-de -vista: empresarial, el de información, el computacional, el de ingeniería, y el tecnológico. Sin embargo los puntos-de-vista de esta norma hacen una partición con enfoque sistémico más que arquitectónico, puesto que no están dirigidos hacia la conformación de la arquitectura (en su definición ni siquiera se hace referencia a las vistas), sino más bien a separar los aspectos que se encuentran en los sistemas de software distribuido, como se puede apreciar en su descripción dada a continuación.

El punto-de-vista-empresarial, está dirigido hacia las necesidades de los usuarios de un sistema de información, siendo el más abstracto de todos porque ahí se declaran las políticas y requisitos empresariales de alto nivel.

El punto-de-vista de información, está enfocado al contenido de la información de la empresa. Las actividades que esto conlleva son: la identificación de elementos de información del sistema, las manipulaciones que pueden ser hechas con estos elementos y los flujos de información en el sistema.

El punto-de-vista computacional, conduce a la partición lógica de las aplicaciones distribuidas, independientemente del ambiente distribuido específico sobre las que ellas se

ejecutan. Éste describe la funcionalidad que el sistema proporciona, manteniendo oculto los detalles de la plataforma distribuida que soporta la aplicación.

El punto-de-vista de ingeniería, trata de los temas del soporte del sistema para las aplicaciones distribuidas, enfocándose a identificar la funcionalidad de la plataforma distribuida requerida para el soporte del modelo computacional.

El punto-de-vista tecnológico, como su nombre lo indica, se enfoca a identificar posibles artefactos técnicos para los mecanismos de ingeniería, y para las estructuras tales como las computacionales, de información y las empresariales.

La consideración y descripción de estos puntos-de-vista de la norma de RM-ODP sólo es útil en este trabajo para apreciar lo que se puede ser considerado por un punto-de-vista, ya que no están orientados hacia la arquitectura de software. Así que el concepto que es usado aquí sobre punto-de-vista es el dado por (IEEE 1471, 2000) porque es el que hace referencia a las vistas.

Para precisar ahora la definición sobre vista con respecto a la arquitectura de software, además de la definición básica analizada anteriormente, se toma en cuenta la dada por el grupo del Instituto de Ingeniería de Software (SEI: Software Engineering Institute), la cual considera a una vista como:

‘una representación de un conjunto coherente de elementos arquitectónicos tal como son escritos y leídos por los actores del sistema’ (Clements et al 2003, pag. 35).

Esta segunda definición sobre vista hace referencia explícitamente a la arquitectura, de esta manera el concepto

de vista arquitectónica que es usado en esta tesis es construido tomando en cuenta las dos definiciones anteriores, el concepto de una vista es:

Vista arquitectónica: *‘Es una representación de una estructura arquitectónica que está compuesta por elementos y de las relaciones asociadas con ellos, dicha estructura representa al sistema completo y es formada desde la perspectiva de un conjunto de concerns relacionados que involucra a los actores del sistema’.*

Una estructura es tomada aquí como: ‘el conjunto de elementos por sí mismos, tal como ellos existen en el software’ (Clements et al., 2003, pag. 35). Una estructura arquitectónica precisa que el conjunto de estos elementos de software, corresponde sólo a los que se toman en cuenta en el nivel arquitectónico, y no a los que están en todos los demás niveles del software.

La representación que una vista hace de una estructura, va más allá de la simple notación gráfica informal que se usó en los inicios de la arquitectura para representarla (líneas, cuadros y círculos), ya que en sí cada vista se fundamenta en un modelo el cual contiene los elementos y las reglas para establecer sus relaciones, es decir lo necesario para modelar a una estructura arquitectónica. Una vista en la arquitectura ha sido modelada de diversas maneras como se muestra a continuación.

3.2 Los modelos propuestos de vistas en la arquitectura

En el capítulo anterior, se ha mencionado que la separación de concerns en los sistemas de software en forma multidimensional reduce los problemas que se presentan en

el diseño del software. Uno de los medios para llevar a cabo esta separación en varias dimensiones, es a través de la descomposición del sistema en estructuras que representan las vistas, pues como se cita en (Courtois, 1985) ‘...la descomposición de un sistema en estructuras es esencial para un entendimiento del sistema con muchos niveles, y para resolver los problemas que surgen durante su análisis...’ y esto es precisamente lo que se pretende con las vistas, descomponer al sistema en estructuras.

En el nivel de requisitos, se ha mostrado cómo las vistas han sido útiles en la reducción de la complejidad del dominio del problema creando marcos de trabajo en función de ellas (Nuseibeh et al, 1994) y para el análisis de los requisitos (Kim et al, 2004). En el caso de la arquitectura, es donde las vistas son más adecuadas para separar concerns, ya que en este nivel coinciden concerns tanto del dominio del problema como de la solución, en este nivel es donde los actores involucrados requieren que la arquitectura represente a los diferentes elementos de alto nivel de abstracción (Shaw 1988), que ellos consideran importantes en el sistema del software.

Desde los inicios de la arquitectura de software se planteó a la necesidad de contar con varias vistas para separar a los diferentes elementos del software, como se describe en (Perry y Wolf, 1992):

‘...el arquitecto del software necesita un número de vistas diferentes de la arquitectura de software para varios usos y usuarios...son requeridas para enfatizar y entender diferentes aspectos de la arquitectura...’

En ese trabajo se plantearon como necesarias tres vistas, la de procesamiento, la de datos y la de conexiones. A partir de ahí se han hecho varias propuestas sobre los tipos de vistas y

numero de ellas que se consideran necesarias para formar la arquitectura, entre las más relevantes se encuentran la de Kruchten (Kruchten, 1995), la de Hofmeister, Nord y Soni (Hofmeister , et al. , 1999), la cual es conocida como la de Siemens¹, y la del SEI² (Clements et al., 2003). A continuación se destacan las vistas que se consideran como importante en cada una de estas propuestas, señalando cómo se realiza la inter-relación entre cada una de las vistas, principal objeto de estudio de esta tesis.

La propuesta de Kruchten. Esta propuesta consiste en un modelo llamado 4+1 vistas, que ha sido el referente de la mayoría de los trabajos sobre vistas publicados hasta la fecha. El nombre de este modelo describe el número de vistas que en él se proponen, 4 vistas son consideradas como ortogonales (aunque no del todo), y la otra es usada para vincular a las demás, en total son 5 vistas. En la Figura 3.2 se presenta el esquema de este modelo.

¹ SIEMS es el nombre de la corporación donde se desarrolló la propuesta

² SEI: Software Engineering Institute

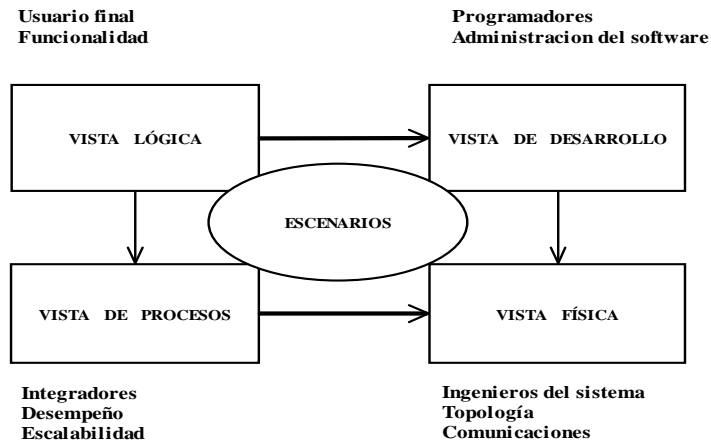


Figura. 3.2 El modelo de vistas 4+1, tomado de (Kruchten, 1995)

Como se muestra en la Figura 3.2, las vistas consideradas como ‘ortogonales’ son la lógica, la del desarrollo, la de procesos y la física, y la que las vincula es la de vista escenarios (también pueden ser los casos de usos), a continuación se describe la perspectiva y concerns que comprende cada vista.

Vista lógica. Esta vista tiene una perspectiva enfocada al dominio del problema, orientándose a la descomposición de los servicios que el sistema debe proveer a los usuarios finales, por lo tanto sus principales concerns son los requerimientos funcionales, los cuales pueden ser modelados a través de clases (en caso de usar orientación a objetos).

Vista de procesos. La perspectiva de esta vista enfoca a una parte del espacio de la solución, que comprende también a los requisitos no-funcionales de desempeño y disponibilidad del sistema. Los procesos son sus principales elementos y se ubican a diferentes niveles de abstracción, los cuales

permiten modelar los aspectos de concurrencia y sincronización del sistema. La forma en que efectúa la separación de concerns es mediante la partición del software en tareas.

Vista de desarrollo. La perspectiva en este caso, está orientada hacia la organización de los módulos de software que ya fueron diseñados e implementados dentro del ambiente de desarrollo. Los concerns considerados incluyen los requisitos internos sobre las facilidades del desarrollo, administración del software, reutilización y restricciones impuestas por el lenguaje de desarrollo.

Vista física. La perspectiva de esta vista comprende algunos de los requisitos no-funcionales, en este caso se abarca a la disponibilidad del sistema, fiabilidad, desempeño y escalabilidad, los cuales constituyen sus principales concerns. Esta vista refleja el aspecto de distribución del sistema porque los elementos de las vistas de desarrollo y de proceso tienen que ser asignados a los nodos de la red que conforman a esta vista física, con lo cual también se establece una correspondencia entre estas vistas.

Vista de escenario. Esta vista sirve para unir a las otras cuatro por medio de secuencias de interacciones que se llegan a establecer entre sus elementos. Tiene dos propósitos, sirve como una guía arquitectónica para descubrir elementos arquitectónicos en el momento del diseño, y para validar e ilustrar el diseño de la arquitectura. Esta vista no crea ningún vínculo persistente que mantenga la relación entre una vista y otra, pues las interacciones que se establecen entre ellas se realizan con fines de prueba y validación.

A pesar de que este modelo no cuente con una manera explícita de relacionar los elementos de una vista con otra,

estás mantienen una relación de correspondencia entre ellas, ya que los elementos de una vista son relacionados con los elementos de otra, lo cual denota que las vistas no son del todo independientes. Esta correspondencia es ilustrada en forma esquemática en la Figura 3.2 mediante las flechas que salen de una vista a otra. El caso de la correspondencia entre las vistas de desarrollo y la de procesos con la vista física ya fue descrito en los párrafos anteriores, en donde se definió a la vista física, las demás correspondencia se describen a continuación:

- La correspondencia entre la vista funcional y la de proceso. Esta relación se establece cuando los elementos de la primera vista, que pueden ser clases que representan las funciones del sistema, son asignados a los procesos y tareas de la vista de procesos.
- La correspondencia entre la vista modular y la de desarrollo. Esta correspondencia se establece al relacionar los elementos de la vista modular con los de la desarrollo en tres maneras diferentes: una de ellas se da cuando una clase es implementada por un modulo, otra cuando una clase grande (no se especifica una medida para cuantificarla) se descompone en múltiples paquetes, y la última correspondencia se forma cuando varias clases relacionadas se agrupan en subsistemas (o paquetes).

Para concluir con el análisis del modelo de Krutchen, hay que señalar que aunque éste fue propuesto para ser usado con cualquier metodología, ha tenido una tendencia a ser usado con la metodología orientada a objetos, como lo demuestra la adaptación que se hizo de éste modelo para ser incluido en el proceso unificado de desarrollo o mejor

conocido como RUP1 (Kruchten 2000). En la descripción que se hace de este proceso, se declara que el RUP está centrado en la arquitectura, aunque la arquitectura que se modela mediante este proceso, es para fines del desarrollo del sistema no para crear la arquitectura de software.

La propuesta de Siemens. Está propuesta plantea cuatro vistas como necesarias para formar la arquitectura, las cuales son: la conceptual, la de módulos, la de ejecución y la de código. Esta división de la arquitectura se fundamenta en el estudio de grandes sistemas de software, y en la propuesta realizada por (Soni et al, 1995) en donde se proponen por vez primera las cuatro dimensiones arquitectónicas que después se designan como vistas. Este enfoque establece una fuerte correspondencia entre los elementos de una vista con los de otra, ya que una vista produce elementos que son usados en otra, y a su vez cada vista está restringida por otra. Estas relaciones de correspondencia son mostradas esquemáticamente en la Figura 3.3 mediante flechas, la flecha de mayor grosor indica lo que una vista aporta a la otra, y la flecha más delgada las acciones o condiciones que una vista impone a la otra. A continuación se describe cada vista y se denota los vínculos que se van formando entre de ellas.

La vista conceptual. Esta vista comprende la estructura funcional del sistema que se describe a través de componentes funcionales enlazados mediante conectores, usando puertos y roles como interfaces. Esta vista es la más cercana al dominio de la aplicación porque está restringida por la plataforma software, como se denota en la Figura 3.3, con la flecha etiquetada como restricciones que sale de la vista de ejecución.

¹ De ahí que también a este modelo de Kruchten sea conocido como RUP/Kruchten 4+1

La vista de módulos. En este caso se modela la estructura concreta de los subsistemas mediante su descomposición en módulos, estos módulos son asignados a diferentes capas los módulos corresponden precisamente a los subsistemas, cada capa se relaciona con otra mediante un vínculo de dependencia llamada uso. Los elementos de la vista conceptual (componentes y conectores) son proyectados a esta vista como se aprecia en la Figura 3.3.

La vista de ejecución. Esta vista consiste en representar la funcionalidad del sistema con elementos tales como procesos y librerías compartidas que corresponden a la plataforma del tiempo de ejecución. Los procesos y librerías están enlazados mediante hilos y elementos de intercomunicación inter-procesos consumiendo recursos que le son asignados a la plataforma física. Esto se puede apreciar en la Figura 3.3, donde las vistas conceptual y modular aportan los elementos funcionales que serán llevados a los procesos de la vista en cuestión, nótese además las relaciones de interacción con la arquitectura del hardware y con el código fuente.

La vista de código. Es una apreciación del sistema en tiempo de diseño (del código), donde se muestra la implementación del sistema organizado dentro del código y componentes de desarrollo. Esta vista por lo tanto usa elementos de la vista de módulos y entidades ejecutables de la vista de ejecución como puede apreciarse en la referida Figura 3.3.

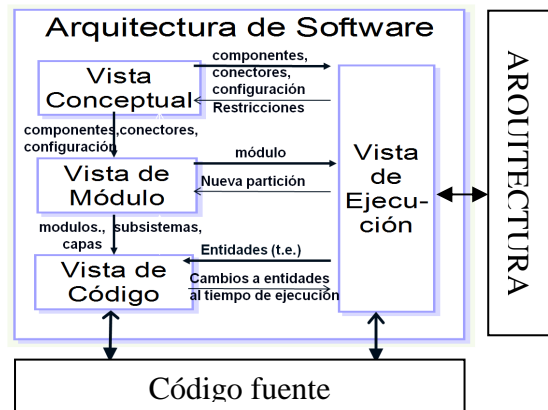


Figura 3.3 Relación entre las cuatro vistas de Siemens, (adaptación hecha de (Hofmeister, et al.,2000, pag. 20))

Como se aprecia en esta propuesta, no se requiere una vista adicional para relacionar a las cuatro vistas consideradas, ya que cada una de ellas se encarga de establecer las relaciones con la vista correspondiente.

La propuesta de SEI. Esta propuesta plantea tres vistas como las estructuras fundamentales de la arquitectura de un sistema de software, estas son la de módulos, la de componentes-y-conectores, y la de asignación. Vale la pena aclarar que en esa propuesta se usa el término de estructura en lugar del de vista, con alusión a lo que realmente representa una vista arquitectónica, es decir a una estructura. Cada una de estas tres vistas agrupa a su vez a otras vistas más específicas, las cuales comparten los mismos tipos de elementos dentro de su grupo, lo que las hace diferentes es la forma en se llegan a relacionar los elementos de cada una. Estas vistas específicas son llamadas en otros trabajos con el término de estilo arquitectónico (Monroe y Garlan, 1996),(Metayer, 1998), (Clements et al., 2002), (Nikunj, 2004). A continuación se destaca lo que cada una de estas vista fundamentales abarca, lo que se pretende

modelar en ellas, los elementos que las forman, sus relaciones, y las relaciones que se establecen entre una vista con otra.

La vista modular. Esta vista hace una partición de las funciones del software agrupándolas en unidades llamadas módulos (de ahí su nombre). Cada módulo representa las responsabilidades que el sistema debe cumplir de acuerdo a la especificación dada por los requisitos. Las relaciones que se establecen entre estos módulos denotan la forma en que se hace la partición del software, se pueden llegar a formar relaciones de dependencia, de descomposición (es otra forma de ver a la composición), y de jerarquía.

La vista de componentes-y-conectores. En esta vista se pretende modelar los aspectos del software en tiempo de ejecución, tales como concurrencia y comunicación de procesos. Los componentes y conectores son los elementos que se usan para representar a las entidades del software que intervienen en tiempo de ejecución (ejemplo de estos son: procesos, objetos, almacenes de datos). Las relaciones que se establecen entre ellos son básicamente de comunicación, donde los componentes se comunican a través de los conectores. El papel que cada componente desempeña determina un tipo de vista específica, así por ejemplo una vista específica llamada cliente-servidor se da precisamente por el papel que los componentes desempeñan como cliente y como servidor.

La vista de asignación. Esta vista sirve para asociar o asignar a los elementos de las otras dos vistas (módulos, componentes y conectores), a elementos del ambiente donde estarán ubicadas o por lo menos asociados. Estos elementos del ambiente son procesadores, unidades de almacenamiento, redes, y personas del equipo de trabajo. Los módulos son asignados a las estructuras de archivos, y a

los equipos de trabajo encargados de su desarrollo. Los componentes y conectores son asignados a las plataformas de ejecución, por ejemplo a procesadores y nodos de red.

La correspondencia que existe entre los elementos de una vista con los de la otra, en términos generales establece una relación entre sus elementos de muchos a muchos, como se describe en (Clements et al 2003, pag. 40). En la correspondencia entre una vista modular con la vistas de componentes-y-conectores, a un módulo le pueden corresponder uno o más componentes, y viceversa. En el caso de la correspondencia entre estas dos vistas y la de asignación, la relación entre sus elementos también puede ser de muchos a muchos si se considera a un sistema distribuido en donde un módulo puede ser asignado a varios unidades de almacenamiento, o un proceso(de componentes) puede ser asignado en varias unidades de procesamiento. En (Clements et al 2003) la relación entre los elementos de las vistas se condiciona al dominio de la aplicación.

Además de las tres propuestas previamente analizadas para modelar las vistas, han surgido otras, argumentado que cubren aspectos que dejan abiertos las demás. Por ejemplo el trabajo de (Tu y Godfrey, 2001) propone una vista llamada de construcción argumentando que el modelo de Kruchten no cubre aspectos arquitectónicos que se presentan al llevar a cabo tareas como compilación, ensamblado y configuración del software para ser ejecutado en la plataforma física, por esto sitúa a su vista de construcción entre las vistas de desarrollo y física del modelo de Kruchten.

Otra propuesta situada precisamente en el lado opuesto de la implementación, es la vista llamada de decisiones que se plantea en (Dueñas, y Capilla, 2005), que se ubica como una

vista previa al modelo Kruchten, situándola entre los requisitos y este modelo, la considera como necesaria para modelar las decisiones arquitectónicas.

Existe también la propuesta de (Herzum y Sims, 1999) que no propone agregar una vista más a la de Kruchten, sino que hace un planteamiento dirigido hacia el desarrollo orientado por componentes, considerando a diferentes arquitecturas (ahí se utiliza el término 'diferentes arquitecturas', en lugar de vista), según su enfoque las cuatro más importantes a considerar en un sistema son: la arquitectura técnica, esta comprende a los elementos del ambiente donde se ejecutan los componentes, y todo los servicios o facilidades para ejecutar a sistemas de este tipo; la arquitectura de la aplicación, que incluye al conjunto de decisiones arquitectónicas, patrones, guías, y estándares para la construcción de sistemas desarrollados con componentes; la arquitectura de la administración del proyecto, que consiste en todo los conceptos, guías, y herramientas necesarias para administrar grandes proyectos de software; y la arquitectura funcional, que comprende la arquitectura donde reside la especificación e implementación del sistema. Estas otras propuestas pueden ser útiles en caso de modelar arquitectura de sistemas específicos, donde se requieran incluir los tipos de concerns que se destacan en cada una de estas propuestas descritas.

Después del breve análisis sobre los diferentes modelos para las vistas, las preguntas que surgen ahora son: ¿cuál es modelo más apropiado a usar para crear las estructuras de las vistas?, ¿Cuál es el modelo apropiado para representarlas? Estas preguntas entre otras serán afrontadas en la siguiente sección.

3.3 Anomalías detectadas en los modelos de las vistas, y propuesta del marco de trabajo para el modelado de las vistas

Anomalías detectadas en los modelos de las vistas

La diversificación en los modelos de vistas ha propiciado que no se cuente con un estándar para su modelado, diseño y documentación, pues el estándar 1471 de la IEEE que se tiene para las arquitecturas software, sólo abarca a las vistas dentro de un contexto genérico, dejando sin precisar cómo se modelan.

Entre las principales situaciones derivadas del análisis de la sección anterior se encuentran las siguientes:

- Dentro de los modelos de vistas revisados, existen por lo menos dos formas distintas de concebir a una vista arquitectónica, una forma es considerarla simplemente como una perspectiva de acuerdo a los elementos que se encuentran en las diferentes etapas de desarrollo del software, y la otra es tratarlas como estructuras que forman parte de la arquitectura de software. Los modelos que siguen al primer enfoque son el 4+1 vistas, el SIEMENS, y otros como los tratados en (Rozanki y Wood, 2006), que vinculan a las vistas débilmente con la arquitectura software, y las vinculan más con una arquitectura en términos genéricos, lo cual ha hecho que se orienten más al desarrollo de software, como el caso del modelo 4+1 que se adaptó al RUP, y se pierda el sentido de planeación que tiene la arquitectura de software.
- El segundo enfoque propuesto por el grupo del SEI, está totalmente orientado hacia la arquitectura de software por ser este mismo grupo dónde inició ésta disciplina. Sin embargo aún dentro de dicho grupo no

existe una total coincidencia, por ejemplo las vistas tratadas en (Clements 2002, b), difieren de las que se presentan en (Clements 2002) y (Clements 2003).

- Algunos modelos de vistas, coinciden con otros en algunas de sus vistas propuestas, con algunas diferencias en la notación y especificación de sus elementos, pero que pueden ser compatibles. Lo cual abre la posibilidad de usar un marco de trabajo común para el modelado de las vistas.
- El problema de la notación de las vistas no está totalmente resuelto, pues aunque en, (Clements et al., 2002), (Clements et al., 2003), se proponen varias alternativas de notación, aun hace falta precisar cómo incluir detalles de las propiedades de los elementos. En (Hofmeister, et al., 1999), se hace una propuesta de una notación orientada a objetos pero sólo es para las vistas que usan componentes y conectores. Muchas otras propuestas hacen uso de la notación UML 2.0 (Roh, et al., 2004),(Ivers et al, 2004),(Miao y Liu, 2005), (Khammaci, et al., 2006), pero otra vez como en los casos anteriores solo se comprende a las vistas con elementos de componentes y conectores.
- Aunque se usen ADL,s como un medio para describir y analizar a las arquitectura software, se necesita además una notación gráfica para poder representar de mejor manera la relaciones entre sus elementos. Algunos ADL,s hacen uso de una notación gráfica adicional (C2-Sadel, ACME, PRISMA), pero estos no representan a vistas sino a una arquitectura basada en componentes y conectores.
- El análisis y modelado de las vistas se ha centrado sólo en aquellas que usan como elementos a componentes y conectores. Solo en el caso del método de diseño dirigido por atributos llamado ADD

(Clements et al., 2003), se consideran a varios tipos de vistas.

- En todos los modelos se admite que hay una relación entre cada modelo de vista, pero en ningún caso se hace explícita, y en el caso del modelo 4+1 vistas se argumenta que una de sus vistas, la adicional, sirve para relacionar a las otras 4, pero esta sólo hace una verificación de los escenarios que se tienen contemplados en cada vista, sin que la relación entre las vistas se haga explícita, y por lo tanto no se cuenta con la traza de las relaciones entre las vistas.
- Al estar una vista enlazada con otra, se crea una relación de dependencia entre ellas, de tal manera que al modificar alguna de éstas por motivos de mantenimiento o evolución de la arquitectura, puede llegarse a perder la consistencia entre ellas por no existir un mecanismo que propague los cambios.

Marco de trabajo propuesto para el modelado de las vistas

Para poder hacer frente a estas anomalías descritas, en esta tesis se propone un marco de trabajo centrado en el modelado de las vistas y sus relaciones, usando como enfoque el desarrollo dirigido por modelos (DDM). Este marco de trabajo se fundamenta en los siguientes principios:

- Una vista es considerada aquí como la representación de una estructura de la arquitectura de software. Cada vista está compuesta de un modelo que determina el tipo de elementos, los tipos de relaciones que se establecen entre ellos y la notación correspondiente.
- Se usan dos vistas como las básicas, la modular, y la de componentes-y-conectores. La primera se encarga de vincular los concerns provenientes de los requisitos (del dominio del problema), con los

concerns del espacio de la solución, y la segunda se encarga de modelar los aspectos del espacio de la solución.

- Dentro de cada vista existen relaciones entre sus elementos que son tipificadas de acuerdo a los llamados estilos arquitectónicos, las cuales son tratadas como patrones para agrupar a los elementos. Esto permite la estructuración de cada vista básica en elementos de interés más específicos para poder llegar a formar otras vistas dentro de las básicas. De esta manera se pueden crear las vistas de los diferentes modelos antes expuestos y se evita la superposición entre las vistas de cada modelo.
- Los elementos usados para cada vista son de tipo arquitectónico, representan a elementos del sistema pero no pertenecen a este ambiente. Ellos contienen la especificación de los elementos que representan con fines de modelado y análisis de un sistema.
- La notación que se usa para los elementos y relaciones en cada modelo de vista está basada en el uso de de perfiles propuestos por el UML 2.0, ya que contiene la suficiente expresividad para representarlos. Además cumple el requisito de tener una representación gráfica en la arquitectura.
- Para la formación de los modelos de cada vista se sigue el enfoque DDM. A través de éste enfoque se modelan las vistas, se establecen, registran, y se mantienen las relaciones entre las dos vistas básicas y entre las demás vistas que se lleguen a formar dentro de ellas.
- El enfoque de DDM, también es el soporte para el análisis de las relaciones que se forman entre los elementos de cada vista, lo cual ayuda a determinar relaciones de dependencias u otras que se requieran.

Los elementos involucrados en este marco de trabajo son ilustrados en el esquema de la Figura 3.4. En esta Figura se aprecia la clase de concerns que se modelan con cada una de las vistas, y la relación que se establece entre las dos vistas básicas. Las vistas específicas son derivadas de las básicas de acuerdo y dirigidos por los concerns arquitectónico, que son propuestos por los actores involucrados. Todas estas vistas forman la arquitectura de software. El modelado de las vistas y sus relaciones es desarrollado siguiendo el enfoque del DDM.

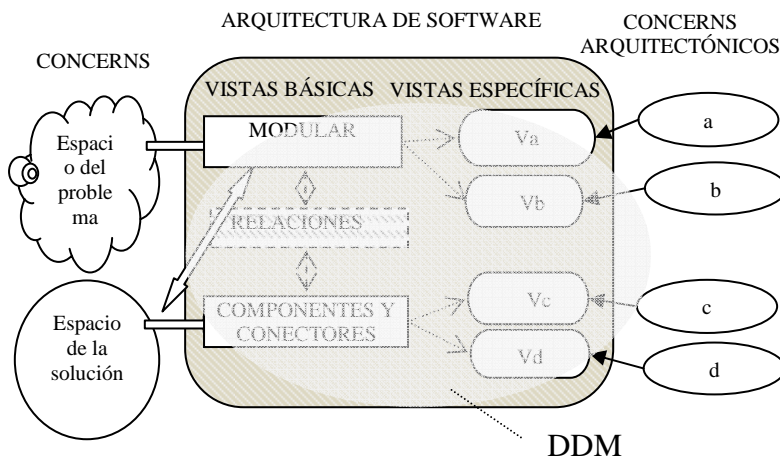


Figura 3.4. Propuesta del modelado de las vistas y sus relaciones, usando el enfoque de DDM

Mediante el enfoque del DDM, se hará posible realizar los principios descritos en el marco de trabajo, y con ello se dará solución a las anomalías detectadas.

El enfoque de DDM implica un proceso que usa y maneja modelos a diferentes grados de abstracción, lenguajes para expresarlos y herramientas para establecer relaciones entre ellos. Para poder comenzar con este proceso, es necesario

que se tenga claro y preciso lo que se pretende modelar. Por lo tanto a continuación se comienza por definir los elementos de las vistas, y se lleva a cabo la especificación de las dos vistas consideradas.

3.4 Bases para la especificación de los modelos de las vistas-arquitectónicas

El hecho de que existan varias propuestas de modelos sobre las vistas que pueden ser consideradas en una arquitectura, no significa que sea necesario establecer un estándar que determine el número y tipo de éstas, puesto que como es especificado en el estándar 1471 de la IEEE una vista se establece en función de la perspectiva que establecen los actores involucrados. Así que cada propuesta como las revisadas en la sección anterior, surge de un modelo que incluye perspectivas diferentes, de ahí que tampoco se puede decir que una propuesta sea mejor que la otra, simplemente que sus enfoques son distintos. Por estas razones no se pretende crear un nuevo modelo de vistas, sino más bien partir de los modelos existentes revisados anteriormente, para identificar a los elementos básicos y los tipos de relaciones que se crean en cada vista. Esto permitirá sentar las bases para poder llegar a formular un meta-modelo para cada vista, y con ellos llegar a generar modelos. Pero para poder construir los meta-modelos de cada vista, primero se tiene que empezar por definir la especificación del producto a obtener, es decir, especificar el modelo de cada vista. En el modelo de una vista se especifican los elementos, y reglas que determinan las relaciones entre sus elementos, y la notación que se usa para representarlos.

Para la especificación de los modelos de cada vista se ha considerado lo siguiente:

- i) Se parte del hecho de que una arquitectura de software (AS) está formada por varias estructuras, y que éstas son representadas mediante las vistas $V_1, V_2 \dots V_n$, de tal manera que:

$$AS = V_1 \cup V_2 \cup \dots V_i, \dots \cup V_n$$

Donde $V_i \cap V_j \neq \emptyset$ si existe *crosscutting* entre sus elementos, es decir si existen elementos que pertenezcan a más de una vista, y $V_i \cap V_j = \emptyset$, si no existe *crosscutting*. Siendo $i \neq j$. Nótese que esto es independiente de que existan relaciones de correspondencia entre una y otra vista.

Una vista V_i está dada por: $V_i = \{ E_{te,i}, R_{tr,i} \}$

Donde:

$E_{te,i}$ representa a los tipos de elementos (*te*) que constituyen a una vista i . Donde cada tipo de elemento está formado por un conjunto de propiedades que definen su naturaleza y su comportamiento, que son visibles a los demás elementos.

$R_{tr,i}$ representa las relaciones de un tipo *tr* que hay entre los tipos de elementos $E_{te,i}$, lo cual es expresado como:

$$E_{p,i} R_{tr,i} E_{q,i}$$

Esto indica que entre los elementos de dos tipos $E_{p,i}$ y $E_{q,i}$, (p , y q indican el tipo), existe una relación R de tipo *tr*, dentro de una vista i . La relación puede ser entre elementos del mismo tipo, entonces $p=q$, o entre elementos de tipos diferentes, es decir $p \neq q$.

Si un elemento e_i perteneciente a un tipo de elemento $E_{p,i}$ de una vista i (V_i) tiene una relación con otro elemento e_j del mismo tipo p ($E_{p,j}$), pero de otra vista j , (V_j), entonces se

dice que existe un *crosscutting* entre los elementos de las vistas V_i y V_j

- ii) Se toman en cuenta como estructuras básicas las tres que se proponen en SEI: la de *módulos*, la de *componentes-y-conectores*, y la de *asignación*. Fundamentalmente porque en ellas se consideran a tipos de elementos y tipos de relaciones arquitectónicas que se tienen en casi todos los sistemas (incluyen estructuras que modelan relaciones estáticas y de comportamiento), aunque no se descarta el uso de otras estructuras para sistemas muy especializados, como por ejemplo para una vista empresarial (Dijkman, et al., 2004). Con estas estructuras básicas como se verá, se pueden construir casi todas las vistas propuestas en los modelos revisados anteriormente.

Cada una de estas estructuras es representada mediante un tipo de vista básico, el cual tiene el mismo nombre que la estructura que representa. A su vez dentro de estos tipos básicos de vistas se pueden construir otras vistas, dependiendo del tipo de relaciones que se consideren en cada una y el rol que se les dé a sus elementos.

Adicionalmente con estas estructuras básicas se hace posible medir varios atributos de calidad. Por ejemplo mediante la estructura de módulos es posible medir que tan modificable es un sistema, con la estructura de componentes-y-conectores es posible medir la seguridad y desempeño del sistema a nivel arquitectónico.

- iii) Para expresar cada modelo de una vista, se usa UML 2.0 (OMG document formal/05-07-04, 2004), puesto que con él es posible expresar los diversos aspectos

arquitectónicos como se muestra en (Roh et al., 2004), donde se plantea su uso como un lenguaje de descripción arquitectónico(ADL), proponiendo un UML-ADL. En versiones anteriores del UML, se ha mostrado su utilidad para representar varias facetas de la arquitectura de software, como por ejemplo: haciendo equivalencias con los ADL,s formales como el C2- SADL, y el Wrigth (Robbins et al., 1998), para modelar la arquitectura (Medvidovic et al., 2002), y para expresar modelos de vistas como el SIEMS (Hofmeister, et al., 1999). Todo esto ha sido posible debido a que el UML puede ser extendido para nuevas funcionalidades por medio de un mecanismo llamado estereotipo.

La versión del UML 2.0, el mecanismo de prototipo ha sido elevado a un nivel de abstracción mayor llamado perfil el cual permite adaptar el UML para dominios específicos, esto es lo que aquí se hace específicamente para los meta-modelos, se crea un perfil para cada meta-modelo de cada vista básica arquitectónica de software.

Con estas consideraciones presentes se especifica a continuación lo que conforma el modelo para las vistas básicas: modular, y de componentes-y-conectores, fundamentando la semántica de sus elementos, y de sus relaciones.

3.5 Especificación del modelo de la vista modular

Esta vista se fundamenta en una estructura que representa una descomposición del sistema usando como criterio las funciones que se esperan que éste realice. La forma de descomponer al sistema mediante esta estructura es

adecuada para modelar a los sistemas complejos que se dan en el software, puesto que con ella se representan relaciones de jerarquía, lo cual es propio de los sistemas complejos, como se argumenta en (Simon y Ando, 1961) al observar la forma que adopta la estructura de varios sistemas complejos, *...estos toman la forma de una jerarquía*.

Por otra parte los módulos han constituido desde los 70's las primeras estructuras para descomponer y agrupar el software, en (Parnas 1972) el módulo se usó para agrupar las responsabilidades del sistema, después se le consideró como el elemento esencial para abordar la complejidad al ocultar los detalles de la implementación (Parnas 1984). Actualmente a nivel arquitectónico esta estructura (modular) se ha consolidado como algo fundamental (Clements et al, 2002), (Clements et al, 2003), (Dashofy et al, 2005). La estructura que se representa con esta vista responde a preguntas tales como las expuestas en (Clements et al, 2003):

- a) *¿Qué hace el sistema?*
- b) *¿Cuál es la responsabilidad principal asignada a cada módulo?*
- c) *¿Qué módulos son usados por otros?*
- d) *¿Qué módulos son permitidos usar por otros módulos?*
- e) *¿Qué módulos están vinculados con otros módulos por medio de relaciones de generalización o especialización?*

Estas preguntas permiten delinear lo que debe contener un módulo y dirigir las relaciones que se establecen entre este tipo de elementos. Para esto, primero se comienza con la definición de lo que aquí se considera como un módulo, la cual es tomada de la siguiente referencia:

Un **módulo** es una unidad de implementación de software que provee una unidad coherente de funcionalidad (Clements et al., 2002).

Aunque el módulo representa una unidad de implementación de software, en realidad al diseñar la arquitectura éste sólo incluye la especificación de las funciones que serán posteriormente diseñadas (en detalle) e implementadas en las siguientes fases del desarrollo, puesto que la intención esencial de la modularidad es incluir decisiones de lo que *‘deben ser’* los módulos antes que lleguen a convertirse en unidades de implementación (Parnas 1972). De esta manera se cumple uno de los propósitos de esta vista, el llegar a ser el anteproyecto del código (Clements et al., 2002).

Los atributos que se usan para definir a un módulo responden a las preguntas (a) y (b) que se plantearon anteriormente, estos atributos son:

- La función principal que desempeña, la cual define la razón del ser del módulo.
- Las responsabilidades que le son asignadas, las cuales responden a los requisitos detectados por la ingeniería de requisitos, y que se consideran deben estar agrupadas en un módulo. Estas pueden ser accesibles a través de interfaces.
- Las interfaces, que hacen posible establecer con precisión los roles que desempeñan los módulos dentro del sistema.
- La información adicional sobre las características de la implementación sobre la plataforma en que un módulo será desarrollado (por ejemplo, Java, .NET).

Por otra parte, las relaciones que se establecen entre los módulos se derivan de las últimas tres preguntas planteadas

anteriormente: (c),(d), y (e). La pregunta (c) da pie a la relación llamada *usa*, la pregunta (d) plantea conocer las relaciones que se establecen entre distintos niveles de abstracción llamadas *usa-capa*, y por último la pregunta (e) plantea dos tipos de relación: la *generalización* y la *especialización* entre módulos. A continuación se fundamentan y definen estas relaciones.

Relación usa, esta relación implica que un elemento tipo módulo usará las funciones de otro elemento de este tipo para completar sus funciones. Esto significa que el primer elemento (el que usa) tenga una fuerte dependencia del segundo (del usado), puesto que aunque son dos módulos independientes el segundo es complementario del primero.

Por ejemplo, sean *a* y *b* dos elementos tipo módulo, la relación usa se puede representar como: $a \rightarrow b$, indicando que el elemento *a* usa alguna o todas las responsabilidades asociadas a la función del elemento *b*. Si se quiere especificar qué es exactamente lo que *a* usa de *b*, se puede usar una interfaz en el módulo *b* que especifique la responsabilidad que se está accediendo de él. Con esta relación es posible conocer qué módulos le son necesarios a otro módulo para analizar el grado de dependencias que existe entre módulos dentro de una arquitectura.

Relación usa-capa, en esta relación se usa el mecanismo de abstracción llamado *capa*, que permite hacer una separación de tareas agrupando a los módulos en diferentes niveles de abstracción, de tal manera que los elementos de una capa se refinan en las funciones que en su conjunto brindan los elementos de otra capa. Las capas equivalen a *maquinas virtuales*, tal como se declara en (Clements et al., 2002, pag. 93) ‘..... las capas intentan ser una maquina virtual... una máquina virtual es una colección de módulos

que en su junto proveen un conjunto de servicios que otros módulos usan...’.

El término de *maquina virtual* (también llamado *maquina abstracta*) fue introducido por (Parnas, 1974) quién surgiere que estas son organizadas en niveles de abstracción, de mayor nivel a menor nivel de abstracción. Esto supone una jerarquía en los niveles de una capa, de mayor a menor nivel de abstracción, donde una capa de nivel superior usa a otra que es de menor nivel, las capas de nivel inferior son las que están más cercanas al hardware. Se supone que los elementos de las capas inferiores soportan a los de la capa superior inmediata. Esta relación entre capas se representa en forma esquemática en la Figura 3.5, la línea indica la separación entre los niveles de abstracción. En este esquema, los elementos de mayor nivel de abstracción están en la capa *a*, los cuales usan a los de la capa *b* que tienen un nivel de abstracción inferior. El hecho de que los elementos de una capa estén encapsulados como máquinas abstractas, hace posible la portabilidad de estos entre sistemas.

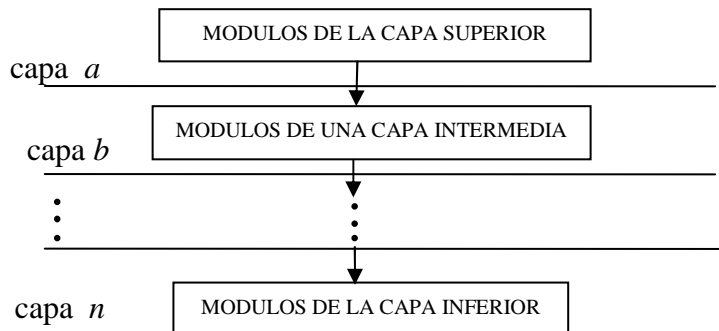


Figura 3.5 Representación esquemática de una relación de usa-capas

Relación de descomposición, esta relación indica que un módulo está compuesto de varios módulos más, creándose una división de un sistema en subsistemas. El objeto de esta división es dar una vista de arriba-a-abajo del sistema, para entender cómo los módulos pueden ser agrupados (o desagrupados), y poder después organizar la asignación de los productos de software a desarrollar al equipo de desarrollo. En la metodología orientada a objetos esta relación es conocida como *es-parte-de*. Esta relación puede ser representada esquemáticamente en dos formas, como se muestra en la Figura 3.6, en ambos casos se indica que el módulo *X* está descompuesto en (compuesto por¹) los módulos *a*, *b* y *c*.

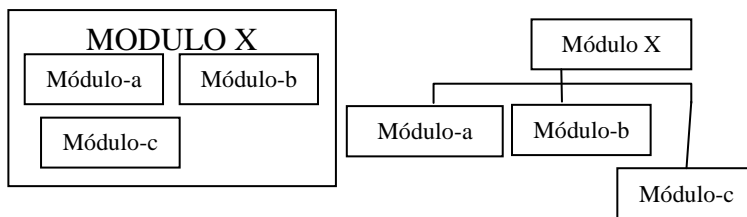


Figura. 3.6 Representación esquemática de la relación de *descomposición*

Relación de generalización, esta relación concentran en un elemento las propiedades comunes que comparten varios elementos de módulos, lo cual es conocido como generalización. En esta relación se crea una estructura jerárquica donde el elemento padre posee las características comunes de los descendientes, estos últimos a su vez agregan otra información que los hace diferentes, por eso también puede verse como una relación de especialización donde un elemento puede ser especializado agregándole o restringiéndole propiedades.

¹ La composición y descomposición en realidad son las 'dos caras de la misma moneda'

Finalmente se establece la notación basada sobre UML 2.0 para los elementos específicos y relaciones que se consideran para esta vista, lo cual se muestra en la Tabla 3.1 y Tabla 3.2 respectivamente. En ellas se denota que el modelo que se usa para su representación será el de clases con estereotipos, incluyendo sus relaciones.

Tabla 3.1 Notación en UML 2.0, de los tipos de elementos de la vista de módulo.

Tip o de ele me nto	Atributos	Notación	Descripción
Módulo	Función Responsabilidades Información- implementación	<div> <div><<Módulo>></div> <div>identificación</div> <div>función</div> <div>responsabilidades</div> <div>información- implementación</div> </div>	<p>Elemento básico de esta vista cuyos atributos privados contienen:</p> <p>la función principal que realiza, derivada del dominio de la aplicación.</p> <p>Las responsabilidades son un conjunto de sub-funciones (funciones más específicas) que son(o serán) implementadas, conteniendo información sobre el tipo de elemento de software que se implementa (proceso o manejador de base de datos)</p> <p>Información adicional sobre la plataforma en que se implementará.</p>
Subsistema	registro_modulos	<div> <div><<Subsistema>></div> <div>identificación</div> <div>-</div> <div>registro_modulos</div> </div>	<p>Es un tipo particular de módulo, que está formado por elementos básicos de módulos, manteniendo en el atributo <i>registro_modulos</i>, información sobre el número de módulos del sub-sistema y de las dependencias entre ellos (módulos que usan a otros).</p>

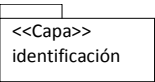

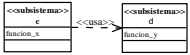

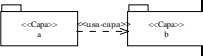

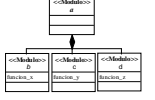


Capa			Agrupar a subsistemas y módulos, dentro de capas
------	--	---	--

Tabla 3.2 Notación en UML 2.0, de las relaciones entre elementos de la vista de módulo.

Relación	notación	Tipo de elementos que relaciona	Cardinalidad	Ejemplos
usa		módulos- módulos, subsistemas- subsistemas	1 – n	
usa-cap		capa-cap	1 – 1	
descomposición		Subsistema- módulos	1 - n	
generalización		Módulos	-	

3.6 Especificación del modelo de la vista componentes-y-conectores.

La vista de componentes-y-conectores es una perspectiva que muestra las estructuras que adoptan los elementos de software usados en tiempo de ejecución. En esta vista se modelan las potenciales interacciones entre las entidades ejecutables de software que se manifiestan al operar el sistema. El modelado a nivel arquitectónico de esta vista consiste en describir cómo estas interacciones llegan a configurarse, el tipo de flujos de comunicación que se establecen entre estas entidades, y el rol que desempeña cada elemento.

Mediante la estructura básica de esta vista se pueden derivar otras más específicas, con el objeto de describir aspectos más particulares, como por ejemplo aspectos de concurrencia y sincronización (Kruchten, 1995), que pueden ser descritos mediante la vista específica de procesos de comunicación (Clements et al., 2002). Esto es posible gracias a los diferentes roles que desempeñan sus dos principales elementos, el componente y el conector.

Esta vista es la que más se ha tratado hasta la fecha dentro de la arquitectura de software, ya que los componentes y conectores fueron considerados desde la concepción de la arquitectura de software. En (Perry y Wolf 1992) se menciona a los componentes como elementos para el procesamiento, y a los conectores se les designa como elementos de interconexión. En (Shaw y Garlan, 1996) se les considera ya como los dos elementos básicos de una arquitectura, situándolos en diferentes niveles de abstracción. Actualmente, estos elementos son el soporte de casi todos lenguajes desarrollados para la descripción arquitectónica (ADL's), desde lenguajes diseñados ex profeso para la arquitectura, como los analizados en

(Medvidovic y Taylor, 2000), o lenguajes genéricos como el UML 2.0 que se han adaptado para tal fin (Roh, et al., 2004), hasta aquellos lenguajes diseñados con una orientación específica de la arquitectura como PRISMA (Perez, 2006) que está diseñado para trabajar con *aspectos*.

Con esta vista de componentes-y-conectores se pretende dar respuesta a preguntas tales como:

- a) *¿Cuáles son los principales componentes en ejecución y como ellos interactúan?*
- b) *¿Cuáles son los principales almacenes de datos compartidos?*
- c) *¿Qué partes del sistema están replicadas?*
- d) *¿Cómo hacer que los datos progresen a través del sistema?*
- e) *¿Qué partes del sistema pueden ejecutarse en paralelo?*
- f) *¿Cómo puede la estructura del sistema cambiar cuando se ejecuta?*
- g) (Clements et al., 2003)

Para ver cómo esta vista puede dar respuesta a las preguntas planteadas anteriormente, se comienza por definir a sus dos principales elementos, al componente y al conector.

El Componente

Un componente de software es definido por (Herzum y Sims, 1999, pag. 7) como una construcción auto-contenida de software que tiene un uso definido, con interfaces para el tiempo de ejecución, y puede ser desarrollado en forma autónoma.... En esta definición se destacan características genéricas de un componente: su autonomía, que desempeña una función específica, y que cuenta con interfaces para interconectarse en tiempo de ejecución. Una

definición también genérica dada por (Hopkins 2000), agrega una característica más, la que un componente es un elemento ejecutable. Sin embargo estas definiciones se refieren a componentes de software en general, por lo que para ubicarse dentro del ámbito de la arquitectura de software se toma la definición dada en (Clements et al., 2002), donde se describe lo que representa un componente dentro de este ámbito de la siguiente manera:

Los componentes son:

‘Los principales elementos computacionales ejecutables, y los almacenes de datos que intervienen en tiempo de ejecución’.

Como se puede observar un *componente* dentro de la arquitectura representa a dos tipos de elementos de un sistema (a elementos ejecutables y a los almacenes de datos¹), quienes juegan un rol importante en el ambiente de ejecución. Algo importante que vale la pena aclarar sobre el elemento computacional al que se hace referencia en esta definición, es que a nivel arquitectónico no interviene ni la metodología ni tecnología con que éste haya sido hecho(o se vaya hacer), por lo tanto un elemento puede ser un objeto o un componente desarrollado bajo cualquier plataforma (CORBA, COM, DCOM, EJB²), o un elemento diseñado con un software especializado. Lo importante aquí es que sea ejecutable y que además sea un elemento principal dentro del sistema. El calificativo de ser *principal*, aunque es muy relativo, indica que sólo interesan elementos ejecutables que tengan una injerencia significativa en el sistema, descartando a muchos elementos que aunque útiles no repercuten en la arquitectura.

¹ En la cita referida de (Clements et al., 2002), no se precisa lo que se entiende por ‘almacén de datos’, por lo que aquí es tomado como una bases de datos, que es a lo se describe en otras secciones de la cita referida.

² CORBA: Common Object Request Broker Architecture del Object Management Group; DCOM, COM (Distributed) Component Object Model de Microsoft; EJB Enterprise JavaBeans de SUN.

Por otro lado, el término *componente* como un elemento arquitectónico, hace alusión a las características señaladas en el desarrollo orientado a componentes, en el sentido de ser un elemento reutilizable, y que hace posible que el sistema (en este caso la arquitectura) evolucione (Hopkins 2000), gracias a la relativa independencia que se tiene para su diseño y desarrollo. Es decir un *componente* arquitectónico se comporta como un componente de software.

Puesto que el propósito de un *componente* arquitectónico¹ es modelar el comportamiento de los elementos del sistema que se usan en tiempo de ejecución, se debe incluir en su especificación la información necesaria que permita describirlos.

La información para un componente genérico, es decir para aquellos que no tienen una especialización en su comportamiento, comprende básicamente lo siguiente:

- La tarea o función computacional en tiempo de ejecución que realiza, la cual lo identifica como tal, especificando también si se trata de un elemento ejecutable o un almacén de datos.
- Las interfaces de comunicación, que consisten en los puertos de entrada y salida por los cuales recibirá y enviará información desde y hacia los demás componentes. Dichos puertos son los puntos por donde se comunica con los demás elementos.
- Información y operaciones internas necesarias para hacer manipulaciones de lo que se modela con él. Ejemplo de estas operación se encuentran en los componentes definidos por algunos ADL's , por

¹ En lo sucesivo se usará sólo el término *componente* para hacer referencia al elemento arquitectónico

ejemplo en Wrigth (Allen, 1997) se incluye las *computaciones*, y en PRISMA (Pérez 2006) las *evaluaciones*.

Un componente genérico es especializado cuando se le asocia con alguna de las funciones que forman parte de ciertas estructuras arquitectónicas identificadas como estilos, las cuales han sido especificados en (Shaw et al , 1995), (Shaw Mary y Garlan David, 1996), y (Clements et al., 2002). Entre las estructuras (o estilos) más comunes que emplean componentes están las siguientes: filtro-tubería, cliente-servidor, igualdad-de-servicios, escritor-lector, datos-compartidos, (estas estructuras serán abordadas en detalle en las relaciones entre componentes y conectores, ya que la atención en este punto es sobre los componentes). Dichas estructuras especializan al componente mediante la especificación de la función computacional particular que debe realizar dentro de ellas, el tipo y número de puertos que debe usar, y los atributos adicionales que debe contener.

Los tipos de componentes que aquí se proponen se muestran en la Tabla 3.3, con la descripción de su función computacional correspondiente, indicando el tipo de puertos y el número de estos, y los atributos adicionales que ellos especializan. Un puerto puede ser de tipo entrada o de salida, por el de salida se comunican los servicios que un componente ofrece a los demás, y por el de entrada se reciben las solicitudes de esos servicios o lo que el mismo componente requiera. Deben existir tantos puertos de cada tipo como servicios diferentes se ofrezca o se requieran, el número de estos dependerá de la función computacional de cada tipo de componente, aunque sólo es posible determinar su mínimo necesario o en algunos casos asociarlo a los tipos de servicios que ofrece o recibe.

Tabla 3.3 Tipos de componentes, sus funciones computacionales y atributos adicionales

Tipo de componente	Función computacional	Número de tipos de puertos	Propiedades y atributos adicionales
		npe: número de puertos de entrada. nps: número de puertos de salida.	
Filtro	Transforman información mediante un criterio de filtrado	npe ≥ 1 nps ≥ 0	-criterio de filtrado o función de transformación. -artefactos
Cliente	Solicitud de servicios de datos, comunicación, de páginas web, etc.	npe = nts nps = nts Donde nts es el número de	-tipos de solicitudes -solicitudes -artefactos tipos de solicitudes
Servidor	Proporcionan servicios especializados, para manejo de datos, de comunicación, servicios web, de nombres de dominio, etc.	npe = ntsp, nps=ntsp Donde ntsp es el número de tipos de servicios proporcionados	-tipo de servidor -tipos de servicios proporcionados -artefactos
Dualidad-de-servicio	Es un elemento que puede cumplir tanto la función de cliente como la de servidor, de ahí su nombre. Lo cual es útil para ambientes distribuidos.	nps = nsm, npe=nsm Donde nsm es el número que resulte mayor del número de tipos de servicios, o el número de tipos de solicitudes	-tipo de servidor -tipos de servicios proporcionados -tipo de solicitudes -solicitud - artefactos
Unidad-concurrente	Puede ser una tarea, un proceso o un hilo de control	npe ≥ 2 nps ≥ 2 El número máximo de npe y de nps dependerá de lo que se llegue a	-tipo de elemento (tarea, proceso, hilo de control) -información que se comunica, incluyendo: -identificación del componente al que fluye

		comunicar	-operaciones que se realiza con la información -artefactos
Datos-compartidos	Repositorios de datos o los denominados pizarra (Shaw Mary y Garlan David, 1996, pag. 49)	npe \geq 1 nps \geq 1	-tipo de datos compartidos -forma de acceder ¹ propiedades sobre el desempeño -artefactos

En la Tabla 3.3 referida, a los tipos de componentes filtro, unidad-concurrente y datos-compartidos se les ha fijado el mínimo número de puertos que deben tener (usando el operador \geq), por ejemplo el tipo filtro por lo menos debe tener un puerto de entrada, que es por donde le llega la información que debe filtrar. Para el caso de los demás tipos de componentes, se establece el número de sus puertos en función de las propiedades de sus atributos, mediante la asignación (=) de sus valores, así por ejemplo para el tipo cliente el número de sus puertos de entrada como los de salida dependen del número de tipos de servicios que solicita.

Los atributos corresponden a información interna necesaria para modelar el comportamiento de cada tipo de componente. Los artefactos son unidades adicionales para la ejecución del componente.

Por último, hay que señalar que estos tipos de componentes pueden ser compuestos, es decir estar formados por otros componentes, por ejemplo en PRISMA (Perez 2006) a un componente compuesto se le designa como sistema.

¹ Las formas de acceder a los datos que se modelan aquí encapsulan los detalles de acceso físico en un solo componente, exponiendo solo operaciones lógicas, ya que el conocimiento del modelo de datos subyacente es mantenido en el código de las aplicaciones. Esto es conocido técnicamente como *data accessor* (Nock, 2003).

El Conector

Con respecto al *conector*, segundo elemento básico de esta vista, su importancia dentro de la arquitectura es manifestada desde los inicios de ésta al ser considerado como un elemento de primera clase (Shaw 2003). En (Shaw et al., 1995) a un conector se le describe como el lugar donde se definen las relaciones entre los componentes, esto significa que sirve como enlace entre los componentes, coordinando las interacciones entre ellos y estableciendo las reglas que las gobiernan, tal como se señala en (Shaw y Garlan, 1996). Recientemente un conector ha sido definido por (Clements et al., 2002) de la siguiente manera:

‘Un conector es una ruta de ejecución de la interacción entre dos o más componentes’

Al considerarse a un conector como una ruta, significa que en ella se tiene que modelar las formas de vincular a los componentes, especificando las reglas de interacción y la forma de coordinarse. De hecho un conector podría compararse con la ‘medula espinal’ del centro nervioso de esta vista de componentes-y-conectores, donde concurren las conexiones de los componentes más importantes.

La forma en que dos o más componentes se vinculan depende del tipo de interacción que sus servicios produzcan y demanden. Los servicios pueden interactuar mediante llamadas de procedimientos, eventos, acceso a datos, o sólo mediante un simple flujo de comunicación. Cada una de estas formas de interacción implica que los conectores se tengan que especializar, incorporando propiedades adecuadas para modelarlas. Desde los inicios de la arquitectura se propusieron a cuatro tipos de estos, para llamadas a procedimientos, para manejo de eventos, de

protocolos para bases de datos y tipo tubería (Shaw y Garlan, 1996, pag. 20).

A nivel general, dentro del software se han detectado varios tipos de conectores, en (Metha et al., 2000) se ha hecho una clasificación de ocho tipos de estos de acuerdo a los servicios de interacción que prestan, estos son llamadas a procedimientos, eventos, acceso-a-datos, enlaces, flujos, arbitro, adaptador y distribuidor. Cada uno tiene diferentes formas de coordinar las interacciones, a continuación se describe lo que realiza cada tipo de conector:

- El tipo *llamadas-a-procedimientos*, establece una coordinación a través de técnicas de invocación, y transfiere datos a través del uso de parámetros.
- El tipo *eventos*, detecta los eventos que le acontecen a un componente, y los mensajes generados por estos eventos son enviados a los componentes involucrados para que ellos los procesen.
- El tipo *acceso-a-datos*, coordina el acceso a componentes de tipo datos-compartidos, pudiendo además prestar otros servicios como conversión de la información que se recibe o envía.
- El tipo *enlace*, sólo proporciona un canal para que la información de los componentes fluya a través de él.
- El tipo *flujo*, coordinan la ejecución de la transferencia de grandes cantidades de información entre procesos autónomos, como por ejemplo entre componentes clientes y servidores.

- El tipo *árbitro*, coordina a los componentes que requieren conocer el estado y las necesidades cada uno del otro, arbitrando los posibles conflictos que se presenten por el uso de recursos compartidos, de ahí su utilidad en sistemas concurrentes.
- El tipo *adaptador* realiza la coordinación a través de las facilidades que se dan para poder comunicar componentes que no han sido diseñados para interoperar entre ellos, debido posiblemente a que pertenezcan a un ambiente heterogéneo con plataformas de computación diferentes, las facilidades de este conector consisten en realizar las transformaciones para hacer corresponder a los componentes, es decir para que puedan tener interacción.
- El tipo *distribuidor* identifica trayectos de interacción, y coordina la información que fluye a través de estos trayectos. Este tipo de conectores nunca actúan solos, son conectores auxiliares de otros tipos como los de flujo y llamadas de procedimientos, quienes les proporcionan los trayectos y la información respectiva.

A nivel arquitectónico, los tipos de conectores descritos anteriormente pueden ser usados para la interacción entre componentes, como lo han hecho los diferentes ADL's diseñados. Por ejemplo ocho de los diez ADL's (ACME, Aesop, C2, MetaH, Rapide, SADL, UniCon) analizados en (Medvidovic y Taylor, 2000) utilizan algunos de los tipos de conectores descritos para la interacción entre los componentes. Cada lenguaje define a los conectores de forma diferente, sin embargo desde sus inicios (Shaw et al., 1995), se han identificado las características básicas que estos deben tener para poder modelarlos a este nivel.

La composición básica de un *conector* de acuerdo a (Clements et al., 2002), incluye:

- Las reglas de interacción, lo cual está determinado por su tipo y por los tipos de componentes que llega a enlazar.
- Los protocolos de estas interacciones cuyas entidades visibles son los roles que se forman al enlazar a los componentes. Los roles son los puntos de enlace de los conectores que enlazan a los puertos de los componentes.
- Propiedades adicionales, que son atributos adicionales requeridos para llevar a cabo la interacción.

Las reglas de interacción de los conectores se fundamenta en la naturaleza de su tipo, lo cual fue descrito anteriormente, estas reglas se hacen explícitas al relacionarse con los tipos de componentes. Pero antes de ver cómo se llegan a relacionar los componentes y conectores, es necesario conocer los posibles roles y propiedades adicionales que cada tipo de conector tiene. En la Tabla 3.4 se muestra cada tipo de conector describiendo la naturaleza de su interacción de acuerdo al tipo de componentes a los que se puede enlazar, lo que determina sus posibles roles y el número de estos, estos roles son los básicos a considerar porque pueden darse otros de acuerdo a los tipos de componentes que vincule. Las propiedades adicionales de cada tipo de conector, están formadas por atributos que requieren ser especificados, pero puede existir otras propiedades de comportamiento que resulten necesarias.

Tabla 3.4 Tipos de conectores con sus componentes asociados, sus roles y atributos

Tipo de conector	Tipo de componentes asociados	Tipo y número de roles	Atributos adicionales
Llamadas a procedimientos	Enlaza a componentes tipo proceso, pero también es útil para componentes tipos cliente y servidor.	<i>invocación</i> : recibe la llamada para ejecutar a un procedimiento. Su número será igual al número procedimientos existentes. <i>envío</i> : envía el resultado del procedimiento ejecutado. Su número será igual o mayor que cero (cero por si no se regresa ningún resultado).	-identificación de procedimientos -parámetros -información
Tubería	Corresponde al tipo de conector de enlace, sirve como conductor por donde circula información, usualmente conecta a tipos de componentes filtro, pero puede ser usado para cualquier otro.	<i>origen</i> : hace la referencia al componente fuente. <i>destino</i> : hace referencia al componente al que dirige la información. El número de estos será uno, pues sólo conecta a un componente con otro	-capacidad del buffer
Eventos	Vincula a componentes tipo proceso que usan como activación a eventos, en lugar de llamadas a procedimientos	<i>receptor</i> : capta la emisión de eventos, y los posibles cambios de estado. <i>notificador</i> : envía la notificación de los cambios de estado a los componentes. El número de ambos tipos de roles debe ser igual o mayor al número de tipos de eventos.	-tipo-de-evento -información -estado

Acceso-a-datos	Enlaza componentes tipo datos-compartido (donde están los datos) con otro componentes como servidores, o unidades-concurrentes.	<i>escritura</i> : inserción de datos <i>leer</i> : consultar datos <i>actualiza</i> :modificar datos. El mínimo número de cada estos debe ser uno	-tamaño-buffer
Flujo	Enlaza a componentes de tipo cliente con los de tipo servidor, sobre todo cuando estos últimos son servidores de datos o de tipo web	<i>solicitud</i> : envían las solicitudes de servicio al servidor. Su número debe ser igual al número de tipo de solicitudes. <i>entrega</i> : repartir a los clientes lo generado por los servidores	-capacidad
Arbitro	Vincula a componentes tipo unidades concurrentes, arbitrando los procesos e hilos de ejecución	<i>detectar</i> : detectar solicitudes de acciones de componentes. <i>acción</i> : realiza acciones en función de disponibilidad del recurso(componente)	-estado -proceso -hilo
Adaptador	Vincula a componentes de tipos diferentes	<i>adaptar</i> : establece la correspondencias entre el componente origen con el destino <i>corresponder</i> : vincula al elemento que se adapta con el (o los) que le corresponde(n).	correspondencias
Distribuidor	Enlaza a componentes de tipos diferentes y a otros conectores	<i>elegir-ruta</i> : ubica la ruta de acuerdo a los itinerarios existentes <i>repartir</i> : distribuye información por la ruta seleccionada No existe limite en su número	-itinerarios

La relaciones entre componentes-conectores-componentes

Para especificar las posibles relaciones que se forman entre los componentes por medio de los conectores, se usará como punto de referencia los patrones o estilos que ya fueron mencionados anteriormente los cuales son: filtro-tubería, cliente-servidor, igualdad-de-servicios, escritor-lector, y datos-compartidos, que aunque no son los únicos, son básicos para formar relaciones más complejas. En la especificación de estas relaciones se define el tipo y número de componentes y los tipos de conectores asociados con ellos. Se hace una representación informal de los elementos con él objeto de representar su topología, para un componente se usará un cuadro y para un conector un ovalo, las flechas indicarán los vínculo de los puertos al conector.

Filtro-tubería, esta relación usa dos tipos de componentes *filtro*, enlazados mediante un conector tipo *tubería*, de ahí el nombre de dicha relación. Aunque el componente tipo *filtro* puede ser en realidad un tipo de componente compuesto, es decir estar formado internamente por otro tipo de relaciones. La información fluye en un solo sentido y la *cardinalidad* es de uno a uno, es decir a un componte filtro le corresponde sólo otro componente *filtro*. Esquemáticamente se representa con una topología lineal como se ilustra en la Figura 3.7.

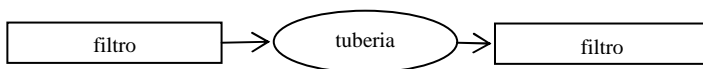


Figura 3.7 Representación esquemática de la relación de filtro-tubería

Cliente-servidor, esta relación es formada por uno o más componentes de tipo *cliente* que solicitan algún servicio de un *servidor*, aquí el conector que los enlaza puede ser de tipo *flujo*, y los servidores pueden estar conectados a componentes tipo *datos-compartidos*. También se puede usar adicionalmente a un conector de tipo *distribuidor* para replicar datos o servicios distribuidos.

La *cardinalidad* es de muchos a uno, ya que muchas instancias del componente *cliente* se comunican con una instancia de un *servidor*. La información en este caso fluye en ambos sentidos, de esta manera una ruta de interacción se forma enlazando los puertos de salida de cada cliente con los de entrada del servidor a través del rol *solicitud* del conector, y los puertos de salida del servidor se enlazan con los de entrada de los *clientes* a través del rol *entrega* del conector. El esquema de esta relación se muestra en la Figura 3.8.

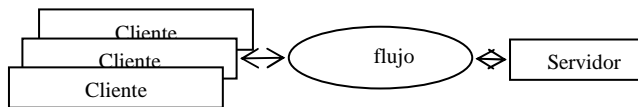


Fig. 3.8 Representación esquemática de la relación de cliente-servidor

Igualdad-de-servicios, esta relación deriva su nombre precisamente del hecho de que el tipo de componente usado *dualidad-de-servicio*, puede desempeñar tanto la función de *cliente* como la de *servidor*. Las instancias de este componente se vinculan entre sí a través de las instancias de un conector de tipo flujo y uno tipo distribuido, este último es usado precisamente para distribuir la carga de los servicios solicitados entre los diferentes componentes.

El esquema de esta relación se muestra en la Figura 3.9, como ahí se aprecia la ruta de interacción está formada por

los conectores *flujo* y *distribuidor* que se interconectan para coordinar los enlaces.

La *cardinalidad* entre los componentes es de muchos a muchos (n a n), ya que un componente puede requerir el servicio de varios componentes, y un componente ofrece a más de uno. En el caso de los conectores su *cardinalidad* es de uno a uno, puesto que se requiere que estos actúen uno a uno para poder coordinar los enlaces.



Figura 3.9 Representación esquemática de la relación *igualdad-de-servicios*

Publicador-consumidor, esta relación indica que los componentes son ejecutados mediante un conjunto de eventos que se llegan a generar (*publicar*), estos generan acciones hacia otros que deben ser notificados para que cambien de estado (los *consumidores*). Por lo tanto los componentes que se usan en esta relación son genéricos solo con la característica de corresponder a elementos ejecutable, no a *datos-compartidos*. La principal función en esta clase de relación recae en el conector, ya que él realiza la tarea de detectar los eventos y distribuirlos en los componentes correspondientes, de ahí que se requiera el uso de dos tipos de conectores el de tipo eventos y el de tipo distribución, el primero hace la función de *publicador* y el segundo el de *consumidor*. En algunos casos se contempla el uso de un componente que haga estas funciones (Clements et al, 2002) en lugar de usar conectores, pero aquí se considera que es mejor el uso de estos dos tipos de conectores, ya que al fin de cuentas ellos tienen definidos

este tipo de interacciones. En la Figura 3.10, se muestran el esquema de esta relación.

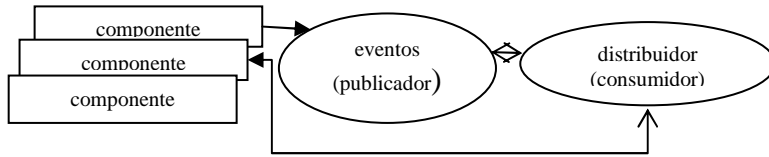


Figura 3.10 Representación esquemática de la relación *publicador-consumidor*

La *cardinalidad* entre componentes es de mucho a muchos, ya que son varias las instancias que publican y también varias las consumidoras. Por otra parte, la *cardinalidad* entre los conectores es de uno a uno, pues sólo es requerida una instancia de cada una de ellas para llevar a cabo la coordinación entre la *publicación* y la *distribución*.

La ruta de interacción se forma conectando los puertos de salida de los componentes (que emiten los eventos) con el rol receptor del conector eventos, éste último a su vez se enlaza con su rol notificador al rol elegir-ruta del conector distribuidor, para que los cambios de estados provocados por los eventos sean notificados a los componentes mediante el rol repartir de este último conector, que se vincula con los puertos de entrada de los componentes, cerrando con esto la ruta de interacción.

Datos-compartidos, esta relación se forma entre componentes genéricos y componentes del tipo *datos-compartidos*, usando como conector al tipo *acceso-a-datos*, tal como se muestra en la Figura 3.11. Aquí la relación entre componentes genéricos no interesa, puesto que lo importante es resaltar cómo éstos acceden a los datos del repositorio. Existen por lo menos dos maneras diferentes en que se provocan las interacciones con los datos. Una de ellas

es por medio de mecanismos disparadores (*triggers*) que se encargan de notificar las acciones que los componentes quieren hacer con los datos, entonces los datos compartidos se comportan como una *pizarra*, porque cada interacción puede actualizarlos. La estructura *pizarra* fue definida en (Garlan and Shaw 1993) para componentes que contenían fuentes de conocimiento y que interactúan aisladamente tomando el conocimiento de los *datos-compartidos*, ya que esto se ha aplicado en sistemas que requieren interpretaciones complejas de señales de procesamiento, tales como reconocimiento de patrones. Pero actualmente la *pizarra* es generalizada para todos aquellos sistemas que usen un administrador del flujo de trabajo (Clements et al., 20002).

La otra forma de hacer interacciones es sólo por consultas que se activan por los propios accesos de datos, en este caso los *datos-compartidos* se comportan como un repositorio.

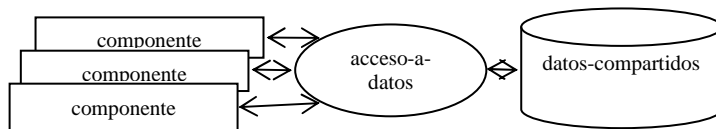


Figura 3.11 Representación esquemática de la relación datos-compartidos

Comunicación-de-Procesos, esta relación se forma por componentes de tipo unidad-concurrente que se enlazan mediante conectores de tipo arbitro. A través de sus enlaces se comunican datos y/o acciones a realizar que por los componentes a los que se comunican. Además, también se pueden incluir componentes tipo datos compartidos cuando los procesos requieran acceso a información persistente. La

manera en que se enlazan estos elementos se muestra en la Figura 3.12.

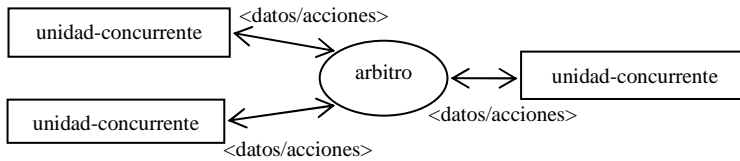


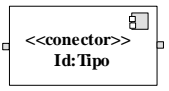
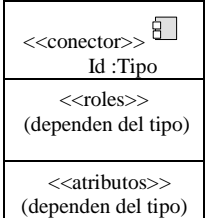


Figura 3.12 Representación esquemática de la relación comunicación-de-procesos

Notación

Finalmente se establece la notación basada sobre UML 2.0 para los elementos específicos de esta *vista*, lo cual se muestra en la Tabla 3.5. El modelo que se usa para su representación está fundamentado en el análisis hecho en (Ivers et al, 2004) para la representación de componentes y conectores, con dos variantes, una es la de usar estereotipos para representar a los componentes y conectores, y la otra es usar dos tipos de notaciones, una de caja negra y la otra notación de caja blanca (OMG document formal, 2004). La caja negra sirve para hacer una representación gráfica y la otra para la especificación de cada elemento.

Tabla 3.5 Notación en UML 2.0, de los elementos de la vista de componente-conector.

Elemento	Atributos	Notación Como ‘caja negra’ Como ‘caja blanca’
componente	Tipo Número de puertos de entrada Número de puertos de salida Atributos adicionales Artefactos	<div>  <p>Donde: Id: identificación de una instancia de un tipo Tipo: indica uno de los tipos descritos en la tabla 3.3.</p> <ul style="list-style-type: none"> □ puerto de entrada ■ puerto de salida </div> <div>  </div>
conector	Tipo Tipo de conectores Roles Atributos adicionales	<div>  <p>Donde: Id: identifica la instancia de un tipo de conector. Tipo: indica uno de los tipos descritos en la tabla 3.4</p> <ul style="list-style-type: none"> □ indican los roles, los cuales serán etiquetados con su nombre. </div> <div>  </div>

En el caso de las relaciones de esta vista, no existe una notación explícita como en el caso de la *vista modular*, ya que la notación de estas relaciones se fundamenta en la de sus elementos. Por lo tanto se muestra a través de ejemplos diferentes formas de relacionar a los componentes siguiendo lo ya especificado en las relaciones de esta vista Figura 6.13.

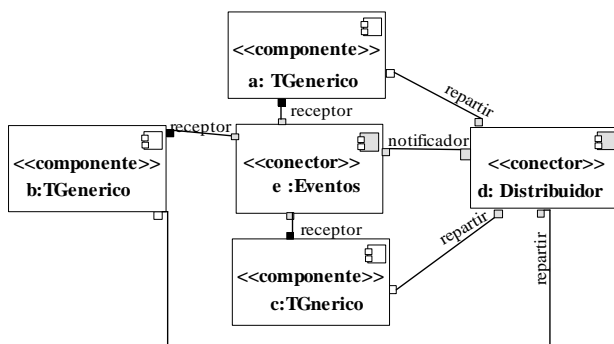
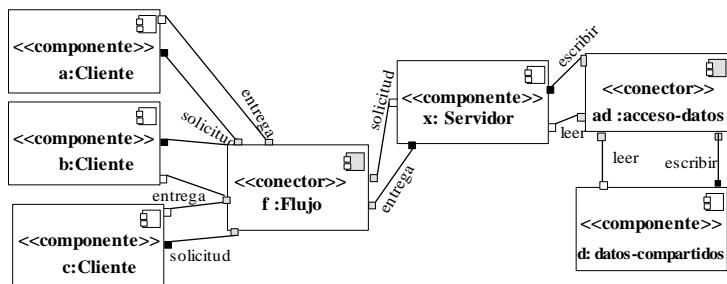
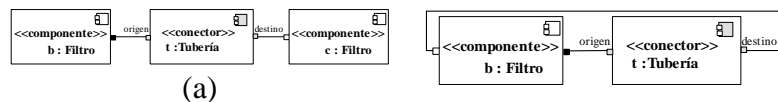


Figura 3.13 Representación en notación UML 2.0 de las relaciones:
 (a) Filtro-tubería, con dos componentes; (b) filtro-tubería, con un solo componente; (c) cliente-servidor combinado con la relación datos-compartidos; (d) publicador-suscriptor

Hasta este punto se tienen especificados los modelos de dos vistas básicas, el modelo de la *vista modular* y el *modelo de componentes-y-conectores*. Ahora hace falta ver cómo se pueden llegar a diseñar, y sobre todo modelar relaciones de correspondencia usando el enfoque dirigido por modelos, para lo cual a continuación se verá en qué consiste este enfoque y como se relaciona con las vistas arquitectónicas.

Capítulo 4

Desarrollo Dirigido por Modelo en las Vistas Arquitectónicas

Este capítulo está dividido en dos partes, en la primera se trata el enfoque de desarrollo dirigido por modelo y su relación con la arquitectura (las tres primeras secciones), y en la segunda parte se presenta la propuesta que se sigue en esta tesis para el modelado de las vistas arquitectónicas siguiendo este enfoque (las dos últimas secciones).

En la primer parte, se inicia por destacar lo que es la aproximación dirigida por modelos, su importancia y los conceptos clave que involucra. Luego la atención se centra en la arquitectura dirigida por modelos (conocida como MDA por sus siglas en ingles), la cual es una aproximación específica de llevar a cabo el modelado, revisando su proceso y el papel que ocupa la arquitectura¹ en él. Se continúa profundizado en una de las principales tareas de este proceso, la transformación de modelos, revisando las estrategias para llevarla a cabo, entre las que destaca la adoptada aquí para llevar a cabo la transformación entre vistas.

En la segunda parte se presenta el enfoque propuesto para llevar a cabo el modelado de las vistas arquitectónicas. Se inicia por especificar las tareas a realizar con estas vistas dentro del esquema que se sigue en MDA, y se concluye con la descripción de las actividades del proceso propuesto para llevar a cabo el modelado y gestión de dichas vistas arquitectónicas.

¹ El término arquitectura usado en MDA, tiene un sentido más general que la arquitectura de software.

4.1 Desarrollo dirigido por modelos

El desarrollo dirigido por modelos (DDM), surge como una manera de elevar el nivel de abstracción en el diseño e implementación de un sistema, evitando la dependencia intrínseca que se genera al desarrollar un sistema dentro de una plataforma específica tal como .net, CORBA, o J2EE. Con ello se pretende reducir la complejidad inherente del software, centrando todo el esfuerzo en el nivel de modelado.

El uso de modelos en el software para reducir la complejidad no es nuevo, pero los modelos han sido usados en forma parcial en el proceso de desarrollo, por ejemplo en las fases de análisis, diseño y de pruebas, sin considerar su uso en la fase de implementación por considerar a esta fase como la materialización de los modelos, es decir donde se incluyen aspectos puramente computacionales. Sin embargo dentro del DDM todo es considerado como un modelo, hasta los productos computacionales que se llegan a obtener (Selic, 2003). Con este enfoque se usan los modelos tanto para razonar en el dominio del problema, como en el dominio de la solución, creando relaciones de dependencia entre estos dominios que sirven como un *tejido* que registra el proceso por el cual la solución fue creada, ayudando a entender las implicaciones de los cambios en cualquier punto de este proceso (Brown et al., 2005).

Para hacer posible que el enfoque de modelado sea viable y aplicable en forma general, se plantean dos situaciones que deben cumplirse, una va en el sentido de la automatización y la otra en el sentido de contar con un estándar, en (Selic , 2003) se propone abordar estas dos situaciones de la siguiente manera:

- Con la automatización se propone cubrir dos aspectos, uno es emplear tecnologías de automatización para generar programas completos a través de un ordenador, y el segundo aspecto es poder verificar los modelos en forma automática. De acuerdo a (Selic 2003) la generación automática del código completa, implica que se deben tener lenguajes de modelado que tomen el rol de los lenguajes de implementación, pero ahora con un mayor nivel de abstracción y que además cualquier modificación sea hecha a través de ellos, sin manipular el código que lleguen a generar.
- La verificación de modelos automáticamente, se refiere al uso del ordenador para analizar si el modelo cumple o no con las propiedades deseables y omite las no deseables, lo cual puede ser hecho en forma formal, o mediante técnicas de análisis para evaluar su desempeño, o bien aplicando enfoques empíricos para la verificación. Lo fundamental aquí es apoyar las decisiones que hacen los ingenieros de software en el diseño de los modelos.
- La estandarización, como es sabido provee beneficios que se reflejan en mejores prácticas, posibilita la reutilización y facilita el intercambio entre herramientas complementarias. El DDM cuenta con la propuesta de MDA para esto, que es la mejor representación de este enfoque ya que ofrece un marco de trabajo conceptual que definen un conjunto de estándares. MDA ha sido desarrollado por el OMG¹, un consorcio sin fines de lucro dedicado al cuidado y establecimiento de estándares como el UML, XMI, y CORBA, lo cual da solides y

¹ OMG siglas en ingles de Object Management Group: Grupo de Gestión de Objetos

soporte al enfoque de DDM. Esta propuesta es la que se abordará aquí, puesto que servirá de base para modelar las vistas arquitectónicas de software, y establecer y mantener sus vínculos.

Además de la automatización y estandarización, el DDM, y por lo tanto el MDA, están fundamentados en conceptos clave como modelo y meta-modelo, por lo cual es necesario revisar y precisar estos conceptos.

Modelo

Dentro del contexto del DDM la forma en que se ha concebido lo que es un modelo, es como una *especificación*. Una de las primeras definiciones de un modelo dentro del contexto del DDM, dice:

‘Un modelo es un conjunto de declaraciones acerca de algún sistema bajo estudio’ (Seidewit, 2003).

El término *declaraciones* de esta definición alude a que un modelo es una especificación, y que está circunscrita a un área o sistema de interés.

Una definición más explícita de modelo es la dada dentro del contexto de MDA:

‘Un modelo, es una descripción o especificación de un sistema y de su ambiente para un propósito determinado’ (OMG Technical Report omg/2003-06-01, 2003).

En esta otra definición además de ubicar al modelo otra vez como una especificación, se incorpora también al ambiente del sistema, y se declara que hay una intencionalidad en su descripción, pero no se precisa el tipo de especificación que se hace, ni su alcance.

Más recientemente dentro del contexto del UML (que está relacionado con el MDA) se define a un modelo como:

‘Una descripción semánticamente completa de un sistema’ (Rumbaugh et al., 2005).

Una vez más en esta tercera definición se vuelve a hacer referencia a un modelo como una descripción, pero precisando que esta descripción es sobre su semántica y que ésta debe ser completa.

La definición que aquí se utiliza incluye lo contenido en las propuestas anteriormente es la siguiente:

Un modelo es la descripción semánticamente completa de un sistema y de su ambiente, para un propósito determinado.

Adicionalmente a lo especificado en esta definición, para que un modelo se comporte como los modelos ingenieriles en el sentido de ser hacerlo útil y efectivo, un modelo de software debe poseer en un cierto grado ciertas características que lo haga ser como tal, (Selic, 2003) propone estas cinco características:

- *Abstracción.* Un modelo es siempre una interpretación reducida del sistema que representa. Esta propiedad también es señalada en una definición genérica de un modelo dada por Hagget y Chorley, 1967), al considerarlo como una abstracción simplificada de la realidad.
- *Comprensible.* La especificación que se haga del modelo debe ser de alguna manera perceptible en forma intuitiva.
- *Preciso.* Un modelo debe representar lo más fielmente posible la realidad de las características de interés del sistema que se modele.

- *Predictivo*. Se debe poder usar un modelo para hacer predicciones de lo interesante que se modele del sistema sin que sean propiedades obvias, mediante la experimentación o a través de un tipo de análisis formal.
- *Rentable*. La construcción y el análisis que se haga del modelo deben ser significativamente más baratos que el sistema modelado.

De acuerdo al grado en que el modelo llegue a cumplir con estas propiedades será la calidad que se le pueda atribuir en su diseño.

Meta-modelo

El segundo concepto total dentro del DDM corresponde a lo que es un meta-modelo. Este término ha sido usado en otras disciplinas de la ciencia para definir modelos matemáticos o físicos. Dentro del software uno de las definiciones más difundidas sobre este término es la dada dentro del UML, donde se usa a un meta-modelo para cada uno de los nueve modelos que forman a este lenguaje.

En (Gašević et al., 2006) un meta-modelo es definido como: *‘una especificación de un modelo para una clase de sistema bajo estudio. Dónde cada sistema bajo estudio en la clase, es él mismo, un modelo válido, expresado en un cierto lenguaje de modelado’*.

Un meta-modelo establece lo que puede ser expresado como válido en el modelo, por ejemplo el UML describe con precisión y detalle el significado de una clase, un atributo, y el significado entre la relación de estos dos conceptos. De hecho un meta-modelo es un *modelo de un lenguaje de modelado* (Seidewitz, 2003).

En la Figura 4.1 se muestra mediante un diagrama de clases de UML, la relación entre el sistema bajo estudio y un lenguaje expresado en un lenguaje de modelado específico. En este caso la clase Meta-modelo es el lenguaje de modelo que contiene la terminología y aseveraciones que define a la clase Modelo.

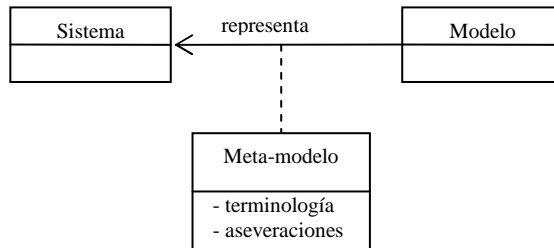


Figura 4.1. La correspondencia entre un modelo y un sistema, tomado de (Gašević et al., 2006)

Una vez que se han precisado los principios del DDM y sus conceptos clave, se continúa detallando como todo esto es aplicado dentro del MDA.

4.2 Elementos de modelado en MDA

La propuesta de MDA materializa los propósitos del DDM, por un lado proporciona una estrategia para realizar el modelado de un sistema basado en los meta-modelos y en diferentes niveles de abstracción, proporcionando un estándar para definir el lenguaje de modelado (es decir el meta-modelo) y por otro lado establece la manera de llevar a cabo las interacciones entre los diferentes niveles de abstracción para poder generar modelos a partir de otros modelos, proporcionando estándares de los lenguajes

necesarios para llevar a cabo la relación entre los meta-modelos y realizar conversiones entre modelos.

Los cuatro principios que fundamentan la propuesta del OMG para MDA (Beydeda, et al., 2005) son:

- *Lo modelos expresados en una notación bien definida constituyen la piedra angular para la comprensión de soluciones de escala empresarial*
- *Los sistemas construidos pueden ser organizados mediante un conjunto de modelos a través de una serie de transformaciones entre modelos, organizados dentro de un marco arquitectónico de capas y transformaciones.*
- *Una formalización para describir modelos en un conjunto de modelos facilita significativamente la integración y transformación entre modelos, y constituye las bases para la automatización a través de herramientas.*
- *La aceptación y amplia adopción de este enfoque basado en modelos requiere estándares industriales para proveer apertura a los consumidores, y fomentar la competencia entre vendedores.*

En el primero de estos principios, la notación desempeña el papel fundamental, ya que con ella se debe expresar de una manera precisa lo que se pretende modelar. La notación en MDA está fundamentada en el UML, ya que tiene la característica de poder ser adaptado a varios dominios, esta adaptación se logra mediante un mecanismo llamado perfil, el cual ha sido definido en la versión 2.0 de este lenguaje (OMG document formal, 2004). Un perfil provee estructura y precisión a los estereotipos y valores etiquetados (mecanismos de extensión y especialización disponibles en las versiones anteriores de este lenguaje). Un ejemplo de su aplicación, se encuentra en la especificación del lenguaje

MOF donde se definen todos los elementos necesarios para construir meta-modelos, como se mostrará más adelante.

El segundo principio, constituye el soporte del proceso de modelado dentro del MDA, porque él determina tanto la organización de la arquitectura como la manera de realizar las transformaciones sobre esta organización. Por lo cual el proceso de transformación se tiene que iniciar por describir cómo se organiza la arquitectura y definir a los elementos subyacentes en ésta.

La organización del marco arquitectónico corresponde a los cuatro niveles de abstracción propuestos en los fundamentos del DDM (Atkinson y Kühne, 2003), estos forman una estructura arquitectónica de *capas* como se ilustra en la Figura 4.2. Cada nivel (o *capa*) está representado mediante una letra M (de modelo) y un número, el número indica el nivel de abstracción, el 0 representa el nivel básico (datos de una realidad particular), el 1 un nivel de abstracción elemental (modelo de una realidad), el 2 es un nivel donde se modelan a los modelos(meta-modelado), y el 3 es el nivel de abstracción más alto, es el meta del meta-nivel. Un nivel inferior de abstracción representa una instancia del siguiente nivel superior, el nivel M0 (datos/realidad) es una instancia del nivel M1 (modelo), y M1 a su vez es una instancia de M2 (meta-modelo), y finalmente M2 representa instancias de M3.

Es importante aclarar que el sentido que se aplica aquí sobre *instanciación* es *lingüístico*, ya que también existe el tipo de *instanciación ontológica* (Gašević et al., 2006), que es aplicado a los conceptos que se manejan dentro de un mismo nivel de abstracción. Por ejemplo en el nivel M2 donde se usa al UML como lenguaje, los llamados perfiles de este lenguaje serían instancias (ontológicas) del meta-

modelo del UML, y a su vez los meta-modelos de los usuarios serían una instancia de los perfiles. Otro ejemplo es mostrado en (Atkinson y Kühne, 2003) con las clases y objetos, donde los objetos son instancias de las clases, y ambos están en el mismo nivel conceptual (misma *capa*).

En el lado derecho de la Figura 4.2, se muestran los lenguajes y la abstracción correspondiente a cada nivel, en el nivel M3 se tiene al lenguaje estándar propuesto por OMG para MDA llamado MOF¹, que es usado para especificar a ese mismo nivel (meta meta-modelo) y para especificar los meta-modelos del nivel M2, que corresponde a los conceptos del UML (Atkinson y Kühne, 2003), con el lenguaje de M2 (meta-modelos), se describe a los modelos que corresponden a los conceptos del usuario del nivel M1, y finalmente estos últimos describen a los datos del usuario.

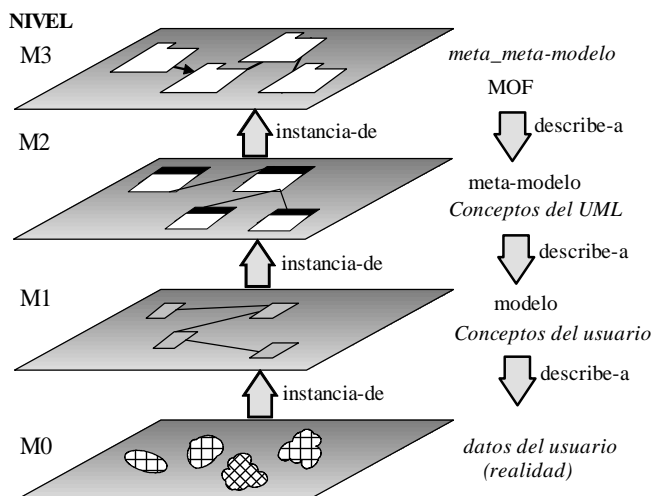


Figura 4.2. Arquitectura de capas del enfoque DDM

¹ MOF corresponde a las siglas de su designación en ingles: Meta-Object Facility

De estos cuatro niveles (o capas) de abstracción propuestos en el DDM, sólo tres de ellos son considerados dentro del contexto de MDA, los niveles M3, M2 y M1, cada uno de estos niveles de abstracción corresponde a las diferentes plataformas donde se ubican los modelos que se llegan a realizar en MDA. En el nivel M3 se ubican los modelos independientes de la computación llamados CIM (por sus siglas en ingles), en el nivel M2 se ubican los modelos que son independientes de una plataforma tecnológica llamados modelos PIM, y al nivel M1 se ubican los modelos que pertenecen a una plataforma específica llamada PSM¹.

Nótese que se ha empleado el término plataforma con el significado de ubicación de los modelos en diferentes niveles de abstracciones con diferentes propósitos de uso, en lugar de dar una definición de este término que puede resultar compleja, como la presentada en (OMG, 2003).

Por otro lado, aunque el significado de los términos CIM, PIM, PSM, usados para cada modelo de cada nivel pueden ser suficientemente explícito (sobre todo para los nativos del idioma inglés), es importante destacar las características adicionales de cada uno de estos modelos.

Un CIM, es un modelo que corresponde a lo que dentro del ámbito de la ingeniería de software se le denomina como modelo de un dominio específico, esto es, tienen un concepto muy similar al de una ontología².

Un PIM, es un modelo que describe a un sistema, sin tener ningún conocimiento de la plataforma específica donde se van a implementar, pero este modelo no es independiente

¹ Los nombre se cada plataforma son la siglas de sus términos en ingles: CIM (computational-independent model), PIM (platform-independent model, PSM (platform specific model)

² Una ontología es una representación explícita de un conocimiento compartido de los conceptos importantes en algún dominio de interés (Kalfoglou, 2001)

de la maquina virtual donde será ejecutado, por lo que es considerado *dependiente-computacionalmente* (Gašević et al., 2006).

Un PSM, es un modelo que describe a un sistema con pleno conocimiento de la plataforma donde será ejecutado. Este tipo de modelo es considerado como la meta final del proceso del desarrollo de un sistema, porque contiene los detalles de la plataforma específica.

Un PSM, es generado mediante un proceso de transformación que utiliza a un PIM como fuente para producirlo, usando un conjunto de reglas y herramientas que permiten la automatización de este proceso, como se muestra en la Figura 4.3. En esta Figura se observa cómo la transformación emplea a estas reglas y herramientas para este propósito. Las reglas se forman con los lenguajes usados para describir a esos modelos, y las herramientas hacen uso de estas reglas, para poder transformar un PIM a un PSM.

La transformación de un PIM a un PSM ilustra el caso de llegar a obtener modelos aplicables a plataformas específicas, es decir entre dos niveles distintos de abstracción. Pero no es el único caso de transformación entre modelos, existen diversas formas y tipos de éstas. Lo cual será abordado a continuación.

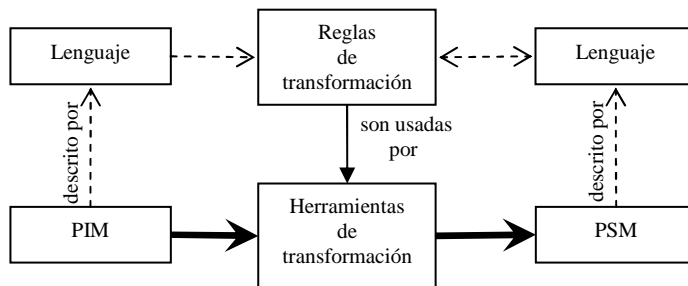


Figura 4.3. Transformación de un PIM a un PSM, adaptación hecha de (Gašević et al., 2006)

4.3 Transformación entre modelos

Para poder tener una mayor precisión sobre lo que es una transformación entre modelos, es conveniente presentar una especificación formal de esta, y a partir de ahí establecer las variaciones que pueden existir de ellas. Una transformación t es definida¹ como:

$$t : M_1(S_1)|_{L_1} \rightarrow M_2(S_2)|_{L_2}$$

Donde, M_1 es el modelo origen, M_2 es el modelo destino, cada modelo describe a su correspondiente sistema S_1 y S_2 , y L_1 y L_2 son los lenguajes que definen la sintaxis (o notación), y la semántica de cada uno de los modelos.

La forma de llevar a cabo una transformación t depende de varios factores que intervienen en su proceso, factores como el nivel o niveles de abstracción donde se aplica, el tipo de relación existente entre cada modelo, y otros más. En (Czarnecki, y Helse 2003) se hace una clasificación de las transformaciones de acuerdo a ocho criterios que constituyen los modelos principales de transformación

¹ Adaptación hecha de lo presentado en (Metzger, 2005).

(reglas de transformación, alcance de aplicación de las reglas, relación fuente-destino, estrategia de aplicación de la regla, programación de la regla, organización de la regla, rastreo, direccionalidad). Estos a su vez son descompuestos en otros, usando un diagrama de características para indicar su relación de dependencia, llegándose a formar más cincuenta tipos diferentes. En (Brown, et al., 2005) se describen tres de los tipos más usuales, la transformación de refactorización, la de modelo a modelo y la de modelo a código. En (Sendall, Kozaczynski, 2003) se agregan otros tipos de transformación usando una variedad de criterios de clasificación, diferentes a los usados en la clasificación de Czarnecki, y Helse.

Una manera de establecer diferentes tipos de transformaciones, es analizando la forma en cómo se vinculan los sistemas(S), lenguajes (L) y modelos (M), de acuerdo a si se usa el mismo (=) sistema, o lenguaje, o modelo, o si se usan diferentes (\neq) sistemas, lenguajes o modelos, combinando estas dos posibilidades que tienen cada elemento (S, L, M), lo cual es propuesto por (Metzger, 2005). Esta clasificación es mostrada en la Figura 4.4.

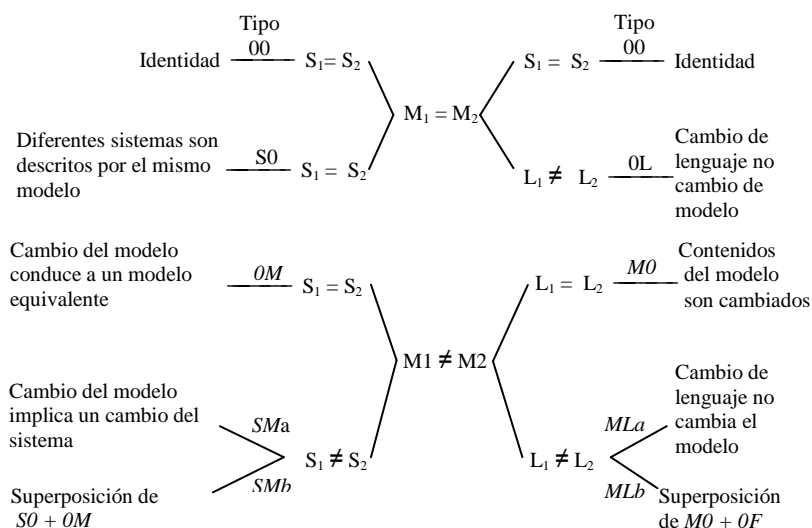


Figura 4.4. Clasificación de tipos de transformación, tomada de (Metzger, 2005)

La Figura 4.4, muestra cómo las combinaciones se forman alrededor de las relaciones entre los modelos (M_s), haciéndose dos grupos, uno con $M_1=M_2$ y el otro con $M_1 \neq M_2$. De ahí se proyectan radialmente todas las combinaciones posibles (de $=$ y \neq) con L y S . Así se establecen nueve tipos de transformaciones identificadas con 00 (de identidad), S0, OL, OM, M0, SMa, SMB, MLa, y MLb, cuyas descripciones se indican en la Figura 4.4 referida. Con este tipo de transformaciones, se pueden analizar cómo se comportan los otros tipos de transformaciones principales que se plantean en (Czarnecki, y Helse 2003), (Brown, et al., 2005), y (Sendall, Kozaczynski, 2003). Para ilustrar esto se analizan las transformaciones de tipo refactorización, de modelo a modelo, y de modelo a código.

Transformación de refactorización, es aquella donde el modelo destino es una versión modificada del modelo origen, el tipo que le corresponde de acuerdo a la Figura 4.4,

es el de identidad, porque $M1=M2$ y $S1=S2$. La *refactorización* reorganiza un modelo usando un criterio bien definido, y puede ser tan simple como sólo un renombrado de algunos elementos del modelo, o más elaborado como el de remplazar una clase por un conjunto de clases y relaciones.

La transformación de modelo a modelo, convierte información de un modelo a otro modelo, es decir se lleva a cabo dentro de un mismo nivel de abstracción (capa M2). El tipo de transformación correspondiente (de la Figura 4.4) es el OM, porque el cambio de modelo conduce a un modelo equivalente, por ejemplo en el caso de una modelo de clases de la orientación a objetos, se puede convertir a una tabla del modelo relacional de una base de datos, no hay cambio de sistema ya que están en el mismo nivel de abstracción, y los modelos son equivalentes.

La transformación de modelo a código, en este caso cada elemento de un modelo se convierte a un fragmento de código. El tipo de transformación al que corresponde de acuerdo a la Figura 4.4, es SMA, porque un modelo pertenece a un tipo de sistema diferente que el del código, es decir pertenecen a niveles diferentes de abstracción, por eso también se le conoce como transformación vertical, porque atraviesa de un nivel a otro. Esta es precisamente una transformación de PIM a PSM, donde el PSM contiene el código producido por la transformación. Este código puede ser de cualquier tipo de lenguaje de programación imperativo o declarativo, o de un lenguaje de dominio específico (DSL), o de un ADL como PRISMA por ejemplo. Aunque este tipo de transformación no siempre resulta efectivo en la práctica (Stahl et al., 2006), debido a la gran cantidad de detalles que contienen los lenguajes, los cuales no siempre son posibles de generar en el proceso de transformación en forma automática.

De los tres tipos de transformaciones fundamentales presentados, el que se usa dentro de esta tesis es el tipo de transformación de modelo a modelo, pues se trata de mantener una sincronización entre las diferentes vistas de un mismo sistema (Czarnecki, y Helse 2003), en este caso entre vistas arquitectónicas de software, por lo que el esfuerzo está dirigido hacia los aspectos que se involucran con este tipo de transformación.

Uno de los aspectos más relevantes en una transformación de modelo a modelo es la forma en que se puede llevar a cabo la transformación, es decir la estrategia usada para convertir un modelo en otro, porque esto involucra el lenguaje, y el soporte formal y tecnológico a usar. La estrategia a seguir puede ser desde una transformación directa empleando lenguajes de propósito general adaptados para esta tarea, como *Jamda*¹ que es un compilador de modelos hecho en java, o bien estrategias orientadas a este fin, como una transformación gráfica o basada en relaciones, o una combinación de varias estrategias (híbrida).

La estrategia de una transformación gráfica hace uso de técnicas de re-escritura de gráficos para manipular gráficos apoyados en el hecho de que la mayoría de los modelos pueden ser representados de esa manera, ya sea en forma directa (por ejemplo, usando grafos) asignándoles tipos, o mediante un *morfismo* gráfico (Grunske et al., 2005). En este caso una regla de transformación es especificada mediante dos gráficas de tipo dirigidas llamadas lado de la *mano-derecha* (LMD) y lado de la *mano-izquierda* (LMI), si la gráfica de LMI corresponde con la gráfica fuente ésta es remplazada

¹ <http://jamda.sourceforge.net/docs/intro.html#N400025>

con la gráfica LMD. Para aplicar la regla de transformación se sigue un algoritmo (Grunske et al., 2005) (Ehrig , et al. , 2006) que no es abordado aquí, ya que la estrategia de transformación gráfica es usada principalmente para fines de análisis lo cual no corresponde con el objetivo de la tesis.

La estrategia de transformación de modelos que usa relaciones, está basada en las relaciones que se establecen entre los elementos del modelo fuente con los elementos del modelo destino, estas relaciones son especificadas mediante reglas de transformación como se hace en ATL¹ (Jouault y Kurtev, 2006), o mediante restricciones como lo hace QVT² (OMG, 2005).

Las siglas ATL significan lenguaje de transformación ATLAS (ATLAS, 2006). Este lenguaje ATLAS posee dos tipos de reglas de transformación: las reglas de correspondencia (usa programación *declarativa*) y las reglas de llamado (usa programación *imperativa*). El primer tipo de reglas constituye el núcleo del lenguaje porque mediante ellas se especifica la transformación entre los modelos, cada regla de este tipo está formada de dos partes, llamadas *patrón fuente* y *patrón destino*. El *patrón fuente* especifica un conjunto de tipos fuentes provenientes del meta-modelo origen, y a un *guarda* que es una expresión booleana escrita en OCL (que sirve para incorporar restricciones). El *patrón destino*, está compuesta de un conjunto de elementos que especifican a un tipo del meta-modelo destino, y a un conjunto de enlaces que vinculan a los elementos de los modelos fuente y destino. Por otra parte, las reglas de *llamado* sirven para realizar acciones complementarias con los elementos del modelo destino, y no establecen enlaces entre los modelos relacionados, son como procedimientos

¹ ATL: siglas de Atlas Transformation Language

² QVT: siglas en ingles de Query, View, Transformation

auxiliares en la tarea de transformación. Por lo tanto, los vínculos o relaciones entre los modelos quedan establecidos sólo a través de la especificación de las *reglas de correspondencia*.

La otra propuesta de transformación basada en relaciones llamada QVT, surge del estándar MDA propuesto por OMG. A diferencia del lenguaje ATLAS, QVT permite especificar relaciones multidimensionales en una transformación. En su versión inicial (OMG, 2002), se planteaba que una relación en QVT estaba complementada con una *correspondencia*, esto es que mediante una relación se establecían los vínculos entre un meta-modelo y otro, y mediante las *correspondencias* se llegaba a implementar la transformación entre modelos. En su versión final (OMG, 2005), QVT dispone de dos lenguajes para realizar esto, el lenguaje QVT-relaciones y el QVT-operacional.

Mediante QVT-relaciones se establecen relaciones de correspondencia entre meta-modelos (llamados modelos candidatos) en forma declarativa, las cuales deben ser mantenidas para llevar a cabo una transformación exitosa, creando en forma implícita una traza entre los meta-modelos y los modelos involucrados, cuando se lleva a cabo una transformación.

El segundo lenguaje, QVT-operacional, es usado para implementar las relaciones de la especificación de relaciones complementando las transformaciones (enfoque híbrido), o bien para definir una transformación por completo con un enfoque imperativo, ya que se tiene que especificar cómo se tiene que hacer una transformación.

Para llevar a cabo la transformación entre las vistas arquitectónicas propuestas en esta tesis, se usará el enfoque de una transformación basada en relaciones, adoptando el

enfoque del MDA por ser un estándar que ofrece no sólo la posibilidad de usar los dos lenguajes antes citados, sino de contar con una notación consistente para expresar las relaciones entre los meta-modelos.

Hasta este punto se ha fijado las bases sobre las cuales se fundamenta la propuesta del modelado de las vistas arquitectónicas, a continuación se procede a detallar cómo esta propuesta se llevará a cabo.

4.4 Propuesta de modelado de vistas arquitectónicas de software basándose en MDA

Como se ha mencionado anteriormente, las vistas de una arquitectura de software están sustentadas por sus modelos, cada vista está soportada por lo menos por un modelo como se mostró en el capítulo anterior. De tal manera que la forma de una arquitectura dependerá de la manera en que se lleguen a conformar los modelos de una vista.

La propuesta de generación de los modelos de las dos vistas básicas y sus vistas particulares usando MDA como estrategia, permitirán no solo llegar a producir una vista a partir de la otra, sino que se podrán establecer vínculos de correspondencia permanentes entre ellas, con lo cual será posible crear mecanismos para conservar estos vínculos, es decir su consistencia, como se ha especificado en los objetivos de esta tesis.

Los modelos de las dos vistas básicas aquí consideradas (la de módulos y de componentes-y-conectores), ya han sido especificados en el capítulo anterior como parte del marco de trabajo para el modelado de las vistas, pero aún falta por definir cómo se pueden llegar a generar estas, profundizar

en sus relaciones y llegar a establecer el mecanismo para mantener la consistencia.

La propuesta que se hace aquí para la generación de los modelos de las vistas está basada en los siguientes principios:

- Los modelos de las vistas básicas aquí consideradas (y por ende todas las derivadas de ellas), usan como fuente de información lo obtenido del análisis de los requisitos, y lo que los ingenieros de diseño consideran necesario que deba cumplirse (atributos de calidad).
- La vista modular es *prevaleciente* sobre la de componentes-y-conectores. Por *prevaleciente* se entiende que la vista modular debe ser formada primero porque sus elementos sirven como base para constituir los elementos de la vista de componentes-y-conectores, está última es considerada como una vista *subordinada*. Sin embargo puede haber elementos de esta vista *subordinada* que sean detectados antes que en la vista *prevaleciente*, esto es porque en el modelado de las vistas interactúan varios actores en forma simultánea que identifican elementos de las dos vistas en forma indistinta. Por lo tanto, cuando esto suceda se deben generar sus elementos correspondientes en la otra vista (no tiene que ser instantáneamente).
- Todos los elementos de la vista modular tienen correspondencia con los elementos de la vista de componentes-y-conectores, pero a la inversa no es así. Esto se debe a que en la vista de componentes y conectores se incluye información de la interacción

entre los componentes, que no se capta en la vista modular.

- La información de las interacciones entre los componentes de la vista de componentes-y-conectores, es derivada de un análisis de su comportamiento. Esto significa que las interacciones también deben ser modeladas a efecto de ser consideradas en el proceso de transformación. El modelo de interacciones permite complementar a un modelo de la vista de componentes haciendo que se especifique con detalle las relaciones entre los componentes y conectores.

La manera de llevar a cabo estos principios a través de la estrategia del MDA es mostrada en la Figura 4.5. Ahí se aprecia lo que en cada nivel de abstracción se realiza, lo cual se describe a continuación.

En el nivel M3, se dispone de MOF como lenguaje para definir a los meta-modelos del nivel M2, esto es a los meta-modelos de las vistas modular y de componentes-y-conectores, y al meta-modelo de interacciones.

En el nivel M2, se establece cómo se llegaran a relacionar a los meta-modelos de las vistas, estas relaciones son identificadas en la referida Figura 4.4 como R1 y R2. Las relaciones contenidas en R1 vinculan a los meta-modelos de las dos vistas en cuestión (modular y de componentes-y-conectores), también R1 vincula al meta-modelo de la vista modular con la de interacciones. Las relaciones de R2 vinculan el meta-modelo de la vista de componentes-y-conectores con el meta-modelo de interacciones, este último está basado en la superestructura del UML 2.0, en la parte que corresponde al modelo de secuencia y/o de

escenarios, ya que ellos contienen la especificación que permite modelar las interacciones de los componentes.

En el nivel M1 los modelos correspondientes a cada meta-modelo son creados. Al formarse un modelo se debe hacer una verificación de que éste cumple con todo lo especificado en su meta-modelo, esto es impuesto por el propio MDA, lo cual se representa en la Figura 4.5 mediante las líneas discontinuas que unen al meta-modelo con su correspondiente modelo. Las flechas en ambos extremos de esas líneas indican que al crearse un modelo se realiza la verificación.

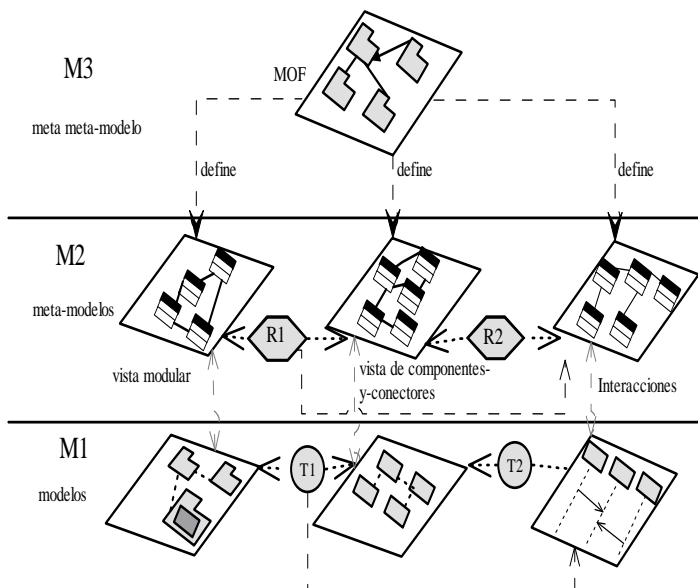


Figura 4.5. Estrategia para transformación de modelos de vistas arquitectónicas de software

En la Figura 4.5 antes referida, se indica también que los modelos de la vista modular y el de interacciones están vinculados con el modelo de la vista componentes-y-conectores mediante dos procesos de transformación

representados como T1 y T2 (encerrados entre círculos). Cada proceso de transformación incluye tareas distintas, las cuales se describen a continuación.

T1 incluye tres tareas:

- a) La generación parcial de un modelo de la vista de componentes-y-conectores a partir un modelo de la vista modular (fuente). El modelo producido es creado en forma parcial, pues contiene solo a los componentes, ya que hace falta incorporarle la información detallada que se formará usando un modelo de interacciones, esta información faltante será incluida en los conectores.
- b) La generación básica de un modelo de interacciones a partir del mismo modelo de la vista modular usado en la tarea (a), el cual deberá ser refinado posteriormente. El refinamiento consiste en agregarle precisamente las interacciones entre los elementos.
- c) La re-generación ya sea del modelo de la vista modular y/o del modelo de la vista de componentes-y-conectores, cuando alguno de los dos sea modificado. El modelo que cambie será usado como fuente para ‘transmitir’ los cambios al otro modelo.

Nótese que la vista modular es usada como modelo fuente en T1 para generar los otros dos modelos, el de componentes-y-conectores y el de interacciones (por lo menos en forma parcial, ya que con T2 se llegará a completar al modelo de componentes y conectores). De tal manera que el primer modelo generado contiene componentes como clases, mientras que el segundo contiene objetos de esos componentes, con los cuales se pueden llevar a cabo

las interacciones, de las cuales se deriva la información de los conectores.

T2 incluye dos tipos de tareas, una tarea de refinamiento y una tarea de transformación propiamente:

- a) El refinamiento se realiza sobre el modelo de interacción (el cual debió haber sido generado previamente por T1). Este modelo se refina mediante la incorporación de información que modela la comunicación entre sus elementos, esta información adicional proviene de los escenarios entre objetos de componentes que es proporcionada por los actores involucrados.
- b) La transformación de un modelo de interacciones a un modelo de componentes-y-conectores, regenerando (o completando) el modelo que se produjo en T1, esto es agregando a los conectores y su enlaces con los componentes.

Es importante destacar que los procesos de transformación aquí propuestos realizan transformaciones de modelo a modelo, ya que tanto los modelos de las dos vistas como el modelo de interacciones están en el mismo nivel de abstracción. De esta manera se llega a producir un PIM mediante otro PIM, lo cual corresponde a una transformación horizontal del tipo¹ M0, obteniéndose modelos diferentes que pertenecen al mismo sistema. Esto se cumple tanto en T1 como en T2.

Ahora falta ver todo hay que hacer para llevar a cabo esta estrategia de MDA, es decir cómo integrar todas las tareas dentro de un proceso global que conduzca a la formación de los modelos de las vistas arquitectónicas.

¹ De acuerdo a la clasificación mostrada en la Figura 4.4

4.5 Propuesta del proceso para el modelado de las vistas mediante MDA

Existen varios métodos de llegar a formar los modelos de las vistas, los cuales van desde la documentación (Clements et al., 2003) hasta la forma de llegar a establecer descripciones mediante un ADL, sin embargo estas maneras de crear modelos de una arquitectura no abordan directamente a las vistas, y las que lo hacen como el método ADD no llegan a establecer relaciones entre sus elementos en forma ni en forma explícita ni persistentes.

El proceso propuesto para el modelado de las vistas se muestra en la Figura 4.6, este comprende las actividades a llevar a cabo, la información de entrada requerida y la que se llega a producir como resultado de llevar a cabo cada actividad. Todo ello está agrupado por nivel (o capa) de abstracción en donde las actividades se llegan a realizar. En el nivel M2 se agrupan actividades dirigidas a la conformación de los meta-modelos y sus relaciones, y en el nivel M1 se comprenden actividades que permiten el diseño de los modelos arquitectónicos. Las actividades del nivel M2 sirven de soporte a las actividades del nivel M1, y se llevan a cabo cada vez que se requiera incorporar un meta-modelo de una vista o se tengan que modificar algunas características de los meta-modelos existentes o de las relaciones entre ellos. Las actividades del nivel M1 permiten definir los modelos de las vistas para un sistema específico y pueden realizarse tantas veces como sean necesarias en forma iterativa hasta llegar a formar las vistas, y volver a repetirse cada vez que el sistema requiera ser modificado.

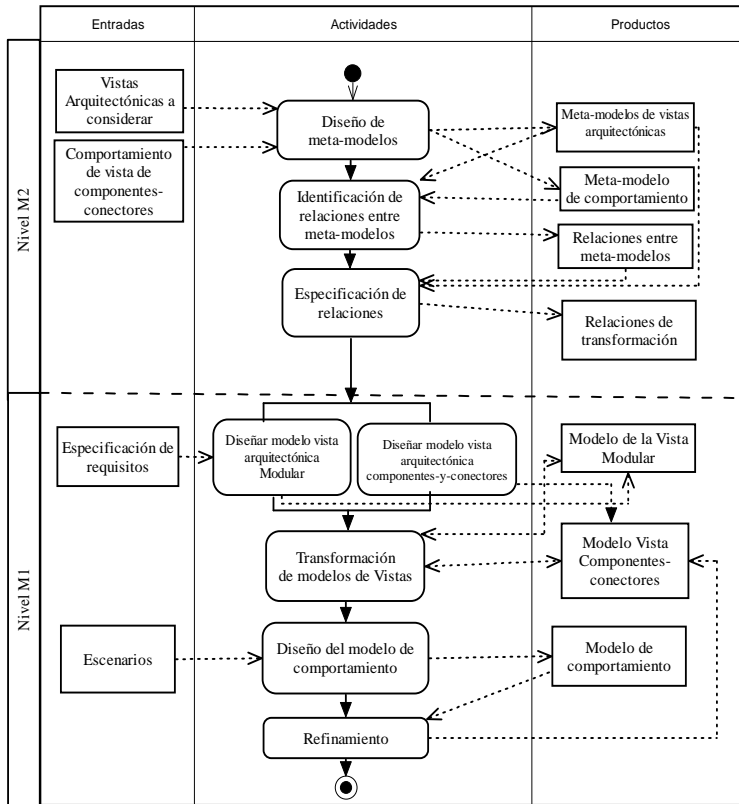


Figura 4.6 Proceso propuesto para modelar vistas arquitectónicas mediante MDA

Para la realización de actividades del nivel M2 se utiliza como entrada cada una de las vistas a modelar (modular y de componentes-y-conectores), y la información que expresa el comportamiento de los elementos de la vista de componentes-y-conectores. De cada vista se proporciona sus elementos, sus propiedades visibles y los tipos de relaciones identificables entre sus elementos, y la información de comportamiento se toma de las posibles vistas especializadas que pueden formarse con los conectores y componentes. Con esta información se procede

primero con la actividad del diseño de sus meta-modelos los cuales constituyen sus principales productos, estos se mantienen en forma persistente ya que son utilizados tanto para la identificación y especificación de sus relaciones como para las actividades subsiguientes del nivel M1.

La siguiente actividad dentro del nivel M2 es la identificación de relaciones entre meta-modelos, la cual permite precisar la manera en que los elementos de cada meta-modelo se vinculan entre sí, estos vínculos son dados por el arquitecto quien determina la correspondencias entre sus elementos. Del nivel de correspondencia depende el grado de acoplamiento entre cada meta-modelo. Estos vínculos son especificados en la siguiente actividad mediante el lenguaje QTV-relaciones, produciéndose las relaciones de transformación que junto con los meta-modelos harán posible la transformación de los modelos de las vistas en el nivel M1.

Continuando con las actividades del nivel M1 descritas en la Figura 4.6, éstas inician usando como entrada la especificación de requisitos del sistema del cual se pretende diseñar su arquitectura de software. Los requisitos pueden ser expresados en forma textual y/o por los modelos de casos de usos. Con esta información se va formando el modelo de la vista modular y/o el modelo de la vista de componentes-y-conectores. Aunque si bien es cierto que la actividad de diseño de la arquitectura de software inicia con la vista modular por ser esta la que considera las estructuras fundamentales, también se puede ir formando parte de la vista de componentes-y-conectores puesto que en esta actividad participa tanto el arquitecto como los diseñadores, y estos últimos pueden detectar componentes que sean parte de esta vista. En cualquier caso, cada elemento del modelo de una u otra vista que se vaya incorporando debe

estar de acuerdo al meta-modelo correspondiente que se diseñó en el nivel M2.

La siguiente actividad consistente en la transformación entre modelos de las vistas arquitectónicas, la cual permite generar el modelo de una vista en función de la otra y mantener la consistencia entre sus modelos. Aquí se contempla que el modelo de la vista modular sea el modelo fuente para generar el modelo de la vista de componentes-y-conectores, pero como existe la posibilidad de que algunos elementos del modelo de la segunda vista sean incorporados directamente por los diseñadores, estos elementos deben ser también incorporados en la otra vista para mantener la consistencia, para lo cual se realiza también la transformación en el otro sentido, es decir usando como modelo fuente al de la vista de componentes-y-conectores y como destino a la vista modular.

Esta transformación bi-direccional se hace más evidente en la etapa posterior al diseño de las vistas, cuando teniéndose ya generados los modelos de ambas vistas alguno de ellos llega a ser modificado, entonces el modelo que se modifica será el modelo fuente y el destino será aquel que deba reflejar los cambios y con ello conservar la consistencia entre las vistas arquitectónicas.

Sin embargo, la transformación en el sentido de la vista modular a la de componentes-y-conectores no es suficiente para formar plenamente a esta última, puesto que como ya se mencionó anteriormente se requiere la incorporación de la información del comportamiento del sistema, para lo cual se tiene que diseñar su modelo correspondiente, esta es la siguiente actividad a considerar.

Para el diseño del modelo de comportamiento se usa como entrada la información sobre los escenarios que se dan en el

sistema a modelar, y a semejanza que los modelos de las vistas este es diseñado de acuerdo al meta-modelo de comportamiento creado en el nivel M2. Los escenarios por su parte son derivados del análisis de la interacción que se presenta en los elementos identificados en los casos de usos. Los elementos identificados son un tipo de objetos que corresponden a los componentes arquitectónicos que fueron derivados ya sea de una transformación de la vista modular o bien identificados directamente por los diseñadores y su interacción es la comunicación que se establece entre ellos, al solicitar y enviar información (eventos, mensajes, o solo datos). La especificación de esta información es una tarea conjunta que se da entre los arquitectos y diseñadores, con la cual ellos llegan a diseñar el modelo de comportamiento, que puede ser expresado mediante un diagrama de escenarios o un diagrama de secuencia.

Una vez que se tiene el modelo del comportamiento, se procede con la actividad de refinamiento del modelo de componentes-y-conectores que se produjo en la transformación previa. Con el refinamiento, se logra que los componentes de este modelo se lleguen a enlazar a través de los conectores al definir los servicios requeridos y ofrecidos entre los componentes, ya que estos servicios son los que los conectores llegan a coordinar entre los componentes relacionados. De esta manera se consigue crear la especificación completa de este modelo de componentes-y-conectores que conjuntamente con el modelo de la vista-modular forman la arquitectura de software del sistema que se esté modelando.

Las actividades del nivel M1 son realizadas tantas veces como lleguen a cambiar alguno de los modelos de las vistas, con el objeto de mantener la consistencia entre las vistas.

Cada una de las actividades del proceso propuesto descrito anteriormente, involucra a un conjunto de situaciones que tienen que ser resueltas y especificadas con precisión. Por ejemplo, la primer actividad consistente en el diseño de los meta-modelos de cada vista, implica un análisis de las relaciones de los elementos de las vistas especificadas en el Capítulo 3, para derivar todos los tipos de vistas que se pueden generar y luego del resultado de este análisis llegar a crear la ontología que sustente a los meta-modelos. De manera semejante cada actividad requiere procesos que tendrán que especificarse con precisión. Por lo tanto en los capítulos siguientes cada actividad del proceso propuesto será desarrollada con detalle.

Capítulo 5

Meta-modelos de las Vistas Arquitectónicas y sus Correspondencias

Dentro de MDA, los meta-modelos constituyen la base sobre la cual se forma a un modelo. En un meta-modelo se definen los constructos de un lenguaje de modelado, sus relaciones, y las restricciones y reglas de modelado, es decir su sintaxis abstracta y semántica estática (Stahl et al., 2006, pags. 57-58). En este caso con los meta-modelos de las vista modular y de componentes-y-conectores podrán formarse sus modelos correspondientes siguiendo el proceso de transformación planteado en el capítulo anterior. Para poder diseñar a estos meta-modelos se partirá de un análisis de los elementos y relaciones que los forman, se empleará el lenguaje MOF para representarlos y se procederá a su implementación a través del marco de trabajo de ECLIPSE. Esto se realiza en las dos primeras secciones de este capítulo.

Dentro de este mismo nivel de meta-modelado, se continúa con el establecimiento de las correspondencias entre los meta-modelos de las vistas, lo que permitirá no solo llevar a cabo las transformación entre modelos sino establecer las relaciones básicas de trazabilidad entre los elementos vinculados que son fundamentales para mantener la consistencia entre modelos. Estas correspondencias entre vistas serán representadas mediante diagramas dentro del contexto de QVT, y serán especificadas a través del lenguaje QVT-relaciones, e implementadas mediante el lenguaje QVT-operacional.

Al final de esta sección también se incluye un ejemplo de cómo se aplican los meta-modelos y relaciones entre ellos para producir un modelo de la vista de componentes y conectores a partir de un modelo de la vista modular, usando el proceso de transformación aquí propuesto.

5.1 Diseñando el meta-modelo de la vista modular

La especificación del tipo de vista modular sobre el tipo de elementos y relaciones que lo conforman se presentó en el capítulo III (Sección 3.4), pero para llegar a diseñar su meta-modelo se requiere además de ello especificar las restricciones que deben cumplir las relaciones entre sus elementos. Estas restricciones serán expuestas usando una representación de la vista con teoría de conjuntos como se indica enseguida.

Una vista modular V_m puede ser expresada como:

$$V_m = \{E_m, R_m\}$$

Donde E_m es el conjunto de elementos; R_m el conjunto de relaciones.

Siendo:

$$E_m = \{M, S, C\}$$

Donde M indica un elemento del tipo módulo; S indica un elemento del tipo *subsistema*; y C indica un elemento del tipo *capa*.

$$R_m = \{A, U, \hat{U}, \bar{U}, G\}$$

Donde A indica una relación de composición (o agregación) de los módulos que forman a un sub-sistema; U indica una relación de *uso* de un modulo a otro; \hat{U} indica una relación

de *uso* de un sistema a otro; \bar{U} indica una relación donde una *capa usa* a otra; y G indica una relación de *generalización*.

Una relación entre dos elementos x,y se expresará como: xAy , en este caso se indica que x es parte-de (o compone a) y , donde x es un módulo, y y es un sub-sistema. De la misma manera se expresa a cada relación aquí descrita.

Las relaciones que están sujetas a restricciones son aquellas que establecen una relación de dependencia, estas son las relaciones de uso (U), la relación de *uso* de sistema (\hat{U}), y la relación de *uso-capas* (\bar{U}) entre capas, ya que la dependencia que crean involucra ciertas condiciones. En cambio la relaciones de *agregación* (A), y *generalización* (G) no tienen ninguna restricción.

Para ilustrar la especificación de las restricciones (que serán referenciadas como RE) a la que están sujetas las relaciones arriba mencionadas, se usará la siguiente información como ejemplo: suponga que se tiene un conjunto de módulos $M=\{(a,b), (c,d,e), (f,g,h)\}$, en donde hay tres sub-sistemas: $S1 = \{(a,b)\}$; $S2 = \{(c,d,e)\}$; y $S3 = \{(f,g,h)\}$. Estos *sub-sistemas* están ubicados en dos capas, $C1 = \{S1,S2\}$, $C2 = \{S3\}$.

RE1: La relación de uso (U) que se da entre elementos de tipo modulo (M), sólo es posible si los elementos pertenecen al mismo sub-sistema (S). Esto es, si $M=\{x,y\}$, entonces xUy si y sólo si $x \in S$ e $y \in S$.

Para el caso del ejemplo planteado anteriormente, la relación U solo es posible aplicarla entre a,b ; c,d,e ; o f,g,h .

RE2: La relación de uso (\hat{U}) que se da entre elementos de tipo sub-sistema (S) pueden ser entre aquellos que están en la misma o en diferente capa (C). En caso de los sub-sistemas que estén en capas diferentes, éstas deben estar contiguas,

es decir que debe darse una relación de usa-capas (\bar{U}) entre las capas donde se dé la relación antes referida. Esto puede ser expresado de la siguiente manera: dado $S=\{x,y\}$ y $C=\{c_1, c_2\}$, siendo $x \in c_1, y \in c_2$, si $x \hat{U} y$ entonces se debe cumplir $c_1 \bar{U} c_2$.

Para el caso del ejemplo planteado anteriormente, podría existir una relación $S_1 \hat{U} S_2$ sin ninguna restricción (porque están en la misma capa). En cambio, si $S_2 \hat{U} S_3$ entonces también deberá cumplirse $C_1 \bar{U} C_2$, ya que S_2, S_3 están en diferente capa.

RE3: La relación de uso-capas (\bar{U}) sólo se deberá tener entre capas continuas, donde una capa de nivel n use a una capa de nivel $n-1$ (pero no pueda usar directamente a una de nivel $n-2$), esto significa que debe haber una seriación entre ellas, restringiéndose a que una capa de un nivel n use a una capa de nivel $n-2$. Por lo que si $C=\{c_1, c_2, c_3\}$, y se tiene que $c_1 \bar{U} c_2$, y $c_2 \bar{U} c_3$, entonces $c_1 \bar{U} c_3$, o $c_3 \bar{U} c_1$ no son permitidos.

Con estas restricciones y la especificación del modelo de la vista modular se procede al diseño de su meta-modelo. Para lo cual se usará la representación gráfica del lenguaje MOF (OMG QVT, 2005) como se muestra en la Figura 5.1. Vale la pena aclarar que una clase MOF representa a una meta-clase, pues como ya se ha mencionado antes, un meta-modelo es un modelo de un lenguaje de modelado (Seidewitz, 2003).

En el meta-modelo descrito se establecen relaciones de asociación binarias entre meta-clases (asociación simple, composición y generalización). Cada vinculo o relación entre dos meta-clases está etiquetado en ambos extremos (a excepción de la generalización) para indicar la navegabilidad entre una y otra meta-clase, además de la especificación de su cardinalidad.

Así por ejemplo la composición entre las meta-clases *Modulo* y *Funcion* del meta-modelo referido en la Figura 5.1 se lee: una instancia *Modulo* está compuesta por una instancia *Funcion* (etiqueta: *funcion*), y una instancia de *Funcion* está contenida (etiqueta: *cont_funcion*) en una instancia *Modulo*. Lo mismo se aplica para todas las demás asociaciones.

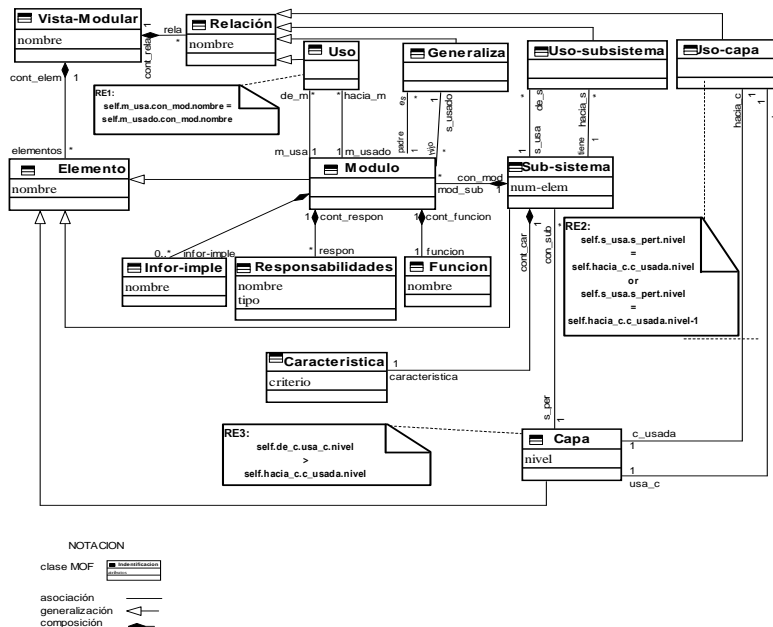


Figura 5.1 Meta-Modelo de la Vista Modular

El meta-modelo de la vista modular presentado en la Figura 5.1, tiene como meta-clase raíz a *Vista-Modular*, que contiene como atributo a nombre que sirve para identificar a una vista modular a nivel de modelo.

La meta-clase raíz está compuesta básicamente de las meta-clases de tipo *Elemento* y de tipo *Relacion*, los cuales definen propiamente a una vista arquitectónica de software, pero lo que caracteriza a una vista modular es el tipo de elementos

y relaciones que la conforman. Como se puede apreciar en la Figura 5.1, cada uno estos tipos tiene como atributo común a nombre, el cual es generalizado en su meta-clase correspondiente, este atributo permite la identificación de cada elemento a nivel de instanciación.

Los tipos de meta-clases contenidos en Elemento son: *Modulo*, *Sub-sistema* y *Capa*. *Modulo* está compuesto por una instancia de *Funcion*, y varias instancias de *Responsabilidades* y de *Infor-Imple*. Puesto que la función describe la naturaleza de una instancia módulo (enlazada con un requisito), sólo hay una, en cambio un *Modulo* puede llegar a tener varias *Responsabilidades* ya que ellas corresponden a tareas específicas que competen a su función.

En el caso de información de implementación (*Infor-imple*), se refiere a información temprana que se aporta sobre aspectos de implementación deseable para el módulo, por lo que puede haber ninguna o varias instancias de esta información.

Por otra parte, también se puede observar que una instancia de la meta-clase *Sub-sistema* está compuesta por una o varias instancias de *Modulo*, cuya cardinalidad de esta composición está comprendida en el atributo num-elem, que sirve para controlar el número de instancias de *Modulo* contenidas en un *Sub-sistema*. La otra meta-clase que es parte de *Sub-sistema* es *Característica*, sirve para indicar el criterio que se aplicará para que un *Modulo* pertenezca a un *Sub-sistema* específico.

El tercer tipo de elemento del meta-modelo es *Capa*, se puede apreciar que esta meta-clase está asociada con varias instancias de *Sub-sistema*. Nótese que *Capa* no se vincula directamente con las instancias de la meta-clase *Modulo*,

estas deben forzosamente estar incluidas dentro de una instancia de un *Sub-sistema*, lo cual hace que los vínculos entre una y otra capa sean a nivel de Sub-sistema. Por su parte una instancia de Capa se asocia con otra instancia de ella misma para lograr la seriación entre las capas a través de la meta-clase *Uso-capa*.

Los otros tipos de meta-clases de esta vista modular, corresponde a los tipo de relaciones que se pueden establecer entre sus elementos, las cuales son: *Uso*, *Generaliza*, *Uso-subsistema* y *Uso-capa*, en otras fuentes bibliográficas son designadas como estilos arquitectónicos (Shaw y Garlan, 1996)(Clements et al 2002).

La relación *Uso*, vincula a instancias de *Modulos* entre sí, como se puede apreciar en la Figura 5.1 se usan dos enlaces de asociación para estos vínculos, un enlace va de la instancia que usa (a un módulo), y el otro enlace va hacia la instancia usada. La restricción RE1 aplicables a esta relación está expresada en el lenguaje OCL, descritas dentro de un comentario (Figura 5.1).

La relación *Generaliza* indica que varios módulos serán generalizados por otro, es decir que uno de ellos contendrá responsabilidades, o características de implementación que serán compartidas por otros, estableciéndose una relación jerárquica (padre-hijo). Esta jerarquía considera que un padre puede tener varios hijos, pero un hijo sólo tiene a un padre.

La relación de *usoSubsistema* es semejante a la de uso para módulos, sólo que en este caso los elementos son subsistemas, esto significa que un sub-sistema puede usar a uno a varios sub-sistemas, además debe de cumplirse la restricción RE2 para garantizar que los subsistemas que guardan esta relación estén en capas que estén vinculadas

entre sí. Dicha restricción se incluye en el meta-modelo expresada en OCL, dentro de un comentario (Figura 5.1).

Por último la meta-clase *Uso-capa* indica la relación que hay entre capas, usando dos enlaces de asociación, una que va de la capa que *usa* y la otra a la capa que es *usada*, en forma semejante a la relación *Uso*. Aquí se le agrega la restricción RE3 para lograr que una instancia de *Capa* sólo se asocie con la de su nivel inmediato inferior, esto es expresado en el lenguaje OCL incluida en el comentario respectivo (Figura 5.1).

Con esto se tiene la especificación de meta-clase de la primer vista, ahora se continúa con el diseño del meta-modelo de la siguiente vista.

5.2 Diseñado el meta-modelo de la vista componentes-y-conectores

La vista de componentes-y-conectores es la vista que tiene más tipos de relaciones, dado que sus elementos pueden desempeñar varios roles que dan origen a una gran diversidad de relaciones.

Para analizar a sus elementos y relaciones se inicia por representar a esta vista de componentes-y-conectores como:

$$V_{cc} = \{E_{cc}, R_{cc}\}$$

Donde E_{cc} es el conjunto de elementos de este tipo de vista, y R_{cc} representa el conjunto de relaciones fundamentales entre sus elementos. Por relaciones fundamentales se consideran aquellas que han sido identificadas como estilos arquitectónicos (Shaw, 1996), dando la posibilidad de incluir

otros tipo de relaciones formadas a partir de las fundamentales.

Siendo:

$E_{cc} = \{C, K\}$, donde C: es el conjunto de componentes, K: conjunto de conectores. Estos conjuntos a su vez comprenden a tipos más específicos (detallados en la sección 3.6), los cuales se describen a continuación.

$C = \{FI, CL, SE, DS, UC, DA\}$, donde cada elemento corresponde a un tipo de componente, FI: *filtro*, CL: *cliente*, SE: *servidor*, DS: *dualidad de servicios*, UC: *unidad concurrente*, DA: *datos compartidos*.

$K = \{LL, TU, EV, AD, EN, FL, AR, DI\}$, donde cada elemento identifica a un tipo de conector. LL: *llamadas a procedimientos*, TU: *tuberías*, EV: *eventos*, AD: *acceso de datos*, EN: *enlace*, FL: *flujo*, AR: *arbitro*, DI: *distribuidor*.

$R_{cc} = \{TF, CS, IS, PC, DC, CP\}$, donde los elementos corresponden a los tipos de relación fundamentales (estilos arquitectónicos) TF: *tubería-filtro*, CS: *Cliente-Servidor*, IS: *Igualdad-de-servicios*, PC: *Escritor-lector*, DC: *Datos-compartidos*, CP: *Comunicación de procesos*.

En cada tipo de relación se vinculan diferentes tipos de componentes, usando distintos tipos de conectores que imponen restricciones particulares.

Por esta razón, en lugar de elaborar un meta-modelo global que comprenda a todas las posibles condiciones que se pueden dar, se crea primero un meta-modelo básico de esta vista donde se plasma los principales elementos que lo conforman, y a partir de éste se deriva un meta-modelo para cada tipo de relación aquí contemplada. De esta manera se

procede a hacer más *claro* y *flexible* al modelado y al proceso de transformación.

Un modelo *claro*, en el sentido que se evita el uso excesivo de restricciones (mediante OCL), y se centra la atención en la expresividad del lenguaje de meta-modelado (MOF).

Un modelado *flexible*, porque separando la parte básica de los tipos de relación se permite incorporar otros tipos de relaciones adicionales a las aquí consideradas, como a los *aspectos* incluidos en PRISMA (Perez J., 2006). Además con ello se logra llevar a cabo la propuesta del proceso de transformación planteado en la Figura 4.5, usando al meta-modelo base de la vista para generar un modelo básico (a nivel de modelo), y dependiendo del comportamiento de sus elementos se empleará posteriormente uno u otro meta-modelo especializado para conseguir su refinamiento.

Meta-modelo base

El meta-modelo *base* de la vista de *Componentes-y-Conectores* se muestra en la Figura 5.2, ahí se aprecia que un modelo de esta vista se forma por sus elementos fundamentales y su meta-clase *Relacion*. Todos los vínculos entre sus meta-clases están etiquetadas en ambos extremos para poder navegar entre ellos (como se hizo con el meta-modelo de la vista modular). Nótese que a los dos elementos fundamentales de esta vista (*Componentes* y *Conectores*) se ha agregado un tercer elemento, el *Comp-compuesto*, este es una meta-clase que está formada por elementos fundamentales para crear un nivel de anidamiento de componentes, además incluye a un elemento *Nivel* el cual es útil para poder ubicar diferentes niveles de plataformas (*Comp-compuesto* es similar a un *Sistema* definido en el meta-modelo de PRISMA (Pérez J., 2006)). Los tres

elementos comparten los atributos *nombre* y *tipo*, el primero sirve para identificarlo y el segundo hace referencia a uno de los tipos que han sido descritos en sus conjuntos correspondientes (C, y K).

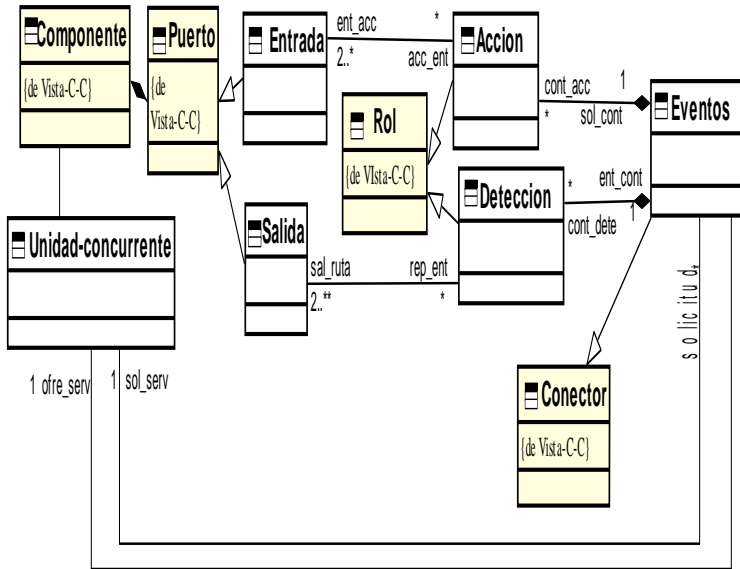


Figura 5.2 Meta-modelo base de la vista de Componentes-y-Conectores

Los *componentes* por su parte están constituidos por los servicios que una instancia de éstos presta, y por los puertos por donde estos se llegan a comunicar. Los *servicios* de un componente, como ya se mencionó anteriormente, modelan su comportamiento al tiempo de ejecución. Los puertos son los canales por donde se accede al *servicio*, el de *entrada* es usado para comunicarle información que éste solicita y el de *salida* se usa para comunicar sus resultados, por eso su *cardinalidad* mínima ha sido fijada con el valor de 1, pues se considera que algo debe entrar o salir de un componente.

En el caso de los *conectores*, ellos están integrados por sus *roles* que son los que llegan a interactuar con los *puertos*,

denotándose esto último en los enlaces de asociación mostradas en el meta-modelo. Esta interconexión entre *roles* y *puertos* permite la interacción entre componentes a través de los conectores, la cual sigue un patrón determinado por el tipo de relación que se forma entre estos elementos (especificadas en la sección 3.6), por esto, la meta-clase *Rol* incluye como uno de sus atributos a *tipo* para que pueda adaptarse a cada clase de relación. Cada una de las relaciones incluidas en el conjunto R_{cc} es presentada a continuación mediante sus meta-modelos correspondientes, los cuales vienen a complementar al meta-modelo base de esta vista.

Relación filtro-tubería

El meta-modelo de esta relación derivado del meta-modelo base es mostrado en la Figura 5.3. La meta-clase *Filtro* es una especialización de *Componente*, y tiene como atributo a *criterio* que es usado para realizar el filtrado. El resultado del servicio de una instancia de *Filtro* se envía por el puerto de tipo *Salida* que se vincula con una instancia de otro *Componente* a través del un conector de tipo *Tubería* mediante un rol *Enlace*.

Nótese que la asociación entre las meta-classes *Filtro-Tubería-Componente* modela lo que un componente *Filtro* comunica a un *Componente* de cualquier tipo, en lugar de hacerlo con uno de tipo *Filtro*, ya que lo que importa de esta relación es lo que sale del primer componente (*Filtro*) sin importar a donde se dirige. Con esto se pueden formar enlaces de esta relación con otras clases de relaciones.

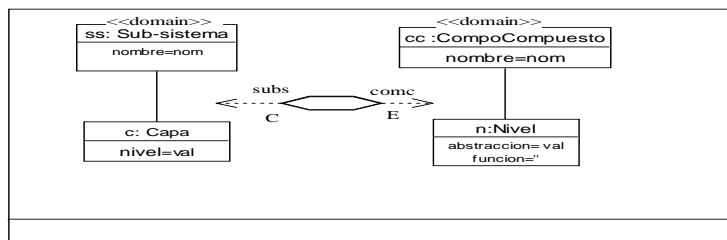


Figura 5.3 Meta-modelo vista C-C con estilo Filtro-Tubería

Relación *Cliente-Servidor*

En esta relación se crea un meta-modelo que emplea a dos tipos de componentes que son *Servidor* y *Cliente*, y a dos conectores tipos *Llamadas-proced* (abreviación de llamadas a procedimientos) y *Distribuidor* como se puede apreciar en la figura 5.4. Las instancias de *Cliente* se vinculan con las de *Servidor* mediante *Llamadas-proced*, a través de las solicitudes que los primeros realizan en función de sus requerimientos, y el *Servidor* se encarga de entenderlas.

La comunicación entre los dos tipos de componentes se establece mediante sus puertos de entrada y salida correspondientes con los roles de los conectores, de la siguiente manera: por el puerto de salida de un *Cliente* se realiza un *Envío* (rol) por *Llamadas-proced* (conector), que resulta en una *Invocación*(rol) hacia el puerto de entrada del *Servidor*. Éste último a su vez entrega el resultado para *Repartir* (rol) por medio de un *Distribuidor* (conector) que a su vez lo encausa por *Elegir-ruta* (rol) hacia el puerto de entrada del *Cliente*. De esta manera se cierra el ciclo de comunicación entre las instancias *Cliente* con *Servidor*.

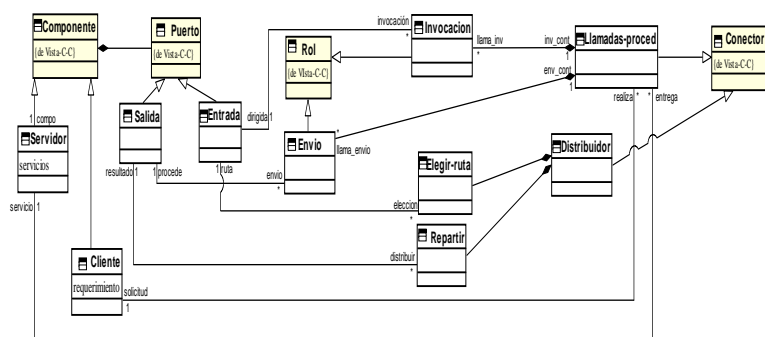


Figura 5.4 Meta-modelo de la relación Cliente-Servidor

Relación *igualdad-de-servicios*

Esta relación también conocida como *par-a-par*, establece que un componente podrá ser tanto *cliente* como *servidor*, por ende se ha incluido a un componente especializado llamado *Dualidad-de-Servicios* para realizar estas tareas. Este componente usa a dos conectores uno tipo *Distribuidor* y el otro tipo *Flujo* para establecer el vínculo entre las instancias de este tipo de elemento, como puede apreciarse en su meta-modelo mostrado en la Figura 5.5.

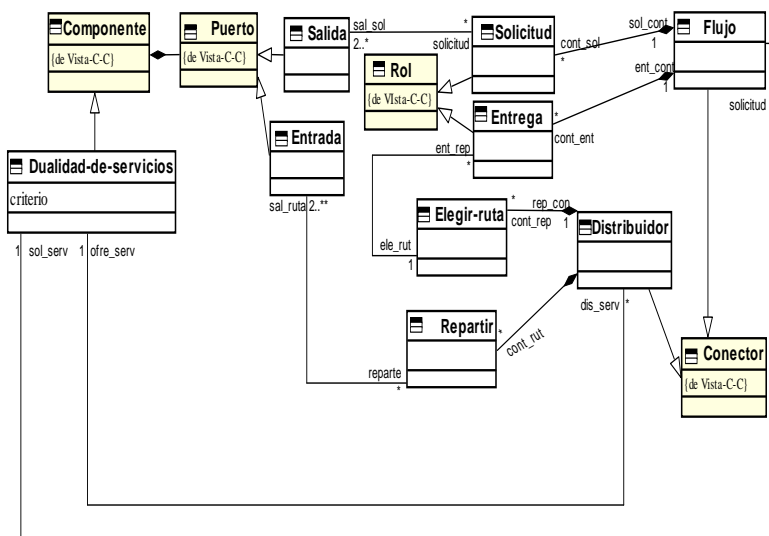


Figura 5.5 Meta-modelo de la relación igualdad-de-servicios

El conector *Flujo* sirve tanto para recibir las solicitudes de servicio como para recibir las respuestas de ellos a través del rol *Solicitud*, esto lo canaliza al conector *Distribuidor* mediante los roles *Entrega* y *Elegir-ruta*. El *Distribuidor* envía la información (solicitudes y servicios) por medio de su rol *Repartir* a los componentes correspondientes tipo *Dualidad-de-servicios*, ya sea se trate de una solicitud como *cliente* o la respuesta del servicio como *servidor*.

Relación escritor-lector

Este tipo de relación no requiere un componente especializado, como en las relaciones anteriores, ya que su comportamiento radica básicamente en la combinación de los conectores de tipo *Eventos* y *Distribuidor*, por lo tanto su meta-modelo está conformado por estos conectores que se

vinculan con un componente de tipo genérico, como se muestra en la Figura 5.6.

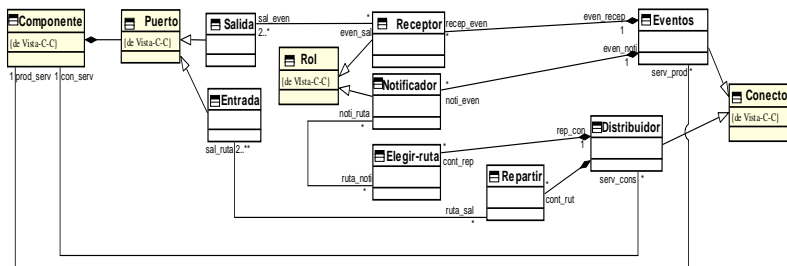


Figura 5.6 Meta-modelo de la relación escritor-lector

El comportamiento del *escritor* se modela haciendo que los componentes se enlacen con el conector de tipo *Eventos*, a su vez este conector notifica lo que se publica al conector tipo *Distribuidor* que se encarga de repartirlo hacia otros componentes que son los *lectores*, como se aprecia en la Figura 5.6. La asociación entre las clases *Componente* y *Eventos* denota el papel *escritor*, y la asociación entre las clases *Componente* y *Distribuidor* denota el papel *lector*, que caracteriza a esta relación.

Relación *acceso a datos*

El acceso a datos está caracterizado tanto por un componente especializado para acceder datos aquí denominado *Datos-compartidos*, y por un conector llamado *Acceso-a-datos*, como puede verse en la Figura 5.7. El conector referido denota a través de sus roles las operaciones que los componentes realizan con los datos (lectura, escritura y actualización). La forma en que el tipo componente *Datos-compartidos* accede a los datos (base de

datos) queda oculta como parte de su comportamiento, y los datos no son mostrados aquí en forma explícita por estar fuera de entorno de los componentes arquitectónicos (es decir no modelan la ejecución de elementos software).

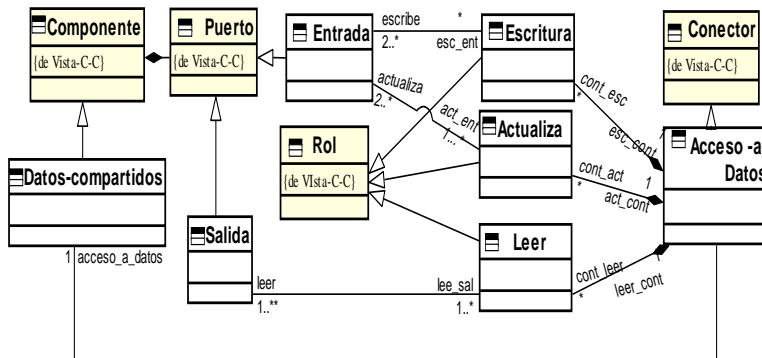


Figura 5.7 Meta-modelo de la relación acceso-a-datos

Relación *comunicación de procesos*

Esta relación se centra en la manera en que los componentes interactúan entre sí en forma simultánea, de ahí que se emplee un tipo de componente denominado *Unidad-concurrente* para generalizar a todos los elementos que denotan este comportamiento como son los procesos, tareas o hilos de control. Este tipo de componente es coordinado por el conector tipo *Arbitro*, como se puede ver en su meta-modelo mostrado en la Figura 5.8. Una instancia de *Unidad-concurrente* puede ser un *proceso*, una *tarea* o un *hilo de control*.

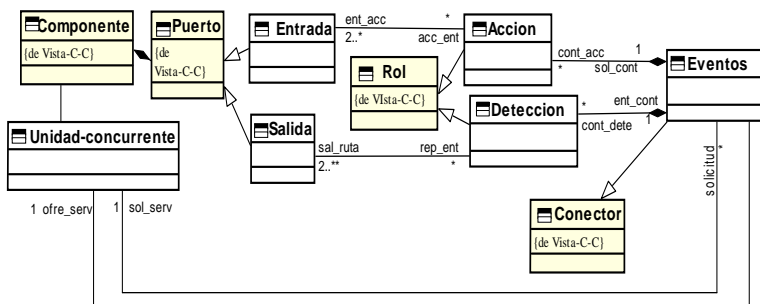


Figura 5.8 Meta-modelo de la relación comunicación-de-procesos

5.3 Meta-modelo para el comportamiento de componentes

Con el objeto de llegar a conformar las relaciones específicas de la vista componentes-y-conectores (V_{cc}), tales como tipo *filtro*, *cliente-servidor*, es necesario realizar primero un análisis del comportamiento sobre los componentes que integran la vista base, ya que en función de su comportamiento se determinará cómo cada conector interactúa con los componentes.

El comportamiento se manifiesta en las *interacciones* entre los componentes al detallar lo que cada servicio produce o requiere de otro. Las *interacciones* constituyen un mecanismo común para describir lo que los sistemas producen a diferente nivel de detalle, en el ámbito de la arquitectura de software son útiles para describir al comunicación entre los elementos de una vista como se señala en (Hofmsteir, 2005).

En la vista de *componentes-y-conectores*, una *interacción* es usada para especificar la manera en que los conectores vinculan a los componentes, definiendo el tipo y número de conectores y puertos requeridos, con lo cual se facilita la

tarea de identificar cual relación representa mejor su comportamiento resultante (*filtro, cliente-servidor, etc.*).

Una *interacción* se forma al desglosar cada servicio conforme se incorporan los diferentes escenarios de los requisitos detectados por el arquitecto, combinando los requisitos funcionales con los de calidad. Cada escenario hace que los servicios que le corresponde a los componentes se vayan desglosando de acuerdo a la secuencia y a las operaciones específicas que ellos proveen o requieren.

Las *interacciones* entre los componentes han sido modeladas en el trabajo presentado en (Limon y Ramos, 2006), dónde se usaron los *mapas de casos de usos* para detectar posibles cruzamientos de *interacciones*, sin embargo estos sólo revelan los componentes implicados y la ruta que sigue los escenarios. En este caso lo que se requiere es detallar las posibles interacciones que se tienen entre ellos, centrando la atención en los servicios que llegan a requerir y proporcionar. Esta *interacción* es representada mediante un modelo de secuencias haciendo una adaptación del modelo respectivo presentado en UML 2.0 (OMG, 2004), en esta tesis se considera que los objetos son instancias de componentes, y las operaciones son los servicios que se proveen y requieren.

La propuesta hecha para modelar las *interacciones* entre componentes es ilustrada mediante un ejemplo, el cual se muestra en la Figura 5.9. En este ejemplo se presenta a cuatro objetos componentes (o_1, o_2, o_3, o_4) y dos interacciones (M y N). Cada interacción agrupa a un conjunto de *mensajes* que se comunican entre los *servicios* de los componentes. Los servicios de cada de componente son fragmentados con el propósito de hacer énfasis en la secuencia de tiempo en que se activan los *mensajes*. La activación de los *mensajes* obedece a los eventos

provocados por actores o condiciones de ejecución del sistema, esto último queda fuera del modelo arquitectónico, sólo se consideran a sus productos que son los *mensajes* y la secuencia en que estos ocurren.

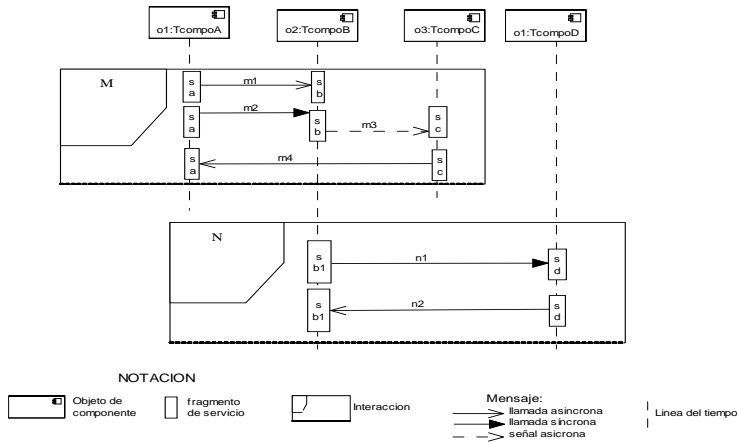


Figura 5.9 Ejemplo de un modelo de interacciones entre componentes

Un *mensaje* es el medio por el cual se comunica un fragmento de servicio con otro, puede ser la llamada de una parte de un servicio en forma síncrona o asíncrona, e incluir información adicional, o bien ser sólo una señal de activación. En el diagrama de secuencia cada tipo de *mensaje* se representa con flechas distintas para diferenciarlos, como se aprecia en las interacciones del ejemplo mostrado.

La *interacción M* del ejemplo (Figura 5.9), comprende a tres objetos componentes que se comunican entre sí, el objeto o1 se comunica con el objeto o2 mediante los *mensajes m1* asíncrono y *m2* síncrono, en ese orden (secuencia), los dos *mensajes* provienen del servicio sa del primer objeto y se dirigen al servicio sb del segundo objeto. A su vez el objeto

o2 envía al objeto *o3* una señal de activación (*m3*). Y por último el objeto *o3* envía un mensaje asíncrono al objeto *o1*.

Por otra parte la interacción *N*, modela la comunicación entre los objetos *o2* y *o4*, cada uno envía un mensaje al otro mediante sus servicios respectivos. Lo importante que hay que destacar en esta segunda interacción es que el objeto *o2* envía el mensaje síncrono *n1* del servicio *sb1* al servicio *sd* del objeto *o2* con los servicios *sd* del objeto *o4*, mediante un mensaje síncrono y asíncrono respectivamente.

El meta-modelo que capta a los elementos clave de un modelo de *interacciones* como el presentado en el ejemplo anterior, se muestra en la Figura 5.10. En el meta-modelo de *interacciones* se describe cómo la clase *Interaccion* está compuesta básicamente de instancias de la clase *Mensaje*, por ser el medio fundamental de comunicación entre objetos componentes. En los atributos de *Mensaje* se incluye al tipo de llamada (llamada síncrona, llamada asíncrona, o señal asíncrona), y a la información que se puede transmitir mediante sus parámetros.

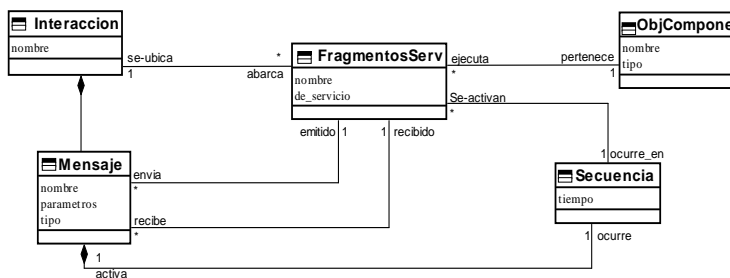


Figura 5.10 Meta-modelo de interacción

Por otra parte, el enlace de asociación entre la clase *Interaccion* con la clase *FragmentosServ* (fragmentos de servicio) denota que estos últimos son abarcados en una sola interacción, y que pertenecen a un solo servicio de su

objeto de componente correspondiente. Es decir no se considera a fragmentos de un servicio esparcidos en varias interacciones. Para que un fragmento haga referencia a su servicio se considera al atributo *de-servicio*, con el cual se establece su enlace.

La comunicación de un *mensaje* desde un fragmento a otro se modela mediante una doble enlace de asociación entre las clases *Mensaje* y *FragmentosServ*, una enlace indica al fragmento que lo envía y la otra indica al fragmento que lo recibe. La secuencia en que ocurren los mensajes se modela mediante el tiempo en que este ocurre, con ello se establece una seriación de estos mensajes. Se considera que el valor del tiempo de la secuencia será igual tanto para el fragmento que emite un mensaje como para el que lo recibe, puesto que se considera que el retardo en su recepción no afecta para diseño de la arquitectura.

Los elementos del meta-modelo de interacción lo constituyen los objetos de los componentes, ya que son ellos los que ejecutan los servicios, son los protagonistas de las interacciones. Por su parte los servicios (de los componentes), presentan varias etapas de ejecución cada vez que existe una comunicación entre uno y otro servicio, de acuerdo a la traza de los escenarios que se vayan identificando, ellos producen información que es comunicada a servicios de otros componentes o bien llaman o activan a otros servicios.

La forma en se comunican los objetos de los componentes puede ser síncrona o asíncrona. Síncrona significa que el objeto receptor debe de estar listo para aceptar lo que el emisor le envíe, de ahí que esto genere la necesidad de tener dos enlaces entre los componentes a través del conector (como en el caso del ejemplo presentado anteriormente). En la forma de comunicación asíncrona sólo

se envía la información por lo que solo se requiere de un enlace para vincularlos. Lo que se llega a comunicar entre un componente y otro puede ser el resultado del servicio solicitado, un evento que activa a otro servicio, o una invocación de alguna parte de otro servicio a través de un mensaje.

Todos los meta-modelos presentados en la vista de *Componentes-y-Conectores*, como el meta-modelo de la vista *Modular*, han sido implementadas dentro del marco de trabajo de *ECLIPSE*, con el objeto de contar con las bases para el proceso de transformación entre vistas en forma automática. Esta implementación puede verse en el Anexo 1. Lo que continua ahora es el establecimiento de las relaciones entre los meta-modelos de estas vistas.

5.4 Correspondencias entre la vista modular y la vista de componentes-y-conectores.

Cuando se planteó en el estándar 1471 el meta-modelo de una arquitectura, no se previó las relaciones que se podrían establecerse entre una y otra vista, recientemente se han hecho observaciones a este respecto dónde se señala la necesidad de establecer correspondencias entre diferentes vistas. Por ejemplo en (Emery y Hilliard, 2008), se menciona la necesidad de establecer esta correspondencias, presentándolo como algo que debería de hacerse aunque sin profundizar en el tema. Esto ya comenzó a ser tomado en cuenta, como se manifiesta en el ISO 42010 (ISO/IEC 42010, 2007) que incluyó al estándar 1471 con algunas adecuaciones, como el concepto sobre la relación entre dos vistas llamado *correspondencia de vistas*, definiéndolo de la siguiente manera:

‘Una correspondencia de vistas (CV) registra una relación entre dos vistas arquitectónicas para capturar: una relación de consistencia, una relación de trazabilidad, una restricción u obligación de una vista sobre otra’.

Este concepto es usado en esta tesis para establecer la correspondencia entre las dos vistas aquí presentadas (la modular y la de componentes-y-conectores) a nivel de meta-modelo, agregando las siguientes anotaciones:

- Una CV entre la V_m y la V_{cc} es una relación inter-modelos, es decir entre dos modelos diferentes, que tienen distintos elementos y relaciones, por lo tanto es una relación horizontal.
- Una CV entre la V_m y la V_{cc} se establece a nivel de meta-modelo, la cual permite mantener la relación de trazabilidad y la de consistencia a nivel de modelos, que es garantizada por el proceso de transformación de MDA aquí seguido.

La *correspondencia* entre estas dos vistas es entendida en el marco de los criterios usados para hacer la separación de *concerns*, pues al establecer las fronteras entre cada vista, también se tiene que establecer la forma de vincularlas. En este caso los criterios usados para la separación son tres:

- i) El ámbito donde surgen las unidades de software y su propósito en cada vista.
- ii) La naturaleza de estas unidades.
- iii) El tipo de relaciones que llegan a formar dichas unidades.

El primer criterio es el más genérico y le da sentido a los demás, los otros dos son útiles para establecer las correspondencias con detalle.

En la Tabla 5.1 se resumen los *concerns* comprendidos por esos criterios en cada vista.

Tabla 5.1 Concerns considerados en las vistas V_m y V_{cc} , de acuerdo a los criterios empleados

Vista Criterio	Ámbito y propósito	Naturaleza de los elementos	Tipo de relaciones
V_m	Espacio del problema Descomposición del sistema	Funciones Responsabilidades Tareas compartidas	Dependencia Generalización Agregación
V_{cc}	Espacio de la solución Operación del sistema	Operaciones computacionales Servicios Comunicación	Interacciones Coordinación Acceso a datos

Las correspondencias entre una vista y otra se derivan de los vínculos establecidos entre los *concerns* de las vistas, agrupados en cada criterio. Los vínculos entre las vistas V_m y V_{cc} pueden verse en dos sentidos¹, en el sentido de la V_m a la V_{cc} ($V_m \rightarrow V_{cc}$), y en el sentido inverso ($V_{cc} \rightarrow V_m$), esto es aplicable a cada criterio usado para separar *concerns*.

En el caso del criterio sobre ámbito y propósito, por su carácter genérico, los vínculos entre sus *concerns* no son posibles expresarlos como una relación detallado, pero para los otros dos criterios los vínculos entre los elementos agrupados en sus *concerns* se relacionan mediante el lenguaje QVT-relaciones, donde los elementos son las clases descritas en los meta-modelos mostrados.

En QVT-relaciones, las clases de un meta-modelo se involucran dentro de un dominio constituido en una clase abstracta llamada *Domain*. Esta clase abstracta contiene

¹ Aunque al fin de cuentas son dos caras de la misma moneda

sub-clases concretas que son responsables de establecer los vínculos específicos entre las clases, una de estas sub-clases concretas es la llamada clase *RelationDomain* la cual especifica el dominio de una relación, indicando qué clases se involucran y la forma de hacerlo. En la Tabla 5.2 se muestra un listado de las relaciones identificadas, cada renglón comprende a una relación establecida en el sentido $V_m \rightarrow V_{cc}$ base¹ y a sus clases involucradas en cada vista, donde el nombre de la relación hace alusión a las principales clases asociadas, también se incluye la forma cómo se activa la relación, *top* indica que la relación es activada en forma automática cuando ésta se llegue a ejecutar, y el ‘-’ indica que la relación se activa por medio de otra relación.

Tabla 5.2 Relaciones identificadas en la correspondencia entre V_m y V_{cc} base.

Nombre de la relación	Tipo de relación	Clases involucradas en cada vista	
		V_m	V_{cc}
<i>moduloAcomponente</i>	<i>top</i>	Modulo	Componente
<i>funciónApropiedades</i>	-	Modulo, Función	Componente, Propiedades
<i>responsabilidadAservicio</i>	-	Modulo, Responsabilidad	Componente, Servicio, Puertos
<i>subsistemaAcompocom</i>	<i>top</i>	Modulo, Sub-sistema	Compo-compuesto, componente
<i>capaAnivel</i>	-	Sub-sistema, Capa	Compo-compuesto, nivel
<i>usoAconector</i>	<i>top</i>	Uso, Modulo, Responsabilidad	Componente, Conector, Servicio, Puerto, Rol
<i>usoSubSistemaAconector</i>	<i>top</i>	UsoSubsistema, Sub-sistema	Compo-compuesto, Conector,
<i>generalizaAconector</i>	<i>top</i>	Generaliza, Modulo,	Conector, Componente

A continuación se detalla el dominio de cada una de estas relaciones, describiéndolas y especificándolas por medio de una gráfica de nodos de objetos para representar sus propiedades, enlaces de asociación, y su nodo raíz que está enlazado a la variable raíz del dominio de una relación. Estas gráficas sirven como base para la codificación de las relaciones en QVT-relaciones detalladas en el Anexo B.

¹ Su relación reciproca se da en el sentido $V_{cc} \rightarrow V_m$, e incluye a los mismos elementos.

Relación *moduloAcomponente*.

Con esta relación se pone de manifiesto la correspondencia entre un elemento software que se detecta en el dominio del problema (derivado de los requisitos), con el elemento software que se usará en el dominio de la solución (en tiempo de ejecución). En la Figura 5.9 se especifica esta relación con el nodo raíz en cada dominio (*m* y *c*) que hace referencia a estos elementos.

El dominio de la relación indica que los objetos del dominio Modulo (*mod*) sólo serán verificables, en cambio los objetos del dominio Componente (*comp*) serán creados si estos no existen, lo cual es referenciado por *C* y *E*¹ respectivamente. Esta relación es útil cuando se está realizando el diseño de las vistas, ya que los módulos son creados primero por los arquitectos, y posteriormente al aplicar esta relación se generarán los componentes.

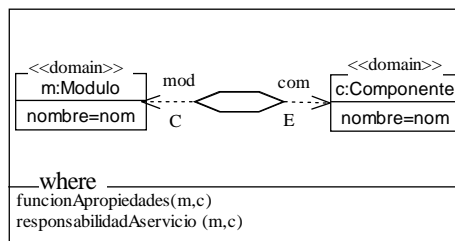


Figura 5.9 Especificación de la relación *moduloAcomponente*

Al crear componentes por medio de módulos, se inicia la construcción de la V_{cc} al desencadenar la activación de otras relaciones que permiten completar los demás elementos de la vista generada. La activación de estas otras relaciones se realiza al concluir la relación *moduloAcomponente*, lo cual se especifica mediante inclusión de las relaciones *funcionApropiedades* y *responsabilidadesAservicios* debajo

¹ Son la letras iniciales de los términos equivalentes usados en el lenguaje QVT-relaciones: **C**heckonly y **E**nforce.

de la clausula *where*, dónde también se le comunican los objetos creados (referenciados por *m* y *c*).

Su relación recíproca en el sentido $V_{cc} \rightarrow V_m$, es usada para los casos en que una vez que se haya generado la V_{cc} se lleguen a detectar nuevos componentes que no hayan sido creados por la V_m , con eso se refuerza la consistencia entre las vistas, propagando los cambios en ambos sentidos.

Relación *funcionApropiedades*.

Una función de un módulo guarda una correspondencia con una de las propiedades de un componente, con la intención de mantener una trazabilidad que ligue a un componente con lo que le corresponda satisfacer del espacio del problema. En la Figura 5.10 se especifica esta relación, mostrándose que ésta se llega a activar como consecuencia de la creación de un componente mediante un módulo (indicado por sus nodos raíces). En este caso el lado derecho del domino de la relación indica que un objeto de la clase *Función* (*f*) crea un objeto de la clase *Propiedad* (*p*) asignándole además del nombre su tipo igual a “*funcion*” que es como se etiqueta a esa propiedad. El lado izquierdo de este dominio indica que un objeto de *Propiedades*(*p*) sólo verifica que se encuentre su correspondiente objeto en *Funcion* (*f*).

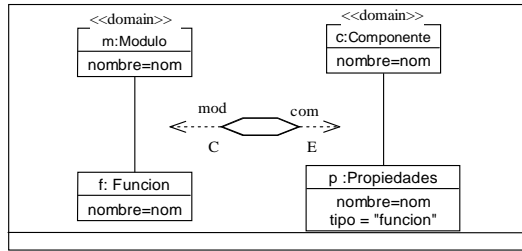


Figura 5.10 Especificación de la relación funcionApropiedades

Relación *responsabilidadAservicio*.

La característica más importante de un componente radica en los servicios computacionales que desempeña, puesto que su interacción define el comportamiento de un sistema. Estos servicios tienen su origen en las responsabilidades que se le asignan a los módulos, y que orientan el desempeño de los componentes. La relación entre estos dos elementos está especificada en la Figura 5.11, ahí se muestra que sus nodos raíces lo forman la clase *Modulo* y la de *Componente*, y de ellos se derivan *responsabilidad* y *servicio* respectivamente. Así cada vez que una *responsabilidad* crea a un servicio éste se le agrega al componente referenciado.

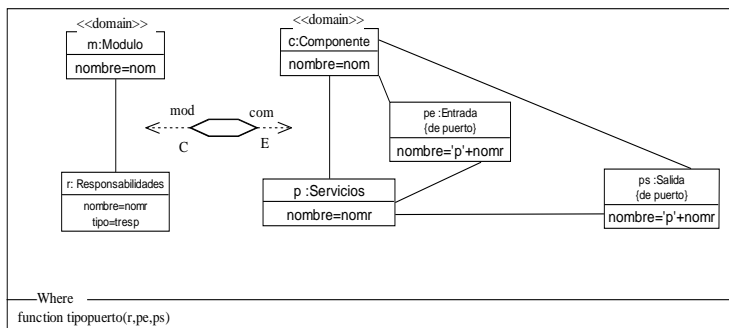


Figura 5.11 Especificación de la relación ResponsabilidadesAservicios

Cada vez que se crea un servicio también se llegan a crear puertos de entrada y/o salida, de ahí la inclusión de la clase *Puerto* en el dominio *Componente*. Un puerto de entrada se crea cuando el tipo de responsabilidad es de tipo *terminal*, esto quiere decir que no llega a involucrar a ningún otro elemento, en cualquier otro caso se crea por lo menos un puerto de entrada y uno de salida, ya que esto indica que el servicio tendrá interacción de información. Esto se hace mediante la función *tipoPuerto* indicada debajo de la clausula *where*.

Relación *subsistemaAcompoCom*.

Este tipo de relación establece vínculos entre elementos compuestos, porque un sub-*sistema* contiene módulos y relaciones, y a su vez el tipo de componente llamado *CompoCompuesto* contiene componentes e interacciones. En la figura 5.12 se muestra la especificación de esta relación, donde se aprecia a sus clases raíces que son las directamente implicadas, y todas las demás clases involucradas. Al llevar a cabo esta relación (ejecutarla), se crean por el lado del domino *CompoCompuesto* tantos componentes como número de elementos tenga el subsistema, esto se hace mediante la función *modAcomp*, donde se le pasa como parámetro los objetos de cada dominio y el atributo *num-elem*. Mientras que por el lado del dominio *Sub-sistema* solo se llega a verificar que existan sus correspondientes elementos.

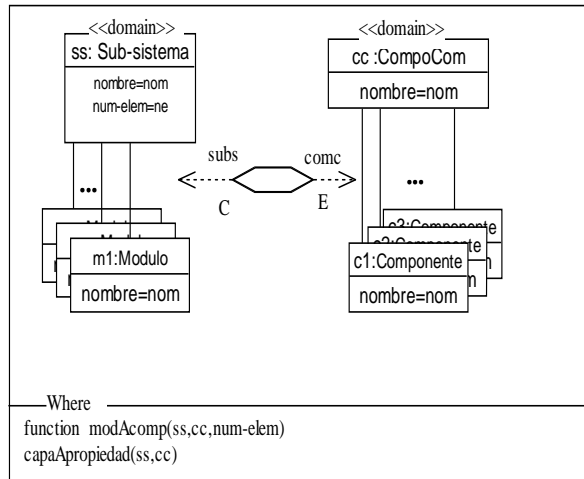


Figura 5.12 Especificación de la relación subsistemaAcompCom

Relación *capaAnivel*.

Esta relación se establece cuando un *sub-sistema* se vincula con un componente compuesto, puesto que todo sub-sistema está asociado con alguna capa, y esta información debe mantenerse también en la vista V_{cc} sólo con el objeto de mantener la trazabilidad con la vista V_m , ya que dentro de los componentes no se tienen diferentes grados de abstracción sino niveles de separación de funciones¹. Esto reafirma la característica que tiene un proceso de transformación en MDA, en el sentido de que no todos los elementos de un modelo deben tener correspondencia con los del otro modelo. La especificación de esta relación es mostrada en la Figura 5.13, como ahí se aprecia sus nodos raíces son las clases de donde dependen capa y nivel, cada vez que se crea un *sub-sistema* se relaciona con la capa que le corresponde entonces al ejecutar esta relación la información un objeto *Capa* crea un objeto *Nivel*,

¹ Aunque muchas veces se confunde el término *capa* con el de *nivel*, estos tienen acepciones diferentes, una *capa* separa abstracciones, en cambio un *nivel* delimita funciones (Clemments et al, 2003, pags. 89-90).

asignándosele el valor (*val*) al atributo *abstracción* de este último objeto, nótese que al atributo *función* se le asigna un valor nulo puesto que su valor se le asignará al momento de refinar la vista correspondiente.

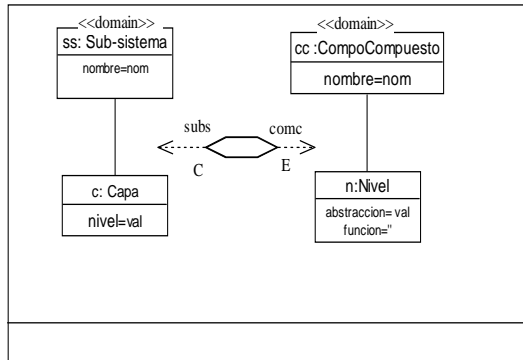


Figura 5.13 Especificación de la relación capaAnivel

Relación *usoAconector*.

La principal forma en que se vinculan los elementos dentro de una V_m es a través de la relación de *uso*, esta relación tiene una correspondencia con la V_{cc} a través de un conector, ya que a través de él los componentes establecen sus interacciones. Por lo tanto cada vez que un módulo *usa* a otro, se genera un conector para vincular a un componente con otro.

La especificación de la relación entre estos elementos es mostrada en la Figura 5.14, ahí se aprecia que las clases *Usa* y *Conector* son las raíces de sus respectivos dominios, cada clase está asociada con los elementos que vincula, *Usa* está asociada con dos objetos de la clase *Modulo* (*ma*, *mb*), y *Conector* con dos objetos de la clase *Componente*(*ca*,*cb*). Así al llevar a cabo esta relación, en su dominio derecho se crea un conector y sus roles (*ra*, *rb*), que estarán asociados a dos

componentes a través de sus puertos (*pa,pb*). También interviene los servicios que son generados. Hay que aclarar que a pesar de crearse la asociación entre un conector con sus componentes, dicha asociación no está completamente definida porque no se conoce el tipo de interacción entre ellos, lo cual será especificado por completo cuando esta relación sea refinada. En el domino izquierdo de esta relación sólo se realizará una verificación de sus elementos.

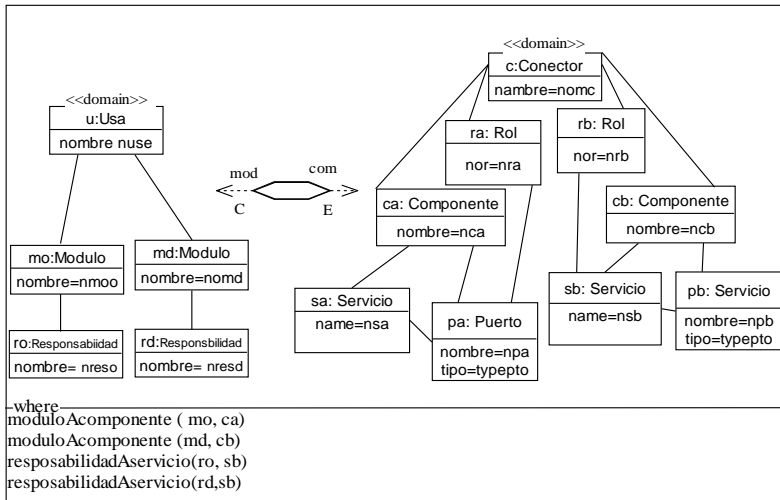


Figura 5.14 Especificación de la relación *usoAconector*

Relación *usoSubSistemaAconector*.

De la misma manera en que *uso* se vincula con conector vía relación de módulos con componentes, también existe un vínculo entre *usoSubsistema* con un conector pero en este caso es vía *sub-sistemas* con componentes-compuestos. Lo que se establece en esta relación es un vínculo entre dos componentes compuestos, lo cual se deriva del vínculo entre dos subsistemas. Por cada objeto de *UsoSub-sistema* se crea un objeto de la clase *Conector*, creando un enlace entre los componentes-compuestos, que corresponden a los

subsistemas de la vista modular, tal como se aprecia en su especificación mostrada en la Figura 5.15.

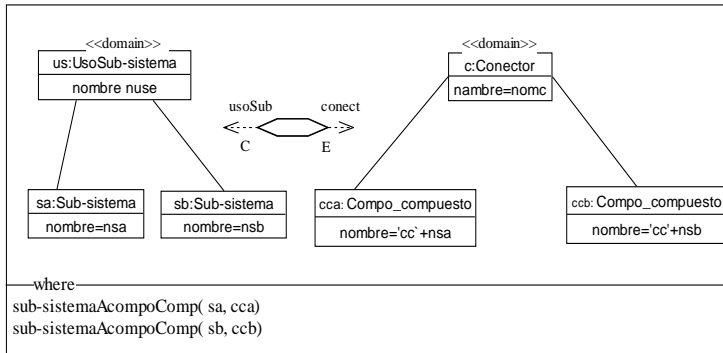


Figura 5.15 Especificación de la relación usoSubsistemaAconector

Relación *generalizaAConector*. Una clase *Generaliza* vincula a módulos que comparten características comunes, lo cual tiene una correspondencia con la clase *Conector* que vincula a componentes, esta correspondencia es especificada mediante la relación *generalizaAconector* que involucra la asociación con sus correspondientes elementos, es decir entre módulos y componentes, la especificación de esta relación es mostrada en la Figura 5.16.

En la relación *generalizaAconector*, se especifica que por cada objeto *Generaliza* que se tenga en un modelo modular, se crea un objeto *Conector* en el modelo de componentes-y-conectores que vincula a dos componentes. Del lado del dominio *Generaliza* se tienen asociados los módulos nombrados *mp* y *mh* que representan un vinculo padre-hijo. Este vínculo se corresponde con una asociación entre componentes usando un conector de tipo *Flujo* para comunicarlos, ya que con este tipo de conector se logra una

interacción entre componentes que comparten información, que es equivalente a la relación *padre-hijo* del nivel modular. Con ello se precisa la interrelación que tiene el conector con los componentes como se puede apreciar en la Figura referida, al designar tanto el tipo de conector a usar (flujo), como los tipos de roles que le corresponden (solicita y entrega), además también se define a los tipos de puertos (entrada y salida) asociados a cada rol.

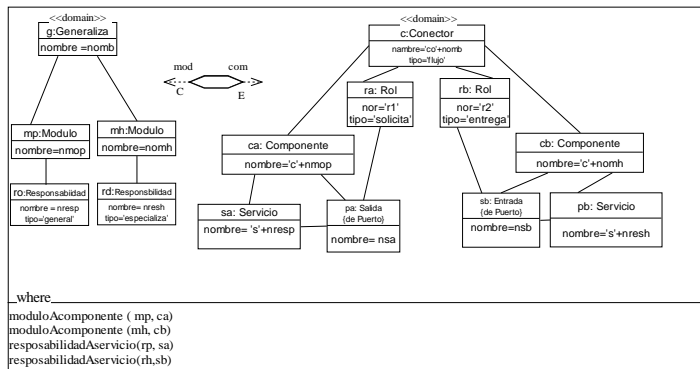


Figura 5.16 Especificación de la relación *generalizaAConector*.
 (b) functionToservice, (c) rUseModToConnector, (d)
 rCompositionModToconn

5.5 Estableciendo relaciones entre interacciones y conectores

El refinamiento de los vínculos entre los *componentes-y-conectores* se inicia estableciendo las relaciones entre su meta-modelo con el meta-modelo de *interacciones*. Para que posteriormente al ejecutar una transformación entre sus modelos respectivos, usando a un modelo de *iteraciones* como fuente, el modelo de *componentes-y-conectores* resultante contará con componentes refinados, porque sus

servicios incorporarán a los mensajes que reciben y emiten, y se determinará el tipo y número de sus puertos. Por su parte los conectores serán refinados porque se establecerán conexiones entre los componentes de acuerdo a los mensajes que se comunican por ellos, definiendo su tipo y número de roles, y se coordinarán las conexiones de acuerdo a la secuencia en que se envíen y reciban los mensajes.

Para establecer las correspondencias entre los elementos del meta-modelo de *interaccion* con los elementos de *componentes-y-conectores*, se ha considerado una relación llamada *interaccionAconector*, formada por el nombre de las clases principales del dominio. La especificación de esta relación es mostrada en la Figura 5.17.

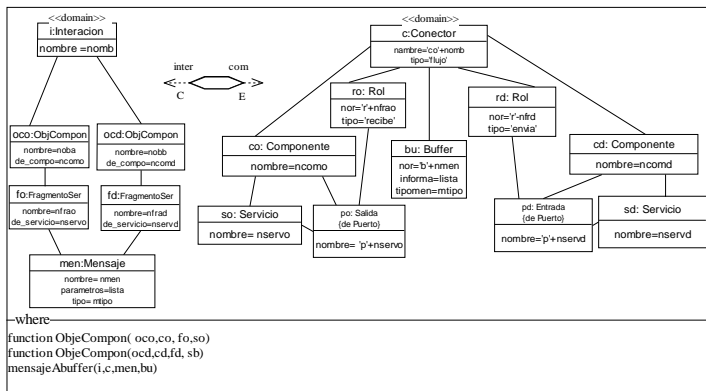


Figura 5.17 Especificación de la relación *interaccionAconector*, (b) *functionToservice*, (c) *rUseModToConnector*, (d) *rCompositionModTocomp*

Como se puede notar en la especificación de la relación *interaccionAconector*, por cada objeto de *Interaccion* (i), se crea un objeto *Conector* (c), los nexos entre los servicios de los componentes se establecen a través de la función *objeCompon* pasándole como parámetros los objetos involucrados de cada dominio, esta función es invocada dos

veces, una para los objetos fuentes y otra para los objetos destinos.

Por su parte la clase *Mensaje* del dominio *Interaccion* que vincula a los objetos destino y fuente de ese dominio, tiene una correspondencia con la clase *Buffer* del dominio *Conector*, esta correspondencia indicada por la invocación de la relación *mensajeAbuffer* (ubicada debajo de la clausula *where*) hace que un objeto *Mensaje* cree un objeto *Buffer* que está asociado con un conector, de esta manera un conector contendrá toda la información común entre los componentes que vincula.

5.6 Generando modelos de la vista de componentes-y-conectores con modelos de la vista de modular

La creación de los meta-modelos y de sus relaciones correspondientes de las dos vistas involucradas son tareas que se hacen una vez o sólo cuando alguno de ellos es sujeto de cambio. Todo esto cobra sentido en el siguiente paso del proceso del MDA, que consiste en la transformación a nivel de modelos aplicado a las vistas arquitectónicas, es decir en la generación de una vista en función de otra. En esta fase una transformación se ejecuta cada vez que se diseña una arquitectura de software para algún sistema o cuando alguna de las vistas cambie por motivos de mantenimiento o evolución.

La transformación de un modelo de vistas se manifiesta cuando se diseña la arquitectura creando primero la vista modular, y luego mediante el proceso de transformación se obtiene la vista de componentes-y-conectores. Existen varias formas de pasar de los requisitos a una arquitectura, por ejemplo dentro del marco del proyecto PRISMA donde se

ubica a esta tesis, se dispone de una estrategia basada en metas para llegar de los requisitos a una arquitectura de software PRISMA (Navarro 2007) siguiendo la aproximación orientada a aspectos. Sin embargo con el fin de centrarse en cómo se transforma una vista en otra, en esta sección se inicia con una vista arquitectónica modular omitiendo el proceso de su diseño¹.

En el proceso de transformación, el modelo de la vista modular es el modelo fuente, y el modelo de la vista de componentes-y-conectores el modelo destino. En una transformación entre estas vistas, se ha identificado que el modelo fuente puede producir dos tipos de modelos destino, dependiendo del tipo de relación que se esté llevando a cabo en la transformación. El primer tipo es aquel en donde se llega a obtener un modelo destino que no requiere refinación, porque las relaciones entre los meta-modelos son suficientemente precisas para especificar cómo los conectores vinculan a sus componentes.

El segundo tipo de transformación es aquel en el que el modelo origen llega a producir un modelo destino que requiere de un refinamiento para especificar cómo los conectores deben interactuar con los componentes. En este segundo caso el refinamiento se lleva a cabo usando un modelo de interacciones, o bien mediante la aplicación de alguno de los estilos de relación entre componentes previamente definidos, tales como el filtro-y-tubería, cliente-servidor, pizarra, etc., ya que estos estilos definen el tipo de conectores. También es posible combinar ambas formas de refinamientos ya que no son excluyentes.

Con el fin de hacer más claro cómo se lleva a cabo cada tipo de transformación, se presentan modelos genéricos de la

¹ El diseño de la vista modular será detallado en el caso de estudio en el capítulo siguiente

vista modular, y se muestra los modelos destinos que se obtienen después de una transformación. En el proceso de transformación se usan los meta-modelos y a sus relaciones creados anteriormente, puesto que constituyen el soporte de este proceso, y se implementa mediante el lenguaje QVT-operacional para su ejecución, ya que este lenguaje ha sido diseñado para tal fin, el cual es ejecutado dentro del ambiente de ECLIPSE. Se presenta el modelo fuente, y el modelo resultante de la transformación para cada caso.

Transformación de una vista que no requiere refinamiento.

Este caso se presenta cuando el modelo de la vista modular contiene relaciones de generalización, es decir donde existe una clase que generaliza a otras como se muestra en la Figura 5.18. El modelo de la vista modular utilizado contiene una clase llamada *General* con sus atributos *función* y *responsabilidad* de donde se derivan tres clases que especializan otras *funciones* y *responsabilidades*, describiendo los valores de cada atributo. Este modelo es creado usando como referencia a su meta-modelo como regulador, su verificación se hace en forma automática gracias a la herramienta de software empleada (ECLIPSE), con la cual también se realiza la ejecución de la transformación mediante el código QTV-operacional y los meta-modelos de ambas vistas.

Al ejecutar la transformación, se genera el modelo de la vista de *componentes-y-conectores* como se aprecia en la parte inferior de la Figura 5.8, en dicho modelo generado se crea un conector de tipo *flujo* que vincula mediante sus roles *entrega* y *solicitud* a los cuatro componentes a través de sus puertos entrada y salida, la especificación detallada de estos elementos es mostrada en la parte inferior de este modelo producido. Nótese que los servicios de los componentes sólo

son modelados como interfaces, puesto que su contenido será especificado a nivel de lenguaje es decir a un nivel inferior de abstracción (nivel de implementación).

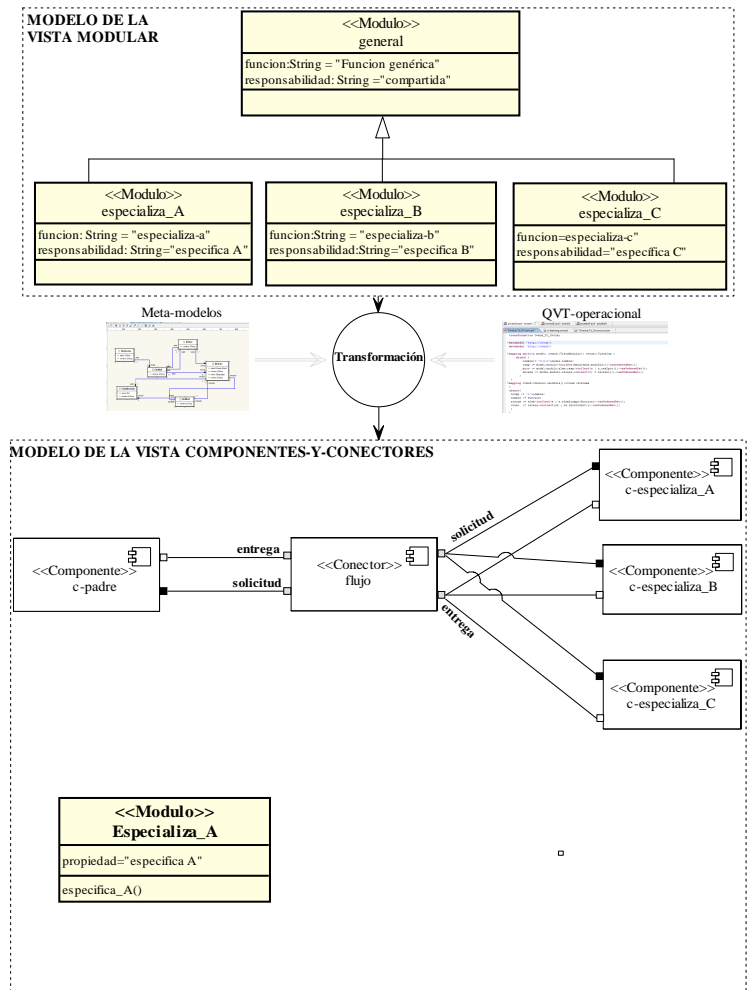


Figura 5.18 Transformación de un modelo de vista modular con relación de generalización a un modelo componentes-y-conectores

Transformación de una vista con refinamiento.

Este tipo de transformación se presenta cuando el modelo origen cuenta con relaciones diferentes a la *generalización*, como las de tipo *usa*, *usa-subsistema* o *usa-capas*, las cuales por sí mismas sólo generan una relación lineal entre los componentes producidos en la vista de componentes-y-conectores al aplicarse una transformación. Donde el conector sólo sirve como enlace entre los componentes, pero no se define ningún rol específico de cómo interactúan, esto es por no contarse con información adicional necesaria para hacerlo.

Para especificar detalladamente la forma en que interactúan los componentes se realiza una segunda fase llamada *refinamiento*. Esta fase consiste en usar a un modelo de *interacciones* que contiene la información sobre el comportamiento de los componentes para ser aplicada a la vista de *componente-y-conectores* que se obtuvo en la transformación anterior. El resultado de este refinamiento hace que el conector establezca roles y enlaces entre los componentes de tal manera que se pueda definir qué y cómo se comunican los componentes.

Para ilustrar el proceso de *transformación y refinamiento* se presenta un ejemplo que utiliza a un modelo de la vista modular que contiene relaciones del tipo <<*usa*>>, como es mostrado en la Figura 5.19 (a). El modelo contiene cuatro módulos con sus propiedades y relaciones definidas, los módulos que *usan* a otros son identificados con el nombre *dependiente_*, y los *usados* llevan el nombre de *independiente_*, se han elegido esos nombres con el objeto de hacer evidente las relaciones de dependencia que se llegan a formar. Las propiedades de cada módulo muestran los valores de sus funciones y responsabilidades.

Al ejecutar el proceso de transformación, el modelo de la vista modular llega a producir el modelo de la vista *componente-y-conectores* mostrado en la Figura 5.19 (b). El modelo resultante contiene los componentes con su información correspondiente, y a los conectores que los vinculan. Sin embargo los conectores que unen a los componentes no especifican cómo se comunican los servicios de estos últimos, ya que la información que permite especificarlo se deriva de la interacción de las ocurrencias de los componentes, la cual será proporcionada por los diseñadores y arquitectos mediante el modelo de *comportamiento*, que su vez es derivado de la especificación de los requisitos.

En este caso el modelo de *interacción* mostrado en la Figura 5.19 (c), presenta algunas ocurrencias de los componentes que se comunican entre sí a través de diversos tipos de mensajes. En este modelo se han propuesto dos interacciones con el propósito de ilustrar cómo se realiza el refinamiento de las relaciones entre los componentes. La primera interacción llamada *X*, establece una comunicación de mensajes entre ocurrencias de los componentes *C-dependiente_a*, *C-dependiente_b* y *C-independiente_d*. La segunda interacción realiza una comunicación entre los componente *C-dependiente_c* y *C-independiente_b*, en donde el primero envía un evento al segundo y el segundo responde a dicho evento ejecutando su servicio. Usando este modelo de interacciones se aplica una transformación (aunque en realidad es un refinamiento) al modelo de la Figura 5.18(b), empleando el lenguaje QVT-operacional, para llegar a obtener el modelo final de componentes-y-conectores mostrado en la Figura 5.19 (d).

La especificación del modelo de *componentes-y-conectores* refinado contiene ahora el tipo de conectores y los roles con que ellos vinculan a los componentes. Cada conector

establece sus enlaces mediante sus roles con los puertos de entrada y salida de los componentes, como se muestra en el diagrama arquitectónico de la Figura 5.19(d), en la parte inferior de él se especifica el *perfil* de los componentes y conectores.

En este proceso se puede adoptar también alguna especialización de las relaciones descritas anteriormente como *filtro-tubería*, *cliente-servidor*, etc., dependiendo de la estrategia arquitectónica que se desee seguir. En tal caso se lleva a cabo una especialización no sólo de los conectores sino de también de los componentes, de acuerdo a los tipos de dichos elementos descritos en el Capítulo 3.

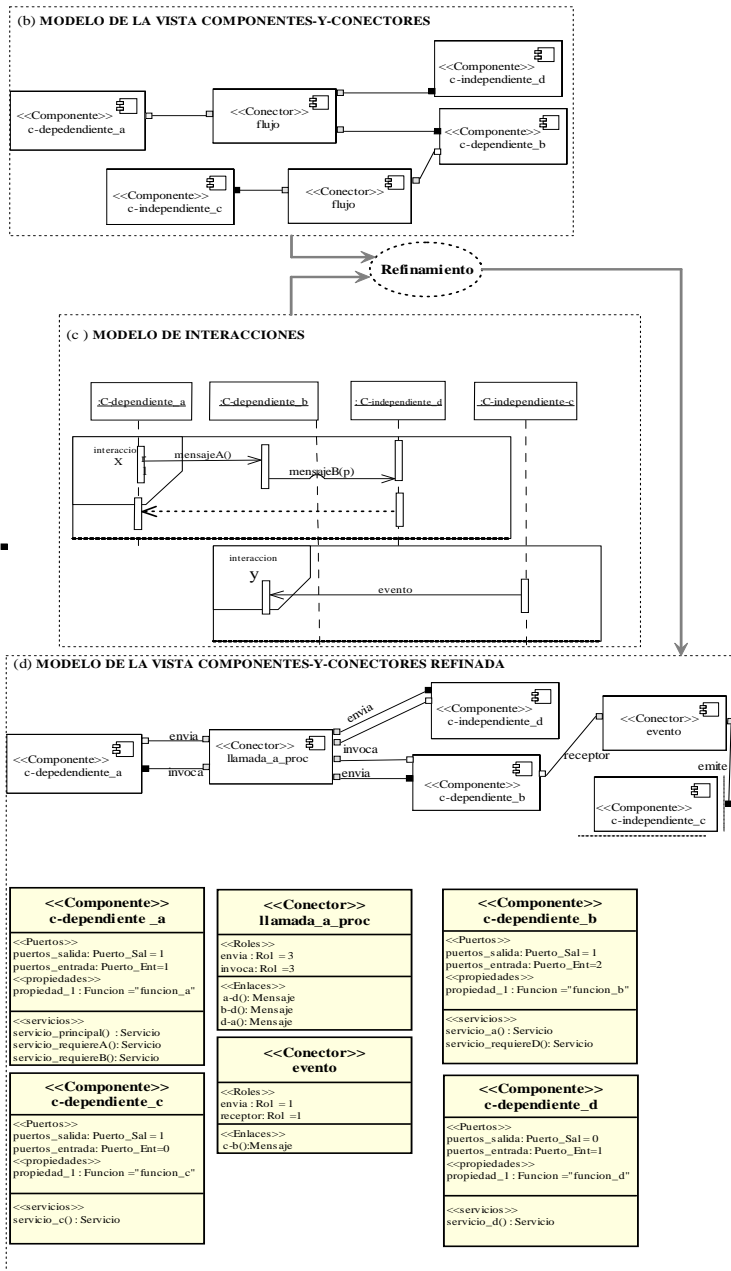


Figura 5.19 (continuación) Transformación de un modelo de vista modular a un modelo de vista de componentes-y-conectores con refinamiento

Capítulo 6

Aplicando la Transformación de Modelos en las Vistas Arquitectónicas

La arquitectura de software, puede ser aplicada a cualquier sistema que se desee desarrollar donde la planificación juega un papel decisivo, ya sea por lo complejo que pueda resultar su desarrollo o por las condiciones especiales que se presenten para su diseño (en seguridad, o criticidad, por ejemplo).

En los casos descritos en la literatura, se han aplicado a sistemas aeronáuticos y navales, que corresponden a sistemas críticos y de muy alta seguridad (Clements at al 2003). En este trabajo se aborda un caso de estudio que es aplicado a un contexto específico, con la intención de que al mismo tiempo que se ilustra la estrategia propuesta se enfrente una problemática real.

6.1 Caso de estudio

En esta sección se orienta el diseño de la arquitectura de software hacia un sistema que por la diversidad de elementos que intervienen en él, se hace necesario que se usen las dos vistas previamente analizadas. La vista modular será aplicada para hacer la separación de concerns como primera instancia, y posteriormente mediante el proceso de transformación de MDA se diseñará la otra vista y se mantendrá la consistencia entre ellas.

El caso de estudio consiste en el diseño de la arquitectura software para la construcción de una red de servicios

médicos para brindar atención a pacientes mediante los servicios públicos de salud en áreas remotas, también se comprende la gestión de consultas y la canalización de pacientes para su atención presencial, mediante la integración de los sistemas de las diversas instituciones de salud pública dentro del contexto mexicano, caso Oaxaca¹. Esto una variante de un sistema de telemedicina², como se describe a continuación.

En México los servicios públicos de salud están divididos en dos grandes grupos, uno de ellos corresponde a todas las instituciones que atienden a ciertos sectores laborales de la población (ellos pagan el servicio en forma directa), y el otro grupo está dedicado a la atención de la población que no está adscrita algún servicio médico (por ser trabajadores no asalariados, como campesinos o artesanos). Todas las instituciones de salud son parte del gobierno, y son coordinadas a través del sector salud a nivel de cada entidad federativa.

El sistema para el cual se diseñará la arquitectura pretende hacer un mejor uso de todos estos recursos compartiendo servicios (personal, clínicas, hospitales) para beneficiar a la población demandante. Esto tiene más sentido para el estado de Oaxaca, debido a la gran dispersión de su población (3,5 millones de habitantes distribuidos en más 3 mil asentamientos humanos, que comprenden a 570 municipios), lo cual dificulta el aprovechamiento de sus 63 hospitales, y sus más de 2 mil clínicas, ya que su cobertura

¹ Se ubica dentro de este contexto geográfico por tener posibilidad de acceder a información necesaria para plantear este caso, y porque constituye una demanda del sector salud de ese lugar.

² Definido por la OMS como: “El suministro de servicios de atención sanitaria en los que la distancia constituye un factor crítico, por profesionales que apelan a tecnologías de la información y de la comunicación con objeto de intercambiar datos para hacer diagnósticos, preconizar tratamientos y prevenir enfermedades y heridas, así como para la formación permanente de los profesionales de atención de salud y en actividades de investigación y de evaluación, con el fin de mejorar la salud de las personas y de las comunidades en que viven”

está limitada por las dificultades de acceso de muchas localidades.

La red de atención médica pretende integrar diversos servicios que hasta el momento funcionan aisladamente en algunas instituciones: gestión de consulta, asistencia médica remota (vía telefónica e internet), canalización de servicios para atención (especializada, auxiliares de diagnóstico, admisión hospitalaria), y suministro de medicamentos. Todo ello implica la coordinación de los sistemas de cada institución y de un trabajo colaborativo para proporcionar la asistencia médica remota compartiendo recursos. Esto último es dirigido básicamente para población rural¹ que está fuera de la cobertura de centros de atención de primer nivel.

Los requisitos agrupados de acuerdo a las funciones principales que se han señalado anteriormente, se muestran a continuación.

Asistencia médica remota. Este es un servicio muy útil para las localidades muy alejadas que no cuenten con médicos especialistas, porque se trata de que el médico o paramédico que esté a cargo del paciente en su localidad, se enlace a través del sistema con uno o más especialistas que les den las indicaciones posibles que procedan para el caso requerido.

En (Stevenson et al., 2008) se presenta un ejemplo de los recursos empleados en un caso de estudio sobre telemedicina, como se muestra en la Figura 6.1. En la foto del lado izquierdo está el médico especialista, que usa tres canales de video y uno de sonido para interactuar con la paciente que se muestra en la fotografía del lado derecho.

¹ Asentamientos de población que vive en el campo y que no llegan a formar más de 2.500 habitantes

Esto corresponde a lo que telemedicina se le llama consulta médica especializada (García B., 2003), lo cual se logra mediante un trabajo colaborativo en donde el médico presencial interactuará con uno o más médicos que brindarán la asistencia remota a través de Internet, y usando los medios tecnológicos a su alcance.

En el caso que nos ocupa, se pretende una comunicación de médico-especialista a medico-general (o paramédico) usando los recursos vía internet, pues lo que importa aquí es la guía del especialista auxiliado por audio y video.

Las funciones a realizar dentro de este requerimiento comienzan cuando el médico general al estar atendiendo a un paciente elabora el diagnóstico preliminar, después al determinar que requiere la asistencia de un especialista para precisar el diagnóstico, establece el contacto con los médicos adecuados (oftalmólogo, ginecólogo, otorrino, etc.) vía chat (o webchat) de acceso restringido, entonces se inicia una sesión entre médico-general y médico-especialista, el médico-general interroga al paciente empleando multimedia para el envío de información cuando esto sea posible (dependiendo del padecimiento).



Figura 6.1 Trabajo colaborativo para telemedicina, tomado de (Stevenson et al., 2008)

El especialista va guiando al médico sobre la forma de llevar a cabo el diagnóstico, éste usa medios como cámaras genéricas para transmitir el aspecto externo del paciente o internas no invasivas como las usadas por los otorrinolaringólogos.

En caso de requerir algún auxiliar de diagnóstico (pruebas de laboratorio, rayos X, etc.) se generará la orden respectiva mandándose a la clínica que pueda realizar la prueba que esté en una zona lo más cercana a donde se localiza el paciente. En cualquier caso (con auxiliar de diagnóstico y sin él) se da la prescripción de los medicamentos que deben administrarse al paciente. Además se crea un programa de seguimiento del paciente, mandándosele al especialista en caso de ser requerido o bien al médico general dependiendo de la afección y la evolución de la misma.

Gestión de consultas. Este servicio tiene dos fases, una es la concertación de una cita y la segunda es el seguimiento de ella. La primera se refiere a que cualquier paciente (previamente registrado), podrá solicitar vía internet o vía telefónica una cita médica (en este caso un operador será el enlace entre el paciente y el sistema), el sistema se encargará de ubicar a qué institución de salud está adscrito el solicitante, y de otorgarle la cita en función de la disponibilidad dentro de su unidad de adscripción, si no hay disponibilidad en su centro de adscripción se buscará en otra institución en un centro cercano al domicilio del solicitante.

Una vez registrada una cita ésta podrá ser cambiada o cancelada por el propio paciente. Algunas citas podrán realizarse directamente con el especialista, otras tendrán que pasar primero con un médico general ya sea porque es un requerimiento médico o porque no exista el especialista requerido cerca de su zona.

La otra parte de la gestión de la consulta consiste en su seguimiento, esto se da en el momento en que el paciente acude a la cita, entonces el médico accederá al historial del paciente para actualizarlo y generará las indicaciones para su tratamiento, parte de una indicación puede ser la canalización del paciente a otros servicios.

La canalización de servicios. Consiste en canalizar a un paciente de una unidad médica básica a un centro de atención con equipamiento y/o especialización donde pueda ser atendido (centro de diagnóstico, hospital, clínica especializada). En este caso el sistema será capaz de encontrar el centro de atención más cercano que cumpla con los requerimientos y que cuente con la disponibilidad, haciendo la reserva y generando la logística para el traslado del paciente para el caso de las comunidades alejadas. En los casos donde se requiera el traslado se hará por ambulancia por lo cual se deberá hacer una programación de estas de acuerdo a los itinerarios considerado que una ambulancia puede no solo lleva al paciente a una unidad médica sino también lo lleva de regreso a su lugar de residencia.

Suministro de medicamentos y material de curación. Este requisito consiste en que el sistema encuentre el lugar de dónde se podrá abastecer el fármaco y material solicitado, en función de su disponibilidad y cercanía de lugares de acopio, generando la orden respectiva. Esto es producido como una consecuencia de un servicio de atención médica remota. A semejanza del sistema de canalización de servicios, aquí también es requerido que el sistema genere un ruteo para el abastecimiento de los medicamentos en el tiempo solicitado.

El sistema a diseñar para los requisitos expuestos tendrá que ser operado por diferentes instituciones y tipos de usuarios. Será un sistema distribuido y adaptable, en el sentido que

cada institución además de compartir parte de su información, deberá adecuar el sistema a sus propias plataformas. Por otro lado, para la atención médica remota, el sistema se encargará de establecer el contacto de qué médicos especialistas están disponibles para realizar una asistencia. Los médicos (y operadores no médicos) también accederán al sistema para canalizar a los pacientes a los centros de atención donde les realicen servicios auxiliares de diagnóstico o a instancias hospitalarias, cuando los casos lo necesiten. Por su parte los pacientes podrán acceder al sistema para el registro de citas médicas y de laboratorio, y gestión de las mismas (cambio de fechas y bajas) y seguimiento de parte de su historial médico (como por ejemplo tratamientos recibidos, médicos que los han atendido, medicamentos suministrados).

De los requisitos expuestos se derivan los *concerns* arquitectónicos, es decir aquellos que sirven para el diseño de las vistas arquitectónicas para un sistema. Estos concerns son expresados por medio de un esquema llamado ciclo de la arquitectura, propuesto por (Clemenst et al, 2003, pags. 9-14), que ilustra los cuatro factores que influyen en la arquitectura: actores del sistema, organización del desarrollo, ambiente técnico y experiencia de los arquitectos, como se muestra en la Figura 6.1.

El ciclo arquitectónico se inicia por los actores que intervienen tanto para la especificación de los concerns arquitectónicos como para la definición de los requisitos funcionales y de calidad en una forma organizada. De esta manera se produce la información sobre los requisitos de calidad como los que se muestran en la Figura 6.1, los cuales a su vez sirven para ir guiando a los arquitectos responsables del modelado de la arquitectura. Cada uno de los requisitos de calidad se va usando para diseñar la arquitectura en forma iterativa, en cada iteración se va construyendo un

modelo parcial que sirve para ir diseñando al sistema (en este caso la red de servicios médicos públicos), cada modelo parcial producido se usa para refinar los factores, el ciclo continúa hasta que se terminen de incorporar los requisitos de calidad, completando con esto la arquitectura. Cada uno de estos factores es descrito a continuación.

Los actores comprendidos en la operación del sistema planteado son los médicos especialistas y generales , los gestores de consulta médica, los proveedores de servicios y los pacientes. Los médicos especialistas brindaran asistencia remota, guiando a los médicos generales e interpretando la información sobre el estado del paciente que le será transmitida, los médicos generales que además de interactuar con los especialistas se encargaran de operar el sistema para seguir las indicaciones para tratar al paciente y trasmitir información que ayude a su diagnostico y tratamiento. Además los médicos (generales y especialistas) deberán operar el sistema respecto a la canalización de los pacientes a los servicios que requieran, solicitar fármacos y material médico.

Los gestores de consultas médicas por su parte asisten vía telefónica la concertación de citas y otros servicios requeridos por los pacientes, ellos usarán al sistema vía internet para esta concertación y tendrán el control sobre el sistema de atención telefónica (que operará mediante protocolo de internet(IP)). Los proveedores de los servicios manipularan al sistema en lo concerniente a la gestión de atención directa general y especializada, auxiliares de diagnostico, hospitalización, tratamientos, para el control de los servicios prestados. Por último, los pacientes harán la reserva de una cita, y la gestionaran (consultarla cancelarla, o cambiarla), además podrán tener acceso a la base de datos de su historial médico en ciertos aspectos(por ejemplo seguimientos de un tratamiento).

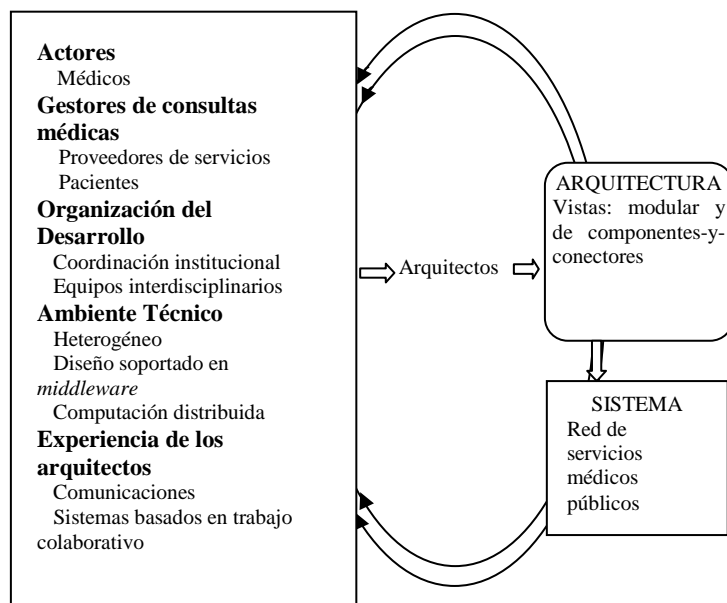


Figura 6.2 Ciclo de la arquitectura aplicado al sistema de la red de servicios médicos públicos

La organización del desarrollo, indica la manera en que se llevará a cabo la organización del equipo de trabajo (personal). En este caso se indica que se debe llevar a cabo una coordinación con las diferentes instituciones de salud tanto para el diseño del sistema como para su operación, y también debe intervenir personal interdisciplinario de las áreas de computo, informática, comunicaciones y especialistas médicos.

El ambiente técnico condiciona de cierta manera los recursos tecnológicos que se usaran, en este caso el empleo de varias plataformas indica que es un ambiente heterogéneo, que tendrá que basarse en un desarrollo con *middleware* para la adaptación del sistema a las diferentes

plataformas, y en un ambiente distribuido para la compartición de información y procesos.

De lo anterior se deriva que los arquitectos deberán contar con experiencia en comunicaciones (para la interconexión) y en sistemas basados en trabajo colaborativo que constituye la base principal de los servicios para este caso de estudio.

De los cuatro factores que intervienen en el ciclo arquitectónico, los actores y la organización del desarrollo son los que más influyen en la arquitectura del sistema. Los arquitectos interactúan con ellos para que en función de los requisitos funcionales deriven los requisitos de calidad que debe cumplir el sistema. Los requisitos de calidad que se han detectado en este caso son usabilidad, seguridad, escalabilidad e interoperabilidad, los cuales se irán detallando al momento de crear la arquitectura de software, esto se hace tomando en cuenta los atributos de calidad descritos en el Capítulo 2 de esta tesis.

La usabilidad es necesaria porque el sistema cuenta con diferentes tipos de usuarios que tienen distintos perfiles profesionales, lo cual incide en el diseño de su interfaz para dar facilidades en su manejo, soporte al usuario y en el soporte al propio sistema.

La seguridad es requerida porque el sistema compartirá información con diferentes niveles de acceso, por ejemplo con respecto a la información sobre historial médico los pacientes sólo podrán acceder a parte de su historial para su consulta, mientras que los médicos podrán tener acceso a todos los detalles realizando las actualizaciones pertinentes, y los operadores auxiliares de diagnóstico solo podrán incorporar resultados. La seguridad también concierne al cuidado de ataques a las bases de datos y a prevenir las fallas en la red de comunicación.

La escalabilidad es importante en el caso del servicio de asistencia remota, porque primeramente este servicio iniciará como una guía de diagnóstico y atención pero al ir creciendo la demanda y equipamiento médico, esta guía podrá ser mejorada con sistemas expertos o especializarse con el uso de equipo médico de diagnóstico que puede transmitir información especializada. En algunas circunstancias tendrán que trabajar varios especialistas en forma colaborativa para tratar casos complicados de pacientes. De tal manera que el sistema llegará a ser escalado en diferentes aspectos.

En el caso de la interoperabilidad, esta se hace necesaria porque los diferentes subsistemas que se desarrollarán tendrán que ser adecuados a las plataformas que maneja cada institución de salud, lo mismo se hará con la forma de manejar las bases de datos puesto que se tendrá que compartir información almacenada en sistemas gestores de bases de datos diferentes, operados con diferentes sistemas.

6.2 Iniciando la arquitectura con el modelo de la vista modular

Diseñar el modelo de la vista modular consiste en identificar y especificar los módulos en que el sistema puede ser dividido formando las estructuras propias de esta vista. Para esto, se usa un proceso de descomposición mediante un método basado en los requisitos funcionales y de calidad, que permiten dirigir la descomposición y formar los módulos en este caso del sistema de red de servicios médicos. Esta propuesta es una variante del método dirigido por atributos de (Bass et al., 2001). Las tareas que comprende son mostradas en la Figura 6.3. las cuales se describen a continuación con un ejemplo y luego se aplica al caso de estudio.

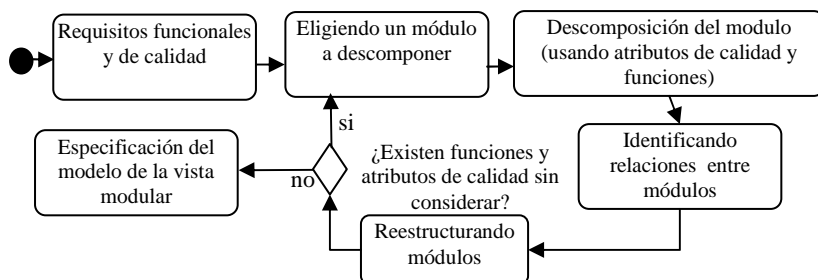


Figura 6.3 Método para la especificación del modelo de la vista

El estado inicial del método comienza por proporcionar los requisitos funcionales y de calidad del sistema en cuestión. Los requisitos funcionales son determinados por la ingeniería de requisitos, en el caso de estudio aquí analizado estos han sido presentados en una forma descriptiva (sección previa), ellos constituyen la entrada del sistema para la descomposición de los módulos, el punto de partida lo constituye el sistema de red de servicios médicos, él será en sí el primer módulo, el cual se procederá a descomponer. Para la descomposición de un modulo se toma un requisito de calidad y en función de él se analizan los requisitos funcionales correspondientes a ese modulo que pueden cumplir con ese requisito de calidad.

En la Figura 6.4 (a) se muestra en forma esquemática cómo un módulo X incluye a tres requisitos funcionales que llegarán a satisfacer, éstos serán promovidos en nuevos módulos para formar la arquitectura. La cuestión aquí es saber cómo se determina la manera en que los requisitos funcionales llegan a cumplir con un requisito de calidad. Como fue analizado en el Capítulo II, Sección 2.2, sólo existen algunos criterios basados en la experiencia de los arquitectos, de ahí que este factor sea clave en el modelado de la arquitectura. En el esquema mostrado se incorpora el requisito de calidad capacidad de modificación, que induce a que las relaciones

que se establezcan entre los módulos conlleven a una débil dependencia para evitar que un cambio repercuta fuertemente en otros.

La relación de composición es la primer relación que se aplica al promover los requisitos funcionales contenidos en un modulo en otros módulos como el presentado en la Figura 6.4 (b), a partir de ahí las interacciones derivadas de los requisitos funcionales propician relaciones de uso y especialización entre los módulos derivados, donde los requisitos de calidad inducen a estas relaciones. Para ilustrar cómo influye el requisito de calidad capacidad de modificación en las relaciones de los módulos del esquema mostrado, se considera que se ha detectado una relación de especialización entre los módulos J y N (indicada en la Figura con $--\rightarrow$), sin embargo esta nueva relación hace que se cree una doble relación de composición entre los modulo N y X , la ya existente y la que se forma por transitividad entre el modulo $N \rightarrow J \rightarrow X$. Como lo que impone el requisito de calidad es simplificar las dependencias entre los módulos entonces se reestructuran la relaciones como se indica en la Figura 6.4(c), eliminando la relación de composición directa entre X y N, de esta manera se simplificará alguna operación de modificación que tenga que hacerse en estos módulos relacionados.

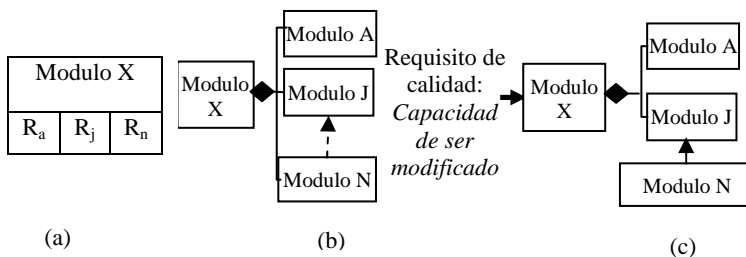


Figura 6.4 (a) Requisitos funcionales de un módulo (b) Relación de composición. (c) Reestructurando los módulos con un requisito de calidad.

La forma de relacionar los módulos resultantes de una descomposición tiene que ver con el criterio aplicado para tal fin. En algunos trabajos sobre descomposición se han empleado criterios que buscan descomponer a un sistema basado en la ontología, como el propuesto por (Paulson y Wand, 1992) que usa heurísticas basadas en los estados de variables para crear relaciones de descomposición de un sistema.

Otros trabajos han combinado diferentes criterios para establecer relaciones, como el de (Chang et al., 2001) que aplica una descomposición funcional con clases, combinando la descomposición estructural con la orientación a objetos, desagregando al sistema mediante relaciones jerárquicas de clases. También se han aplicado técnicas de agrupación para formar módulos, como en la propuesta de (Mitchell y Mancoridis, 2006) que usa algoritmos para agrupar los módulos e indicadores para medir la calidad de la relación de descomposición de los módulos.

En nuestro caso para crear las relaciones de los módulos se usa como principal fuente la especificación de requisitos funcionales, y se aplican las tácticas de requisito para afinar las relaciones. De esta manera la arquitectura inicial se forma con los cuatro grupos de requisitos descritos previamente: asistencia médica remota, gestión de consultas, canalización de servicios, suministro de medicamentos y material de curación, cada grupo representa en sí un módulo como se muestra en la Figura 6.5. Cada uno de ellos tiene una función y al menos una responsabilidad.

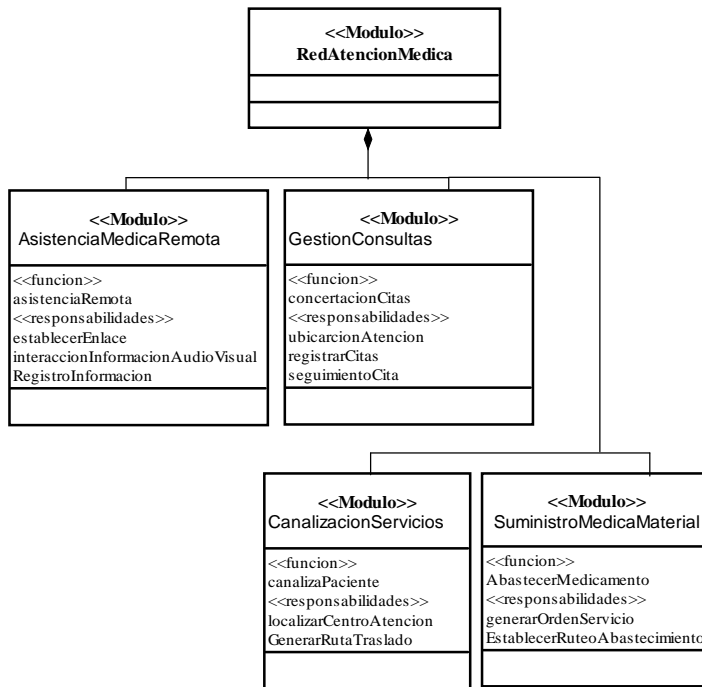


Figura 6.5 Primer arquitectura de la vista modular del sistema red-atención-médica

Para iniciar con la descomposición de esta primer arquitectura se elige al módulo de *AsistenciaMedicaRemota* (la elección por cuál iniciar no influye en el resultado final), en este caso cada una de las responsabilidades se llega e convertir en un nuevo modulo más, con una función principal y sus responsabilidades. La función del cada nuevo módulo es derivada de su responsabilidad y las nuevas responsabilidades se generan de la profundización de sus requisitos, esto es mostrado en la Figura 6.6.

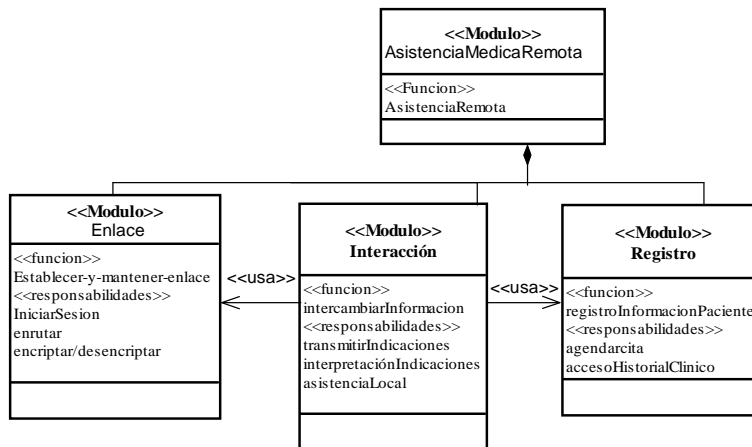


Figura 6.6. Descomposición del módulo *AsistenciaMedicaRemota* y las relaciones identificadas entre los módulos derivados

Ahora se procede a identificar las relaciones que se presentan entre los nuevos módulos. En primer lugar se detecta en forma intuitiva que existen relaciones entre el módulo de Interacción con los módulos Enlace y Registro. La relación entre Interacción-Enlace se presenta porque el primer modulo requiere que se establezca y mantenga un enlace entre un médico especialista con otro médico para poder brindar la asistencia médica, así el primer módulo usa al segundo. De manera semejante el módulo interacción usa al módulo registro con el propósito de verificar la situación en que este solicitó su atención (para ubicar el tipo de atención requerida), y también para generar otros registros como consecuencia de la propia atención prestada al paciente. Estas relaciones de uso son mostradas en la Figura 6.7 antes referida.

La restructuración de los módulos en este punto no es necesaria porque son las primeras relaciones detectadas, por tanto se continúa con la descomposición del siguiente

módulo presentado en la arquitectura inicial (Figura 6.5) el cual es GestionConsultas , cada una de sus responsabilidades llega a formar en si un módulo independiente como se muestra en la Figura 6.7. Ahí se aprecia también las relaciones detectadas entre ellos, dos relaciones de uso y una de generalización.

Una relación de uso se produce porque el módulo RegistrarCitas usa al de UbicaciónAtención para encontrar la unidad de adscripción donde se le brindará atención medica al paciente o el tratamiento prescrito, y la otra relación de uso se da entre el primer modulo con el de SeguimientoCitas , ya que al registrar una nueva cita también se tienen que revisar otras citas que haya realizado un paciente para establecer vínculos entre ellas.

Por otra parte el módulo UbicacionAtencion es una generalización de los módulos Atencion-medica y Atención-tratamiento, porque incorpora la función común de asignar la unidad donde será remitido el paciente ya sea para atención médica o para algún tratamiento.

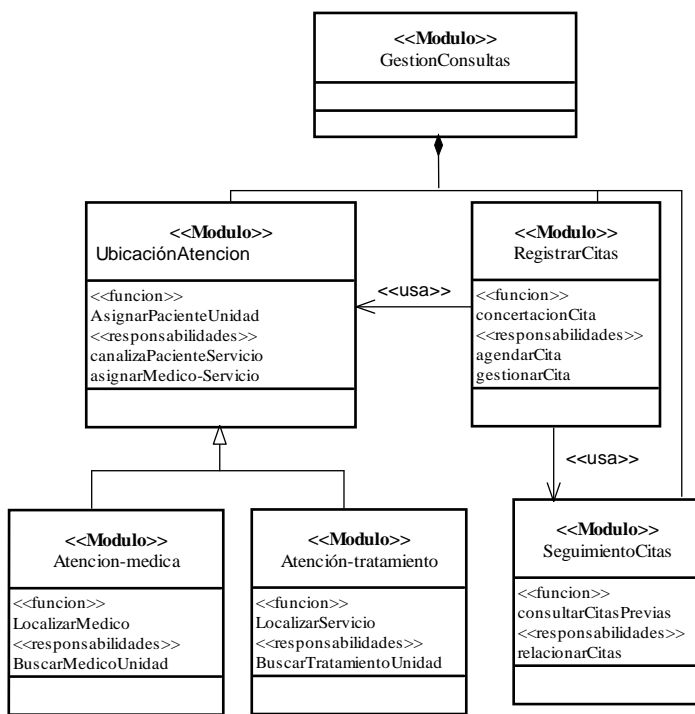


Figura 6.7 Descomposición del módulo *GestionConsultas* con sus relaciones identificadas

Los dos últimos módulos de la arquitectura inicial llamados *CanalizacionServicios* y *SuministroMedicoMaterial* no necesitan ser descompuestos en otros puesto que sus responsabilidades son puntuales, es decir muy específicas, pero como incluyen una responsabilidad que implica transportar tanto a pacientes o la distribución de medicamentos y material de laboratorio, se hace necesario un módulo adicional que tenga como función la generación de rutas para esos fines, donde sus responsabilidades sean proporcionar la ruta optima que combine los recursos que se disponen con los tiempos asignados, u otras rutas que sólo tenga como prioridad algún criterio como el tiempo o la cantidad de medicamentos o material a distribuir. Este

nuevo módulo así como su relación con los primeros citados se muestra en la Figura 6.8.

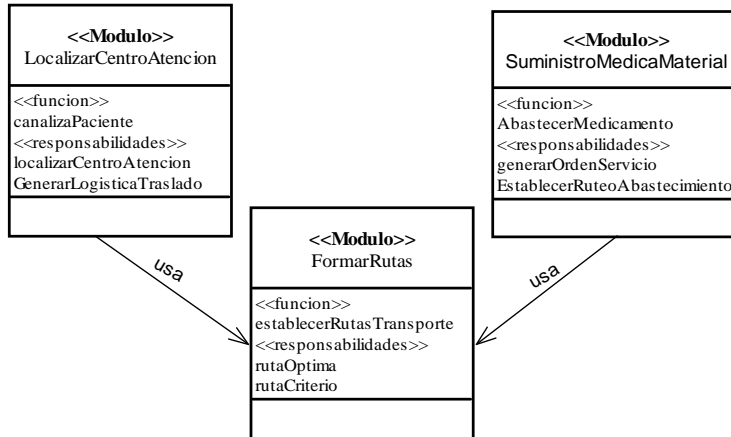


Figura 6.8 Relaciones de uso entre los módulos LocalizarCentroAtencion y SuministroMedicaMaterial con FormarRutas

Las relaciones entre los módulos obtenidos en el proceso de descomposición hace que se llegue a la arquitectura final de esta vista modular como se muestra en la Figura 6.9, en donde se incluyen las relaciones de uso que se dan entre los módulos obtenidos al final de este proceso. Una de ellas es entre el módulo Interaccion con el de GestionDeConsultas, y otra se establece entre el módulo CanalizacionServicios con el de UbicacionAtencion. Además hay que notar que los módulos que han sido descompuestos en otros no poseen responsabilidades propias, ya que sus responsabilidades son asumidas en los módulos que los constituyen, de tal manera que este tipo de módulos funcionan como sub-sistemas, dependientes directamente del sistema principal.

6.3 Obteniendo el modelo arquitectónico de la vista de componentes-conectores

Como se expresó en el capítulo V, el modelo de la vista modular es usado como fuente para la obtención de la primer versión del modelo de la vista de componentes y conectores. Por lo tanto, lo primero que se hace es representar al modelo de la vista modular como una instancia de su meta-modelo creado en el entorno de ECLIPSE para que le sean aplicadas las relaciones entre sus meta-modelos mediante la transformación con el lenguaje QVT-operacional, generando el modelo de la vista componente-conector correspondiente.

Las relaciones que son aplicadas a este caso son de varios tipos, cada una de ellas va transformando a los elementos módulos en componentes y a sus relaciones en conectores. Aunque el proceso es realizado en forma automática gracias a la implementación hecha en el entorno de ECLIPSE, es importante destacar cómo se lleva cabo cada una las diversas transformaciones en el caso propuesto.

Las transformaciones parten de lo general a lo específico, es decir desde los elementos de tipo subsistemas que contienen a otros, hasta llegar a los elementos específicos contenidos en elementos unitarios. De esta manera en el caso presentado se inicia por la conversión de los subsistemas AsistenciaMedicaRemota y GestionConsultas a componentes compuestos (aplicando la relación subsistemaAcompocom descrita en la sección 5.4 del Capítulo V), donde los elementos de estos últimos son componentes obtenidos de la transformación de los módulos.

En esta primera fase de transformación las relaciones entre los componentes “contenidos” aún no se toman en cuenta, ni la comunicación entre los componentes internos con su contenedor, ya que esto será derivado del modelo de interacción. Pero sí se generan a los componentes respectivos de cada módulo mediante su conversión creándose sus puertos básicos (entrada y salida), donde su identidad se deriva de los módulos. Además se especifica su tipo de acuerdo a los servicios que cada componente proporciona, por ejemplo el componente de *C-Interacción* es de tipo *DualidadServicios*, porque este componente es usado tanto como *servidor* y como *cliente*. En cambio los componentes *C-Enlace*, *C-ubicacionAtencion* y *C-SeguimientoCitas*, son de tipo *Servidor*, ya que sólo brindan los servicios especificados. El componente *C-RegistrarCitas* es de tipo *UnidadConcurrente* por tener servicios distribuidos, y por último el componente *C-Registro* sirve como acceso a una base de datos, de ahí que sea de tipo *DatosCompartidos*. Estos componentes son mostrados en el modelo de la vista de componentes-conectores generado, ilustrado en la Figura 6.10 (b).

Así también, como parte de esta transformación las funciones de los módulos son convertidas en atributos y sus responsabilidades en servicios, los cuales están encapsulados en los componentes, por eso sólo son especificados a nivel de caja blanca en los estereotipos <<Componentes>>, como se aprecia en la Figura 6.10 (c).

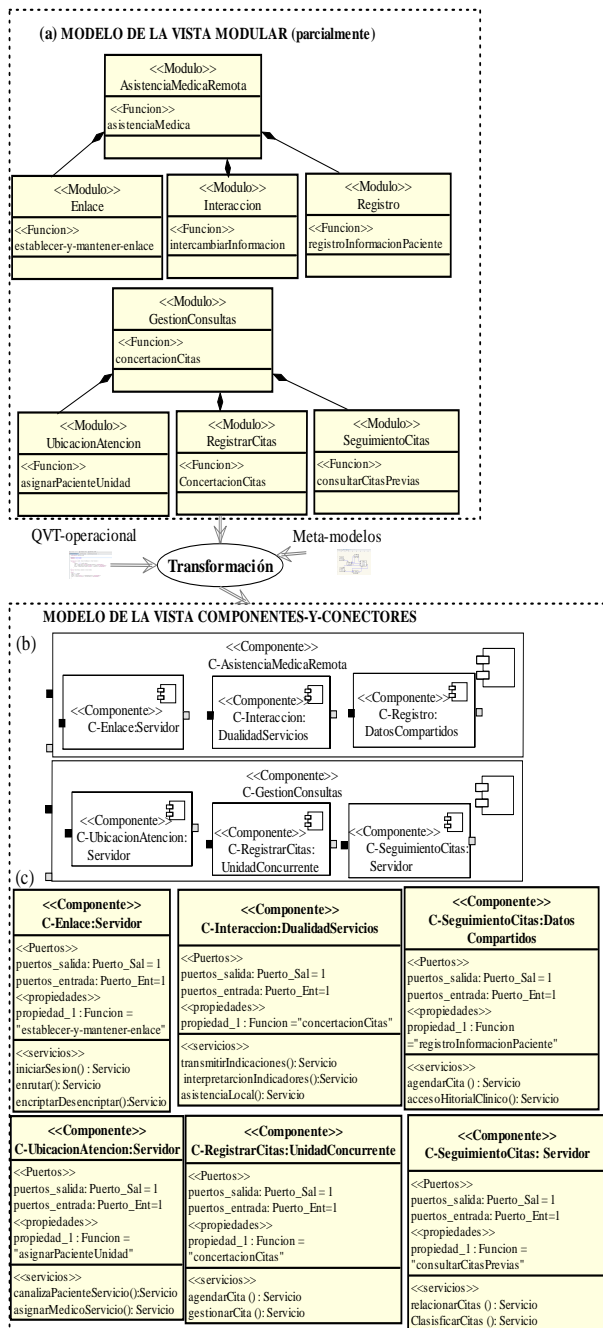


Figura 6.10 Primera fase de transformación entre el modelo del vista modular y el de la vista componentes-y-conectores

La siguiente fase en este proceso de transformación es convertir las relaciones que existen entre los módulos incluidos en una composición en conectores del modelo componentes-y-conectores. Existen dos tipos de relaciones a transformar en el modelo modular en cuestión, la de *usa* que está presente en los dos módulos que forman composiciones *AsistenciaMedicaRemota* y *GestionConsultas*, y la relación de generalización que se tiene sólo en la segunda composición.

Transformación de las relaciones *usa*

Cada una de las relaciones *usa* que enlaza a los módulos que están en esta circunstancia es transformada aplicando la relación de correspondencia *usaConector* (sección 5.4 , Capítulo V), creándose por cada una de estas relaciones un conector que vincula a los componentes respectivos obtenidos en la primera fase de la transformación, posibilitando su interacción mediante sus roles.

De esta manera se aplica la relación de correspondencia *usaConector* a cada relación *usa* existente entre los módulos

interacción $\xrightarrow{\text{usa}}$ *enlace*, *interacción* $\xrightarrow{\text{usa}}$ *registro*,
registrarCitas $\xrightarrow{\text{usa}}$ *ubicarAtención*, y entre
registrarCitas $\xrightarrow{\text{usa}}$ *SeguimientoCitas* (Figura 6.10(a)).

Los conectores que se obtienen en cada transformación pueden ser especificados de manera completa o sólo en forma parcial, dependiendo del tipo de componentes que vinculan y de la información disponible que se tenga hasta ese punto sobre su interacción. Un conector es especificado en forma completa cuando se determinan todos los vínculos entre sus componentes, los enlaces entre sus roles y los procesos que este tiene que realizar (cuando existan).

Con el propósito de ilustrar cómo se realiza la transformación de una relación *usa* y la especificación del conector generado en ella, en la Figura 6.11 se presenta a nivel de modelo la transformación de la relación *interacción* $\xrightarrow{\text{usa}}$ *enlace*. La identidad del conector que se obtiene en esta transformación se forma por la composición de los nombres de los componentes vinculados (*interacción-Enlace*), y su tipo se define en función del tipo de los componentes. En este caso el tipo del conector es definido como *Llamada-a-procedimiento*, porque es el adecuado para vincular componentes de tipo *cliente-servidor* (Tabla 3.4, Capítulo III), siendo que el componente *C-Enlace* es de tipo *servidor*, y *C-iteración* es de tipo *cliente* (el tipo *DualidadServicios* lo incluye).

De esta manera los roles del conector se definen en función de lo que el cliente solicita (o lo que el servidor provee), esto se pone de manifiesto en la designación de sus nombres. Así el rol *solicitaEnlace* indica que el componente *C-iteracion* (el cliente) es quien realiza esta solicitud a través de su puerto *ps*, y mediante el rol *invocaEnlace* el conector establece el vínculo con el componente *C-enlace* por el puerto *pe*.

De manera reciproca este último componente se vincula con el conector a través del rol *enlace*, lo cual es transmitido al otro componente por el rol *estableceEnlace*. Estos vínculos entre los roles con los puertos de los componentes son especificados en el prototipo del conector mostrado en la Figura 6.11(c), donde también se definen sus roles y los enlaces internos que existen entre ellos. En forma semejante es llevada a cabo la transformación de lo otras relaciones *usa* antes referidas.

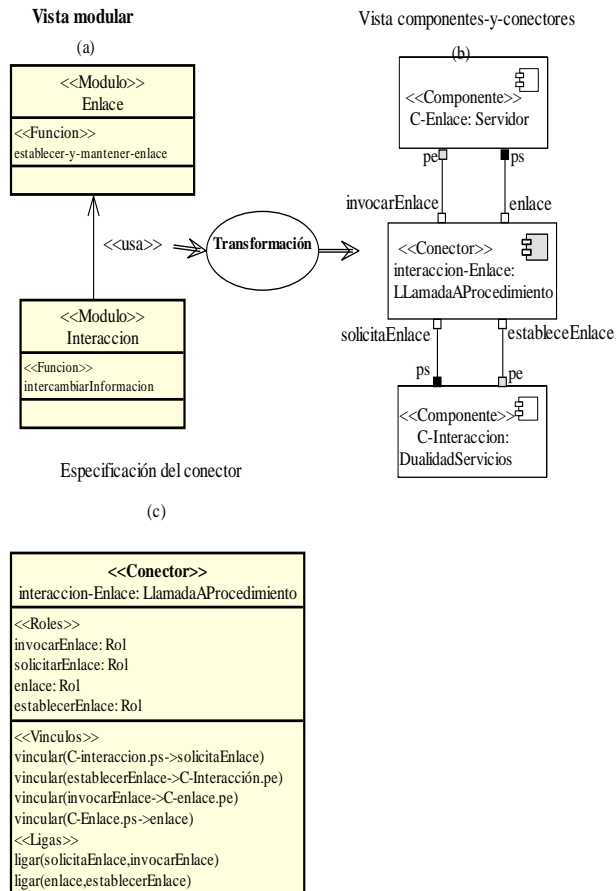


Figura 6.11 Transformando una relación usa del modelo de la vista modular al modelo de la vista componentes-y-conectores

Transformación de la relación de generalización

Para el caso de la relación de generalización que se presenta en el módulo UbicacionAtencion:

$$UbicacionAtencion \leftarrow \frac{generaliza}{AtencionTratamiento} AtencionMedica$$

Esta relación es transformada aplicando la regla de correspondencia generalizaAConector (Sección 5.4 Capítulo V) , generándose los componentes correspondientes llamados C-AtenciónMedica y C-AtencionTratamiento, y un conector que los vincula denominado Atencion, lo cual es mostrado en la Figura 6.12. Los componentes generados son de tipo DatosCompartidos ya que constituyen accesos a bases de datos, ellos son requeridos por el componente C-UbicaAtencion que es de tipo servidor . La comunicación entre todos estos componentes se establece a través del conector Atención, que al ser de tipo Distribuidor (de acuerdo a la regla de correspondencia aplicada) se encarga de indicar hacia cuál de los componentes deben ser dirigidos los servicios solicitados.

Los vínculos que se establecen entre los roles del conector y los puertos de los componentes denotan su interacción, así el vínculo C-Ubicación.ps→solicitaAtencion indica que el servidor requiere acceder a datos sobre atención médica y/o tratamiento, por lo que el conector se encargará de coordinar el servicio hacia los otros dos componentes los cuales se encargaran de ubicar la atención solicitada por el componente C-UbicaciónAtencion. Esta característica de distribución del conector requiere de un proceso interno que coordine al componente que solicita los servicios con los componentes que los proveen. La especificación de los vínculos, enlaces y el proceso de coordinación son definidos dentro del conector como se muestra en la Figura 6.12 antes referida.

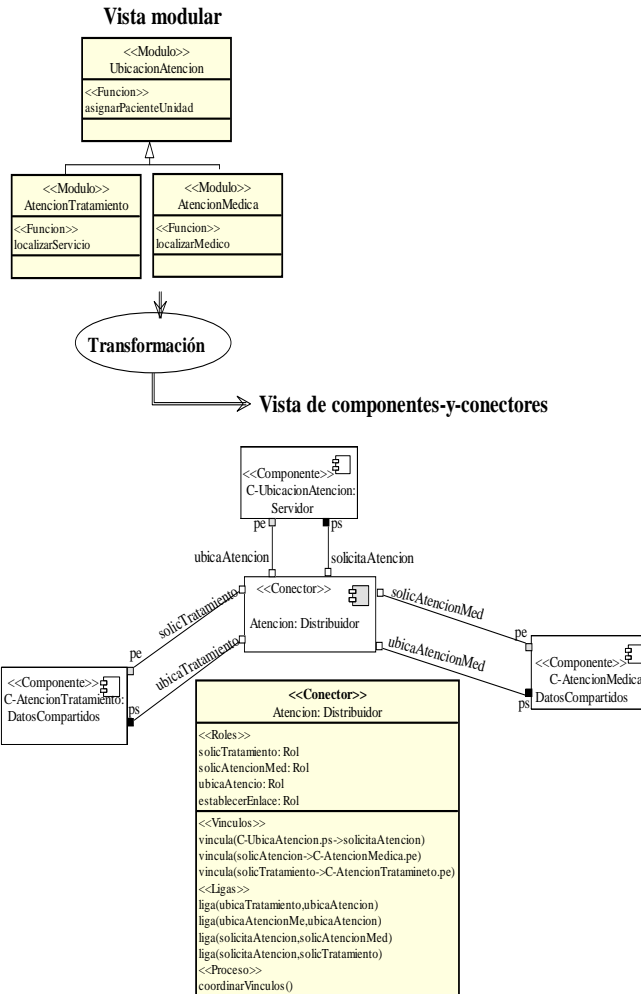


Figura 6.12 Transformación de una relación generalización del modelo de la vista modular al modelo de la vista componentes-y-conectores

Una vez que se han aplicado las transformaciones correspondientes a los elementos internos de los componentes compuestos, se procede con la aplicación de la regla de transformación usaConector a los demás relaciones usa del modelo modular que no fueron descompuestos, llegándose a obtener el modelo de la vista componente-y-

conectores de esta segunda fase de transformación, como se muestra en la Figura 6.13, nótese que aun hay componentes sin vincular pues aún hace falta un refinamiento.

**MODELO DE LA VISTA COMPONENTES-Y-CONECTORES
(despues de la segunda fase de transformación)**

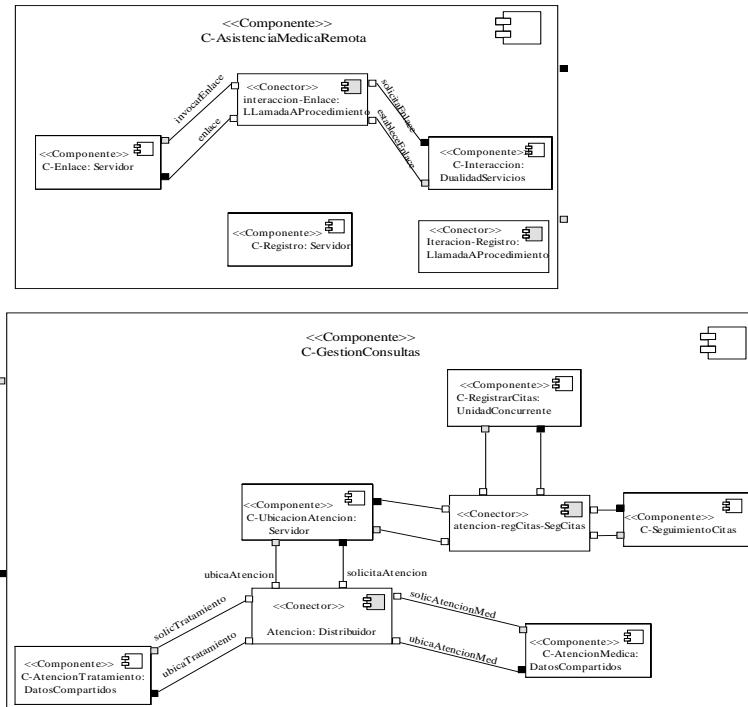


Figura 6.13. Modelo de la vista de componentes y conectores resultante de la transformación

6.4 Refinamiento de relaciones

Algunas de las interacciones entre los componentes ya se han establecido a través de sus conectores en la transformación de relaciones específicas como en la de generalización, pero otras interacciones aún faltan ser afinadas para determinar el tipo de conectores y sus roles, además se tiene que encontrar cómo estos últimos se vinculan con los puertos de los componentes con que se comunican. Este refinamiento se consigue mediante la especificación del modelo de interacciones de los componentes obtenidos en la fase anterior, cuyo modelo se construye usando los requisitos presentados en el planteamiento del caso, los requisitos de calidad y los patrones arquitectónicos aplicables a los componentes.

Se toma como referencia uno de los requisitos de calidad que debe cumplir el caso de estudio -la interoperabilidad del sistema-, debido a que algunos de los componentes del sistema red de servicios médicos tienen que interactuar con otros sistemas (de cada una de las instituciones médicas que forman la red). Los componentes involucrados con este requisito de calidad son C-Registro y C-GestionConsultas, ya que los servicios que ellos proporcionan son básicamente accesos a información la cual es manejada por diferentes tipos de sistemas, por lo que se hace necesario aplicar un estilo arquitectónico como el llamado de varios-niveles (conocido técnicamente como n-tier) para garantizar que la operación de los servicios sea independiente de los sistemas que lo manejen. Con este propósito se incorpora a un componente tipo interfaz, cuya función será brindar accesibilidad a los usuarios (formando un nivel más), dejando la tarea de manipulación de datos a cada componente con se corresponde (en un nivel distinto).

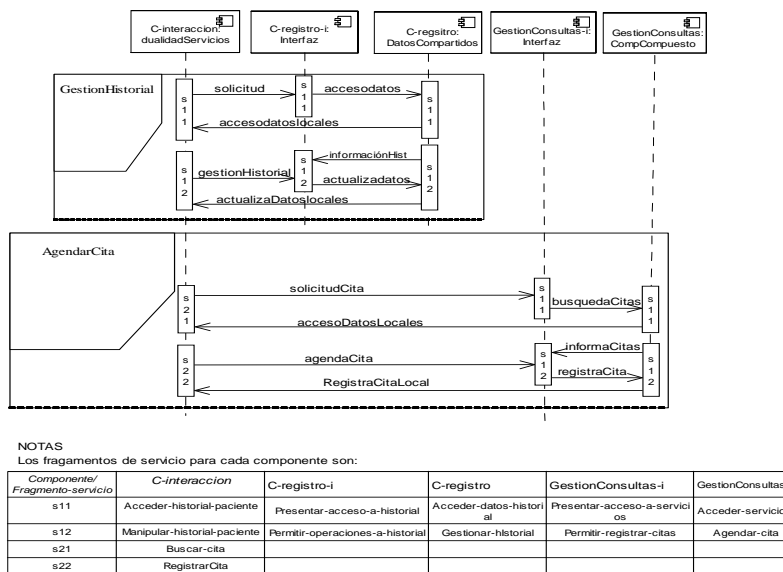


Figura 6.14 Modelo de interacciones entre componentes

De esta manera se incorpora un componente de tipo interfaz para el componente C-Registro y otro para el C-Gestion, los cuales servirán para acceder a sus respectivos servicios. Al componente que le corresponde acceder a los servicios de dichos componentes es C-interaccion.

Las dos interacciones que se forman entre estos componentes se muestran en la Figura 6.14, una es llamada GestionHistorial y la otra AgendarCita, cuyo nombre de cada interacción por si mismo indica lo que se realiza en cada una de ellas. Cada interacción está formada por los componentes participantes, por los servicios utilizados de estos componentes, y por los mensajes que se envían entre uno y otro servicio en un cierto orden indicado por la línea del tiempo (-----).

Los mensajes indican una invocación o respuesta de los servicios, y acceden sólo a una parte del servicio haciendo que éstos sean divididos “*fragmentados*”. Estos fragmentos de servicios son etiquetados para diferenciar sus acciones, los nombres correspondientes a cada etiqueta están descritos en el cuadro inferior de la Figura 6.14 referida, en el primer renglón se muestra a que componente pertenece, y en los renglones siguientes sus etiquetas describen acciones específicas de su servicio.

En cada una de las dos interacciones mostradas en la Figura 6.14 el componente C-interacción utiliza los servicios de los componentes de tipo interfaz (C-registro-i, GestionConsultas-i) para acceder a los servicios de los otros componentes, los cuales a su vez se encargan de interactuar con los datos que se requieren. Así por ejemplo en la interacción GestionHistorial los servicios de los componentes son divididos en dos tipos de fragmentos, un tipo de ellos es usado para acceder a los datos y el otro para realizar la gestión del historial del paciente.

Nótese que también se acceden datos del lado del componente C-interacción, puesto que al ser de tipo DualidadServicios una parte de sus servicios se desempeña como servidor de datos.

El modelo de interacciones obtenido es ahora usado para refinar la comunicación entre los componentes involucrados mediante la aplicación de la regla de correspondencia interaccionConector (Sección 5.5, Capítulo V), donde a cada interacción le corresponde un conector, y a cada fragmento de servicio le corresponde un rol, siendo los mensajes procesos que serán activados por los componentes.

En la Figura 6.15 se ilustra el resultado de aplicar este refinamiento usando la interacción GestionHistorial

mostrándose el modelo obtenido junto con la especificación del conector.

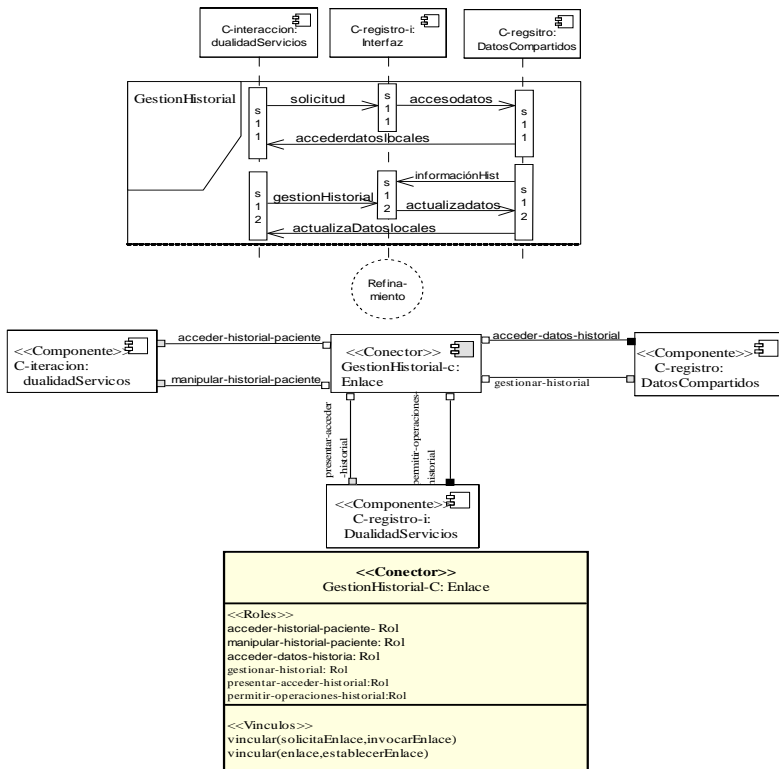


Figura 6.15 Refinamiento del modelo de componentes-y-conectores, usando la iteración GestionHistorial

Con la aplicación del refinamiento se llega a obtener un modelo más detallado de vista de componentes-y-conectores que al integrarse al resto de los componentes obtenidos anteriormente llegan a constituir la arquitectura completa de esta vista del sistema de red-de-servicios-médicos.

Por último, hay que señalar que además de los productos principales generados en cada transformación (los modelos), existe otra información que se produce durante la ejecución de una transformación, la cual consiste en los vínculos establecidos entre los objetos de los modelos participantes y las relaciones que las produjeron. Esto está en concordancia con la propuesta indicado por el estándar del manual de referencia de QVT propuesto por el OMG. Esto permite realizar un análisis del desempeño de la ejecución y detectar qué elementos y relaciones son las más y menos empleadas en las transformaciones efectuadas, lo cual compete a la gestión de modelos.

6.5 Gestión de los modelos de las vistas

Una vez que se tienen los modelos de las dos vistas, surgen las siguientes preguntas: ¿cómo actualizar la versión entre un modelo de la misma vista, cuando este sea modificado?, ¿cómo conserva la consistencia un modelo y otro correspondiente a vistas diferentes cuando uno de ellos cambie? . Esto tiene que ver con la gestión de modelos, es decir de llevar un control sobre los cambios de un modelo, y las repercusiones de estos cambios en el mismo y los otros modelos relacionados, es decir de elementos que pertenecen a una vista distinta.

Por una parte hay que ubicar los tipos de cambios que se pueden dar dentro de un modelo, un modelo puede cambiar al suprimir uno de sus elementos, al incorporar un nuevo elemento y/o relación entre los existentes, y al modificar alguna propiedad de sus elementos. En cualquier caso se tiene que realizar una gestión de los cambios para llevar un control sobre los modelos y para conservar la consistencia entre ellos.

La gestión de los cambios entre un modelo de la misma vista se pueden dividir en dos partes. Una corresponde a los cambios del modelo que va teniendo en el transcurso de su desarrollo es decir a los cambios históricos que se van dando. En este caso cada vez que ocurre un cambio en el modelo se dispone de la versión anterior y la nueva. El manejador de modelos que se dispone dentro de Eclipse permite hacer un seguimiento de cada uno de estos cambios usando la representación del modelo en formato XMI¹, lo cual identifica cada cambio que se tiene del modelo en las fechas en que estos ocurrieron, aunque esto solamente es

¹ XMI son siglas en ingles de 'XML Metada Interchange'. Estándar de la OMG que mapea MOF a XML. XMI define como deben ser usadas las etiquetas XML que son usadas para representar modelos MOF serializados en XML.

útil para llevar un registro de los cambios que este va teniendo. Lo importante es que en caso de que la modificación de un modelo no resulte apropiada para una arquitectura, se puede regresar al modelo de alguna fecha anterior. Un ejemplo de este tipo de cambio es el que se presenta en el diseño del modelo modular mostrado en las Figuras 6.5-6.8.

El otro tipo de cambio que puede ocurrir en un mismo modelo es semejante al anterior, pero en este caso se presenta cuando el cambio es entre versiones, es decir cuando un modelo ya terminado tiene que ser modificado por cuestiones de evolución. Al igual que en el caso anterior las versiones pueden ser comparadas para detectar los cambios.

Esto se detecta por medio de la comparación de las versiones de los modelos usando la propia herramienta de software donde estos se implementaron. En la Figura 6.16, se ilustra dos tipos de cambios, arriba se despliega parte de la estructura (en forma de árbol) del modelo de la vista modular llamado RedMedica en dos versiones (la 1 en la parte izquierda y la 2 en la derecha).

En la versión 2, se agregó dentro del módulo Consulta una responsabilidad más llamada -notificar a medico-, el modelo al ser comparado con su versión anterior detecta la omisión de dicha responsabilidad, como se resalta en el enlace señalado con un rectángulo (X missing). Otra modificación que se detectó en dicha comparación de versiones fue el cambio de la propiedad llamada -tipo- de la responsabilidad diagnosticar como se ilustra en la Figura 6.16 (b), con un enlace que señala el cambio de esta propiedad entre las dos versiones.

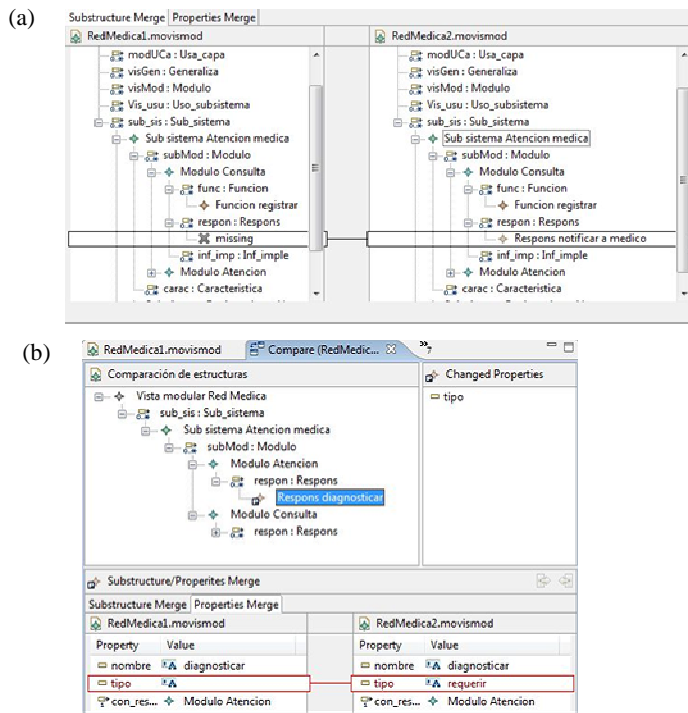


Figura 6.16 Resultado de aplicar una comparación entre dos versiones del modelo de la vista modular: (a) detectando un elemento nuevo, y (b) detectando un cambio en una propiedad.

La detección de cambios entre versiones es útil para llevar el control en la evolución de un modelo, pero lo más interesante de ello es propagar los cambios de un modelo de una vista al modelo de otra vista para conservar la consistencia entre ellos. La propagación de los cambios se logra mediante la aplicación de las reglas de transformación usando como fuente al modelo que ha cambiado, ya que en las reglas se mantiene la información de cómo deben vincularse los elementos. La transformación que se aplica y por consiguiente sus reglas dependen del modelo que llegue a cambiar, como se muestra en el esquema de la Figura 6.17.

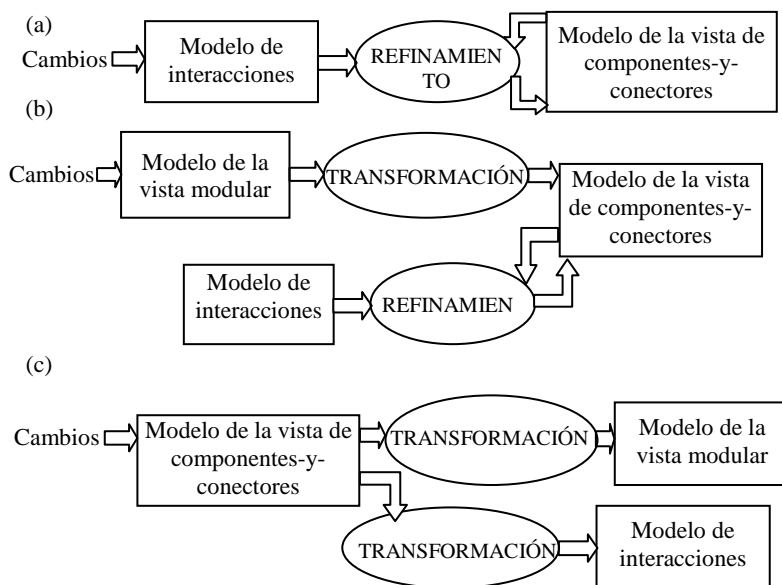


Figura 6.17. Transformaciones requeridas por cambios en el modelo (a) de interacciones, (b) de la vista modular, y (c) de la vista de componentes-y-conectores

De acuerdo a lo mostrado en la Figura 6.17 (a), si los cambios se producen en el modelo de interacciones entonces la propagación de estos sólo repercute en el modelo de la vista componentes-y-conectores ya que no tienen que ver con los elementos de la vista modular. En tal caso se aplica de nuevo el modelo de interacciones como fuente en la aplicación de la transformación (o refinamiento) y se obtiene la vista actualizada, como se muestra en el esquema (a) de la Figura antes referida.

Por ejemplo partiendo del modelo de interacción mostrado en la Figura 6.15, si se requiere hacer un cambio por agregar un objeto componente llamado C-respaldo de tipo DatosCompartidos, el cual sirve para conservar una copia de seguridad de los datos de registro de los pacientes, entonces se aplica una nueva transformación al modelo modificado y

el modelo resultante de componentes-y-conectores reflejará este cambio como se ilustra en la Figura 6.18, donde se han sombreado los elementos que no cambiaron, también se han modificado los enlaces dentro de conector que vincula al nuevo elemento con los demás.

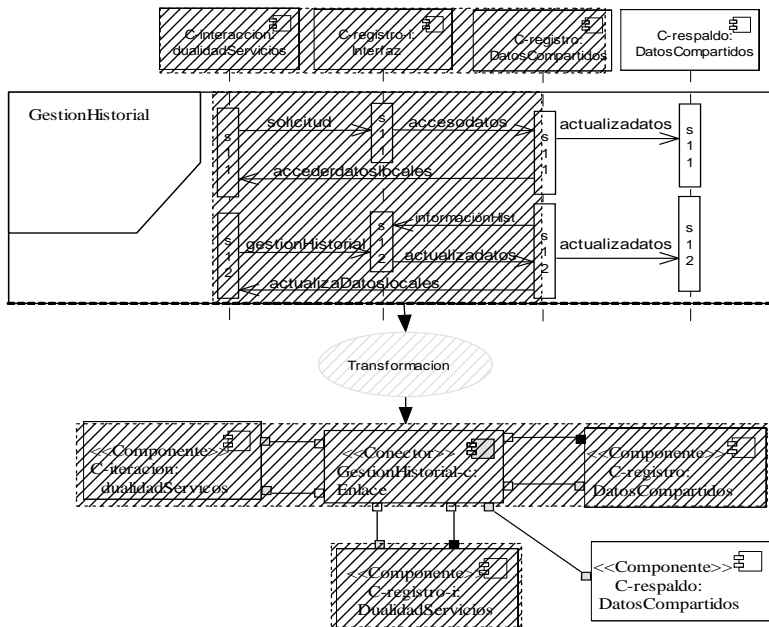


Figura 6.18 Transformación con un cambio en el modelo de interacción

De acuerdo al esquema mostrado en la Figura 6.17 (b), en caso de que haya una modificación en el modelo de la vista modular, entonces se tendrá que hacer otra vez la transformación para generar la nueva vista de componentes y luego aplicar un refinamiento desde el modelo de interacción al de componentes para detallar los enlaces en los conectores. Para mostrar la forma de realizar este tipo de

tipos de elementos que pueden cambiar en esta vista son los componentes y/o conectores.

En el caso de sea un componente el que se modifique llegará a afectar a elementos de la vista modular, pero también puede alterar al modelo de interacción si este componente posee objetos que estén involucrados. Pero en caso de que sea un conector el que sea modificado invariablemente alterará a los otros dos modelos, puesto que afecta las relaciones entre elementos.

Como se aprecia en cada caso, siempre que ocurre un cambio se aplica una transformación usando al modelo modificado en forma completa (parte modificada + parte sin cambios), por lo que el modelo destino se reconstruye totalmente. Esto se debe en parte a que la clave para mantener la consistencia entre los modelos reside en que se usen a sus meta-modelos como referencias y a las relaciones entre ellos como el adhesivo para crear los vínculos, en lugar de guardar enlaces entre los elementos a nivel de modelos. Por lo que los meta-modelos de las vistas arquitectónicas y sus relaciones aquí diseñadas (Capítulo V), son el fundamento para llevar a cabo la gestión de sus modelos mediante las secuencias de transformaciones presentadas en esta sección.

Esto hace que la forma de mantener la consistencia sea más funcional y operativa a diferencias de las propuestas hechas por (Fradet et al., 1999) y (Muskens et al., 2005) que sólo lo tratan la consistencia entre vistas arquitectónicas a nivel teórico, además de que usan una percepción muy particular de lo que es una vista, apartada del concepto dado SEI aquí tomada como base.

6.6 Análisis de las relaciones en el modelo de la vista modular

Hasta este punto se ha visto cómo se genera un modelo de una vista de componentes-y-conectores a partir de un modelo de la vista modular, y la manera de mantener la consistencia mediante la gestión de modelos a través de las transformaciones, pero aún falta realizar un análisis a nivel de modelo de las vistas arquitectónicas, sobre todo de la vista modular que es de donde se deriva la otra vista, a esto se dedica esta sección.

Cuando se diseña un modelo de alguna de la vistas arquitectónicas se tiene como referencia a su meta-modelo correspondiente, de tal manera que éste se crea de acuerdo a una estructura previamente definida sin que haya la posibilidad de incorporar algún elemento o relación ajeno a esta estructura, esto hace que la creación de un modelo sea consistente (ver Apéndice A). La pregunta que surge por lo tanto es ¿Qué propiedades deben de cuidarse al diseñar el modelo de una arquitectura de software?.

Para dar respuesta a ello, se parte de la taxonomía hecha por (Bratthall y Runeson, 1999) que agrupa a las propiedades de una arquitectura de software en tres dimensiones: nivel de abstracción, dinamismo y nivel de agregación.

En la primera dimensión se contemplan las propiedades que tienen que ver con los meta-modelos ya que ahí es donde se plasma la abstracción, en la segunda dimensión se incluyen propiedades del comportamiento de la arquitectura (en esta tesis esto se trata a nivel de modelado en la vista de componentes-y-conectores y en su modelo de interacciones), y en la tercer propiedad se considera las

propiedades que son parte del desarrollo y evolución de una arquitectura que corresponden precisamente a la vista modular, estas propiedades son las que aquí serán tomadas en cuenta, específicamente la dependencia que se crea entre sus elementos.

La dependencia entre los elementos de una arquitectura de software han sido abordada por (Sangal et al., 2005) para determinar la complejidad de una arquitectura, ahí se usa una matriz de estructura de dependencias que permite administrar los elementos mediante la inserción, eliminación y modificación de columnas y/o renglones.

Siguiendo este principio, aquí también se usa un tipo de matriz de dependencias pero en este caso para representar los tipos de relación que se forma en cada vista, y para evaluar el grado de dependencia que tiene un elemento de un modelo (modulo o componente) sobre otro elemento a través de un conjunto de indicadores.

La matriz llamada de relación aquí empleada (Limon y Ramos, 2006), parte del hecho de que dos elementos arquitectónicos (módulos o componentes) se encuentran vinculados a través de algún enlace que califica o define su relación. La matriz básica que muestra el vínculo entre dos elementos se describe en la Figura 6.20

↓→	A	B
A	0	V_x
B	V_y	0

Figura 6.20. Matriz de relaciones básica entre dos elementos

Las celdas con valores V_x , y V_y representan algún tipo de relación entre los elementos de su columna y renglón respectivo, dependiendo del valor numérico que se le asigne tendrá una interpretación. El conjunto de valores que

podrán tomar las celdas es una sucesión de valores a partir del 0 hasta R-1, donde R representa la cardinalidad de ese conjunto y cuyo valor dependerá de la diversidad de relaciones (estilos en nuestro caso) que se quieran manejar, por ejemplo si $R = 3$, significa que los valores podrán ser $\{0,1,2\}$, el valor 0 indica que no existe relación alguna, 1 indica una relación en un sentido, 2 indica que existe una relación en dos sentidos, en la matriz que no se consideran la relación de un elemento con el mismo (de ahí que los valores de su diagonal principal sean de cero).

La matriz de relación se lee partiendo de los elementos que están en cada renglón: el elemento del renglón (A o B) se relaciona de x manera con el elemento que está en una columna (A o B). Los valores V_x y V_y en una celda son usados para especificar ya sea una relación de dependencia entre un elemento o el sentido en se comunican. En la figura 6.21 se muestran tres posibles casos de relación con un valor de que R igual a 2, así los valores de una celda pueden ser $\{0, 1\}$, en forma genérica 0 significa ninguna relación, 1 un tipo de relación(o comunicación en un sentido).

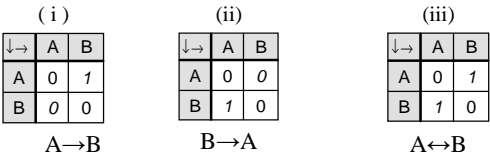


Figura 6.21. Tres casos de relaciones con $R=2$

En el caso (i) representado también como $A \rightarrow B$, se interpretaría como que A le comunica algo a B. El caso (ii) es el recíproco del caso anterior, y el caso (iii) indica una comunicación entre A y B, y otra entre B y A ($A \leftrightarrow B$). Así que sólo hay dos tipos distintos de relaciones (i, iii, o ii,iii).

Si $R = 3$, los valores a usar son $\{0,1,2\}$, 0 sin comunicación, 1 un tipo x de comunicación (indicada con \rightarrow), 2 otro tipo y de comunicación (indicada con \Rightarrow), en la figura 6.22 se muestran los casos posibles (en la parte inferior de cada matriz se representa el tipo de relación), se excluyen los ya presentados en la figura 6.21, note que en total solo hay cinco casos diferentes de los ocho posibles presentados.

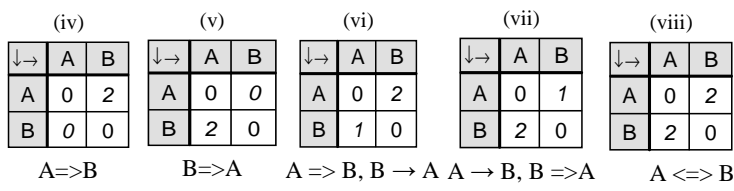


Figura 6.22. Complemento de los casos para $R=3$

Las siguientes notaciones complementan la propuesta: a) Si R es el número de tipo de relaciones que se establecen, entonces se pueden generar $R^2 - R$ casos diferentes de relaciones entre dos elementos; b) El valor de R dependerá del número de tipos de relaciones (dependiendo de los estilos arquitectónicos) que se consideren en una vista.

Para el caso de la vista modular cuyos tipos de relaciones son descomposición, generalización, uso, y usa-cap (Sección 3.4, Capítulo III)¹, se han hecho sus respectivas representaciones matriciales (Limon y Ramos, 2006-b), las cuales se describen a continuación.

¹ Aunque en la vista modular también se incluye a la relación de generalización, para este análisis no será considerada por no generar una dependencia entre elementos, ya que solo son especializaciones.

↓→	A	D		E
		B	C	
A	0	V _i	V _{ii}	V _{iii}
D	B	V	0	V _v
	C	V _{vi}	V _{vii}	0
E	V _{ix}	V _x	V _{xi}	0

(a)

↓→	A	D		E
		B	C	
A	0	0	0	0
D	B	1	0	1
	C	1	1	0
E	2	0	0	0

(b)

↓→	A	D	E
A	0	0	0
D	+	1	0
E	2	0	0

(c)

Figura 6.23. Relación de descomposición: (a) en forma general, (b) un ejemplo que incluye una relación, (c) ocultando los elementos.

La relación de descomposición. Como fue tratado anteriormente, esta relación indica que un módulo ha sido descompuesto en otros, de tal manera que la matriz debe poseer un elemento contenedor como se muestra en la Figura 6.23(a), donde el elemento D (contenedor) ha sido descompuesto en dos elementos (B,C), en la parte (b) de esta Figura se presentan un ejemplo de cómo se pueden relacionar, el mismo ejemplo se presenta en forma simplificada en la parte (c) de esta Figura, ocultándose los elementos del contenedor (+ indica que puede ser expandido).

Relación usa. Esta relación expresa una relación de dependencia sobre el elemento que se usa, y constituye en sí el fundamento de cualquier otra relación de esta vista modular, en la Figura 6.24 (a) se muestra la representación de la relación en UML y su equivalente representación en forma matricial, donde se aprecia que el elemento A depende de B, y en la parte (b) de esta Figura se describe una relación de descomposición con una de usa.

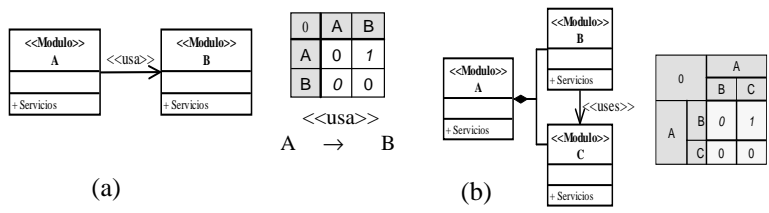


Figura 6.24. Representación matricial de la relación *usa*, (a) en forma básica, (b) combinándola con una relación de descomposición

Relación *usa-cap*. Este tipo de relación se semeja en su representación matricial al de descomposición, sólo que con la variante precisamente de agregarle la información sobre la capa en que está ubicando y la identificación de la misma tal como se ilustra en la Figura 6.25. El identificador *IdeEle* representa a un elemento de otra capa usado por un elemento ubicado en la capa mostrada (que se encuentra en el renglón), por ejemplo en la Figura referida hay dos elementos de otras capas ('X', 'Y') que están siendo usados por elementos ('A', 'D') que están en la capa representada en la matriz.

Num. Capa	IdEle	A	D		E
			B	C	
D	A	X	0	v	v
	B	—	v	0	v
	C	—	v	0	v
E	Y	v	v	v	0

Donde: NumCapa = 0,1,2,3....
V= 0,1,2....
IdEle = identificación (literal o número) de un elemento de una matriz de diferente capa

Figura 6.25 Representación matricial de la relación de

La aplicación de la matriz de dependencia al modelo de la vista modular de la red-medica (Figura 6.9) se presenta en la Figura 6.26. En los encabezados de las columnas y renglones de esta matriz se encuentran los nombres de los módulos que forman el modelo. Las relaciones que se describen son descomposición y *usa* (en este caso no se contempló una relación de capa).

La relación de descomposición se indica mediante la desagregación que se hace de los módulos *AsistenciaMedicaRemota* y *GestionConsultas*, y la relación de usa mediante los valores numéricos de cada celda, el número además de indicar esta relación hace mención a las responsabilidades de módulo que se usa, la parte entera de éste hace alusión a la relación y la parte decimal indica el número de responsabilidades que tiene el módulo que se usa.

Así por ejemplo la relación usa entre interacción y Enlace se expresa con 1.3, el valor 1 indica la relación y el 3 indica que el primero emplea 3 responsabilidades del segundo elemento.

red-médica		AsistenciaMedicaRemota			GestionConsultas			Canalizac	Suminsist	FormarRu
		Enlace (a)	Interaccio n (b)	registro (c)	Ubicacion Atencion (d)	Registrar Citas (e)	Seguimie ntoCitas (f)	ionServici os (g)	roMedical Material (h)	tas (i)
Asisten ciaMedi caRemo ta	Enlace (a)									
	Interac cion (b)	1.3		1.3	1.2	1.2	1.2			
	registr o (c)									
Gestion Consult as	Ubicaci onAten cion (d)									
	Registr arCitas (e)				1.2		1.2			
	Seguim ientoCit as (f)									
CanalizacionServi cios (g)					1.2					1.2
SuminsistroMedica lMaterial (h)										1.2

Figura 6.26 Representación matricial del modelo de la vista modular del sistema red-médica

La representación matricial del modelo hace más clara la relación entre sus elementos, y puede ser usada desde el

momento en que se inicia con el diseño de la arquitectura como una herramienta analítica. Por ejemplo mediante ella se puede conocer el número de módulos que son usados por otros (sumando los módulos que estén en su renglón respectivo), qué módulos son los más usados (determinando el máximo de los módulos que son usados en cada columna), cuántos y qué módulos están contenidos en otros (sumando los módulos de cada modulo contenedor) .

Por esta razón en (Limon y Ramos, 2007) se presentan algunos indicadores que permiten cuantificar el grado de dependencia y el nivel de descomposición que se tiene en un modelo, estos son mostrados en Tabla 6.1.

Tabla 6.1 Indicadores para medir el nivel de dependencia

INDICADORES	
	MU _{i,m} : Significa que el modulo <i>m</i> es usado por el modulo <i>i</i> .
(1)	ND _m : Nivel de dependencia sobre un modulo <i>m</i> , representa el número de módulos que usan al modulo <i>m</i> .
(2)	NX : Máximo nivel de dependencia en el modelo de la vista , esto es el MAX(ND _m)
(3)	NM _m : Número de módulos incluidos en el modulo <i>m</i> . Para el caso de la relación de descomposición.
(4)	TM : Total de módulos contenedores que hay en la vista
(5)	PM : Promedio de módulos contenidos en módulos contenedores
(6)	DM _m : Densidad del modulo <i>m</i> , donde <i>m</i> es el identificador de un módulo de tipo contenedor.
FORMA DE EVALUARLOS	
	ND _m = Σ _{i=1} MU _{i,m} , donde i ≠ m
	NM _m = Modulos m
	TM = Modulos _v v: representa la vista de un modelo
	PM = Σ NM _i / TM
	DM _m = NM _m /PM

El indicador ND expresa la medida en que un módulo es usado por otros, mientras su valor sea más cercano al máximo nivel de dependencia (MAX(ND)), significa que se

deben implementar medidas de seguridad para garantizar la continuidad del sistema.

Para el caso del sistema red-medica, el máximo nivel de dependencia lo tiene el módulo identificado en la matriz como (d) (Figura 6.26), el valor de su ND es 3, así que se deben tomar medidas para garantizar que las responsabilidades de ese módulo (UbicaciónAtencion) se lleguen a cumplir. Por otra parte los módulos SeguimientoCitas y FormarRutas son los que más se acercan al $MAX(ND)$ con un valor de ND igual a 2, los módulos restantes apenas tienen un valor de ND igual 1.

Los indicadores del (3) al (6) (Tabla 6.1) se usan para identificar el nivel de desagregación de los módulos contenedores tal como lo indica su descripción. Los resultados que se obtienen al ser aplicados al caso de la red-medica son los siguientes: El número de módulos (NM) incluidos en los módulos contenedores AsistenciaMedicaRemota y GestionConsultas, es 3 en ambos casos. El promedio de módulos (PM) que están contenidos en otros, también vale 3, y como sólo son dos módulos contenedores su densidad modular (DM) vale 1, lo que indica que están justo en el promedio y no existe una desagregación fuerte en ellos.

Las conclusiones a las que se llegan por la evaluación de los indicadores para el caso del modelo de la vista modular del sistema de la red-medica son dos: una es que existe una débil dependencia entre los módulos, y la otra es que se tiene una baja desagregación entre sus elementos. Esto significa que la estructura de ésta arquitectura puede soportar cambios sin sufrir afectaciones (por la débil dependencia), y que hay un buen balance en la desagregación de sus módulos contenedores (por tener la misma densidad). Estas características del modelo de la vista

modular repercuten no sólo él, sino que también se llegan a reflejar en el modelo de la vista de componentes-y-conectores que se llegó a generar mediante el proceso de transformación.

Capítulo 7

Conclusiones y trabajos futuros.

En este capítulo, se presentan las conclusiones a las que se han llegado con el desarrollo de esta tesis, se detallan las contribuciones más importantes que se han realizado, las publicaciones derivadas de este trabajo de investigación y por último los trabajos futuros previstos.

7.1 Conclusiones

Partiendo de los objetivos planteados en el Capítulo I, estos han sido cubiertos de la siguiente manera:

Se ha establecido un marco de trabajo para el diseño y análisis de arquitecturas de software, este marco consiste en la especificación de los modelos base de las vistas arquitectónicas (la modular y la de componentes y conectores), y en la propuesta de un proceso para el modelado de estas vistas. La especificación de los modelos de las vistas ha sido el resultado de un análisis exhaustivo de las diversas propuestas hechas por los expertos en el área de las vistas en la arquitectura de software (estado del arte), obteniéndose la especificación de cuatro tipos de relaciones base en la vista modular (usa, usa-capa, descomposición y generalización), y seis tipos de relaciones base en la vista de componentes-y-conectores (filtro-tubería, cliente-servidor, igualdad-de-servicios, escritor-lector, y datos-compartidos) que conjuntamente con la especificación de sus tipos de elementos constituyen el fundamento para poder expresar la diversidad de modelos propuestos por los

expertos en el área y de aquellas aplicaciones que demanden una combinación de los modelos base para integrar los dos tipos vistas arquitectónicas.

La otra parte del marco de trabajo referente al proceso de modelado de las vistas, está conformado por una estrategia basada en la propuesta de OMG de la arquitectura dirigida por modelos (MDA), lo cual incorpora al modelo de las vistas las ventajas de este enfoque, la principal de ellas es la poder crear los meta-modelos de las vistas para generar modelos. De esta manera el proceso propuesto contempla las entradas, actividades, secuencia y productos en cada uno de estos dos niveles de abstracción. Las actividades indican lo que debe hacerse para llevar a cabo el modelado de las vistas, desde de la creación de los meta-modelos de las vista hasta de generación de sus modelos, dicho proceso ha sido aplicado al caso de estudio analizado aquí, demostrando su efectividad en la medida que llegó a responder a las expectativas de los requisitos de calidad planteados por los usuarios, por lo que puede ser usado para otros casos donde se requiera modelar una arquitectura software¹.

Una de las contribuciones clave en el proceso de modelado de las vistas lo constituye la forma en que se han diseñado sus meta-modelos, el hecho de haber creado los meta-modelos en forma modular separado las partes comunes (modelos base) de las relaciones que los especializan (para los dos tipos de vista considerados) hacen que no se sobrecargue su implementación y sobre todo se consigue un más ágil y menos complejo modelado de las vistas. Además en caso de que se tener que incorporar algún otro tipo de relación se puede hacer sin alterar los demás meta-modelos,

¹ En el caso de estudio propuesto se incluyen situaciones que combinan varios modelos base, de esta manera se garantiza que el proceso pueden ser aplicado a más casos.

por ejemplo la incorporación de aspectos del modelo PRISMA o de sistemas expertos en líneas de producto BON. La otra aportación que se ha hecho con el propósito de llevar a cabo el proceso de transformación es el diseño de los modelos de las relaciones que se establecen entre los elementos del meta-modelo de la vista modular con los elementos del meta-modelo de la vista de componentes-y-conectores.

La formalización de los ocho tipo de relaciones diferentes entre los meta-modelos base de las dos vistas sirven de guía para la implementación de las reglas de transformación. Han sido realizadas usando una representación para aplicarla a los lenguajes QVT-relaciones y QVT-operacional, pero son útiles también para guiar otro tipo de implementación, como con ATL por ejemplo. De esta manera se cumple con el objetivo de vincular las dos vistas a través de los modelos que relaciones a cada meta-modelo.

La estrategia de transformación que se ha seguido aquí, apegada a la propuesta de OMG de la arquitectura dirigida por modelo, ha permitido cumplir con otros dos objetivos planteado en la tesis. Uno de ellos es el de poder llevar a cabo la generación de modelos de una vista en función de otro modelo, específicamente de la vista modular a la vista de componentes-y-conectores y viceversa, con lo cual se logra una transitividad entre modelos vía transformaciones.

Además se ha establecido un vínculo entre la estructura y el comportamiento que existe entre algunos elementos arquitectónicos, mediante la incorporación de un modelo de interacciones se captura la interrelaciones que ocurre entre los componentes de la vista de componentes-y-conectores, de ahí que dicho modelo sirva de puente entre esta vista y la vista modular.

El otro objetivo que se ha cumplido usando la transformación de modelos es el de mantener la consistencia entre los modelos de las dos vistas aquí consideradas cuando alguna de ellas cambie por razones de mantenimiento o evolución.

Esto se ha logrado gracias al rol que juegan cada uno de los elementos implicados en el proceso de transformación. Por un lado los meta-modelos de las vistas sirven como plantillas para crear los modelos, así que al crear un modelo de cualquiera de las vistas estos contienen sólo lo que está especificado en su plantilla, es decir no hay elementos ni relaciones sorpresas.

Por otro lado las relaciones que se establecen entre los meta-modelos proporcionan la transitividad segura entre un modelo a otro, de tal manera que cuando se produce un cambio en alguno de los modelos basta con aplicar la transformación correspondiente (usando las relaciones) para llegar a restablecer la consistencia entre ellos. Todo ello hace que la gestión de modelos entre las vista arquitectónicas se haga casi en forma automática, ya que lo único manual que se realiza es elegir el tipo de transformación y ejecutarla.

El objetivo sobre el análisis de dependencias entre elementos de la vista modular ha sido cubierto mediante un manejo matricial para representar cada tipo de relación que se incluye en esta vista, que es donde se generan las dependencias, dicho manejo matricial ha permitido un tratamiento cuantitativo y a la vez analítico, de esta manera se pueden llegar a encontrar relaciones de dependencias que puede afectar durante el ciclo de vida de un modelo de la vista modular, ya que cuando existen fuertes dependencias, estas son reestructuradas antes de que el modelo sea usado en un proceso de transformación para

llegar a producir un modelo de la otra vista. Además tal análisis matricial se ha complementado mediante un conjunto de indicadores que sirven para precisar el grado de dependencia que se tiene dentro de un modelo. Con este análisis se logra diseñar modelos de la vista modular sin anomalías, garantizando modelos fiables.

La manera de mostrar cómo se han cumplido cada uno de los objetivos planteados en esta tesis se plasma en el caso de estudio presentado (red de atención médica). Siguiendo la metodología para la creación de sus vistas arquitectónicas y basándose en el marco de trabajo se ha logrado diseñar la arquitectura de software.

La vista modular se fue conformando al ir aplicando el proceso de descomposición iterativa, obteniéndose al final del proceso el modelo correspondiente de esta vista (especificación de módulos y relaciones). Luego al usar este modelo como fuente, y al aplicar la transformación se obtuvo la vista base de componentes-y-conectores, la cual gracias al modelo de interacciones que se incorporó se llegó a refinar esta segunda vista a través de una segunda transformación llegando a obtenerla en forma precisa.

Para probar la efectividad de la conservación de la consistencia entre los dos modelos de las vistas, se incorporó una modificación al modelo de la primera vista y una vez más gracias a la transformación de modelos se logró conservar la consistencia entre estos modelos.

En el caso de estudio también se probaron los indicadores para conocer el estado del diseño de la vista modular lo cual mostró un resultado satisfactorio en cuanto a la dependencia de los módulos, al no contar con módulos que tuvieran una interdependencia fuerte.

7.2 Aportaciones

Las aportaciones que se llegaron a producir como resultado de esta investigación son:

- Realización de una caracterización de las vistas arquitectónicas de software, esto posibilita la identificación de los modelos para las vistas arquitectónicas propuestos por diferentes como los Kruchten y SIEMS.
- Creación de una metodología para el modelado de las vista modular y de componentes-y- conectores, mediante ella se hace posible el modelado de la vista modular aplicando una descomposición iterativa y se especifican las acciones para la generación de la otra vista vía la transformación de modelos.
- Como resultado de practicar el análisis de las relaciones entre los elementos de cada una de las dos vistas, se logró determinar las relaciones básicas entre ellos y relaciones que hacen que una vista se especialice.
- Se diseñaron los meta-modelos de las dos vistas antes referidas de una manera formal, haciendo una división de los elementos base y de los que la especializan. Con ello se obtiene una mayor flexibilidad tanto en su implementación como en la generación de modelos.
- Se incorporó el comportamiento de una arquitectura mediante el diseño del meta-modelo de interacciones de un sistema, que conjuntamente con las transformaciones se consigue plasmarlo en vista

de componentes y conectores de forma refinada usando a la vista modular como punto de partida.

- Como resultado del análisis de las relaciones entre las dos vistas, se obtuvo la formalización de las correspondencias entre sus meta-modelos (usando QVT-relaciones). Lo cual condujo a su implementación (con QVT-operacional), y sirve también como referente para otras implementaciones (como ATLAS, por ejemplo).
- Se realizó la integración entre la propuesta de MDA con el diseño y gestión de vistas arquitectónicas de software. Esto produjo una simbiosis porque por el lado de las vista arquitectónicas se logran producir modelos en forma, extensiva, segura y consistente. Y por el lado del MDA se incorpora una estrategia para sustentar mejor los modelos arquitectónicos de un sistema.
- Se ha conseguido mantener la consistencia entre modelos de distintas vistas gracias a la aplicación de su gestión usando el enfoque de transformación de MDA.
- Mediante la implementación de las relaciones de la vista modular en forma matricial se ha conseguido hacer práctico su análisis, para que con el uso un conjunto de indicadores se posibilite la detección de situaciones de sobre dependencias y otras relaciones anómalas entre los elementos de esa vista.

7.3 Publicaciones

Durante el desarrollo de este trabajo de investigación, se han realizado las siguientes publicaciones:

-Limon y Ramos, Transformación de vistas arquitectónicas orientada por modelos, VI Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento,(JIISIC'07) ,31 de enero al 2 de feb del 2007,Lima, Perú pp. 161-170.

-Limon y Ramos, Analyzing Styles of the Modular Software Architecture View 1st European Conference on Software Architecture(ECSA 07) Septiembre 24-26, 2007 ,Aranjuez (Madrid) – España pp. 275-278

-Limon y Ramos, Relating Software Architecture Views by using MDA, International Conference on Computational Science and Its Applications, (ICCSA 2007), 26-29 de agosto, 2007, Kuala Lumpur, Malaysia, pp. 104-114

-Limon y Ramos, Establishing Relations among Software Architecture View through MDA for SLP, 13th. International Congress on Computer Science Research (CIICC'07) ISBN: 13978-970-95771-0-5, CIICC'07-SICI'07, 7-9 nov. 2007, ORIZABA , MEXICO

-Limon y Ramos, 2006, Using Styles to Improve the Architectural Views Design, Proceedings of the International Conference on Software Engineering Advances (ICSEA'06), 2006

- Maria Eugenia; Ramos, Isidro; Gomez, Abel; Limon, Rogelio. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for the Expert Systems development

Cabello, ACIIDS 2009. 1st. Asian Conference on Intelligent Information and Database Systems, 2009.

7.4 Trabajos Futuros

Dado que este trabajo ha sido desarrollado dentro del contexto de un grupo de investigación sobre ingeniería de software y sistemas de información (ISSI), la dirección que se pretende seguir con los trabajos a desarrollar va en dos sentidos, uno es continuar con una profundización sobre temas relacionados con la arquitectura de software y meta-modelado, y el otro sentido va sobre la vinculación de las vistas con otros trabajos desarrollados por el grupo ISSI.

Por el lado de la profundización sobre vistas arquitectónicas, se pretende abarcar transformaciones entre otro tipos de vistas como la de concurrencia que son aplicables a sistemas de tiempo real, donde los meta-modelos de la vista de componentes-y-conectores tienen que ser aún más especializados al tener que incorporar componentes que realicen multitareas y componentes que realicen el control entre ellos. Con ello se pretende extender las transformaciones a tipos de sistemas más dinámicos.

Otro reto inmediato a enfrentar también en el sentido de profundizar este trabajo, es sobre el nivel de refinamiento de la arquitectura de software. Las transformaciones realizadas aquí han sido del tipo modelo-a-modelo ya que al pasar de una vista a otra se continúa en el mismo nivel de abstracción, pero para poder acercarse al nivel de implementación hace falta una transformación de modelo-a-código lo cual posibilita que una vista arquitectónica sea pasada a una granularidad más detallada, es decir a un código específico para ser ejecutada. Por lo que una continuación inmediata de esta tesis será la conversión de

los modelos de las vista de componentes-y-conectores al lenguaje PRISMA, con ello se aprovechará todo el desarrollo sobre implementación que tiene este lenguaje creando una sinergia entre ambos trabajos. Esto servirá como referente para establecer conversiones hacia otros lenguajes arquitectónicos.

En la vertiente correspondiente a la vinculación de las vistas arquitectónicas con los otros productos realizados en ISSI, se pretende integrar las vistas-arquitectónicas con los proyectos que tienen relación con la arquitectura de software o que son una extensión de la misma. En el primer caso se pretende la incorporación de las vistas al principal proyecto que abrió la línea sobre arquitecturas-software conocido como PRISMA. Este proyecto maneja la arquitectura con orientación a aspectos usando componentes y conectores específicos para tal fin, lo que se pretende ahora es incorporar a PRISMA como una vista especializada.

Así que tomando como referencia la metodología para el tratamiento y diseño entre las vistas aquí analizadas (Figura 4.6), se hará una adaptación para que los aspectos de un sistema sean identificados e incorporados en la vista de componentes-y-conectores y a través de transformaciones se llegue a producir una vista PRISMA. Esto sería complementado con el refinamiento de la vista a nivel de código, como se menciona en el párrafo inicial de esta página, para tener tanto el modelo de esta vista y a la vez su implementación.

El otro proyecto con que se pretende vincular a las vistas arquitectónicas es con el proyecto BOM. Dicho proyecto es el resultado de una tesis doctoral (Cabello 2008), en el cual se destaca la aplicación de líneas de producto con la generación de sistemas expertos. Uno de los primeros

acercamientos en esta vinculación ha sido presentada en (Cabello et al, 2009).

Como se aprecia, en la gestión de las vistas usando transformación de modelos basados en MDA hay mucho camino que recorrer, lo realizado en este trabajo es sólo el primer paso.

APENDICE A

Implementación de los meta-modelos de las vistas

El diseño de los meta-modelos de las vistas modular y de componentes-y-conectores expuestos en el capítulo V se han implementado dentro del marco del software ECLIPSE. Este software tiene la ventaja de ser adaptativo al permitir incorporar lo ahí llamado conectores (conocidos como *plugins*), dicha propiedad hace que se puedan incorporar conectores para poder implementar los meta-modelos y llevar a cabo el proceso de transformación, permitiendo la continuidad entre el diseño de un meta-modelo con sus modelos. En este apéndice se aborda cómo se implementó a los meta-modelos de las vistas y cómo se realiza la transición hacia los modelos.

La implementación de los meta-modelos consiste en dos fases, la primera comprende la especificación de cada meta-clase y la segunda en el enlace de navegación entre las clases.

Cada clase es especificada con su identificador y atributos, en este caso no se incluyen métodos, puesto que precisamente con el enfoque MDA se representa sólo la estructura de un modelo. Los atributos pertenecen al dominio de tipos de OCL.

La especificación de los enlaces de navegación corresponde al diseño de las relaciones de los meta-modelos de las vistas con el enfoque MDA, estas son adaptadas a las características del lenguaje que se usa y a la herramienta tecnológica empleada para implementarlo, en este caso se

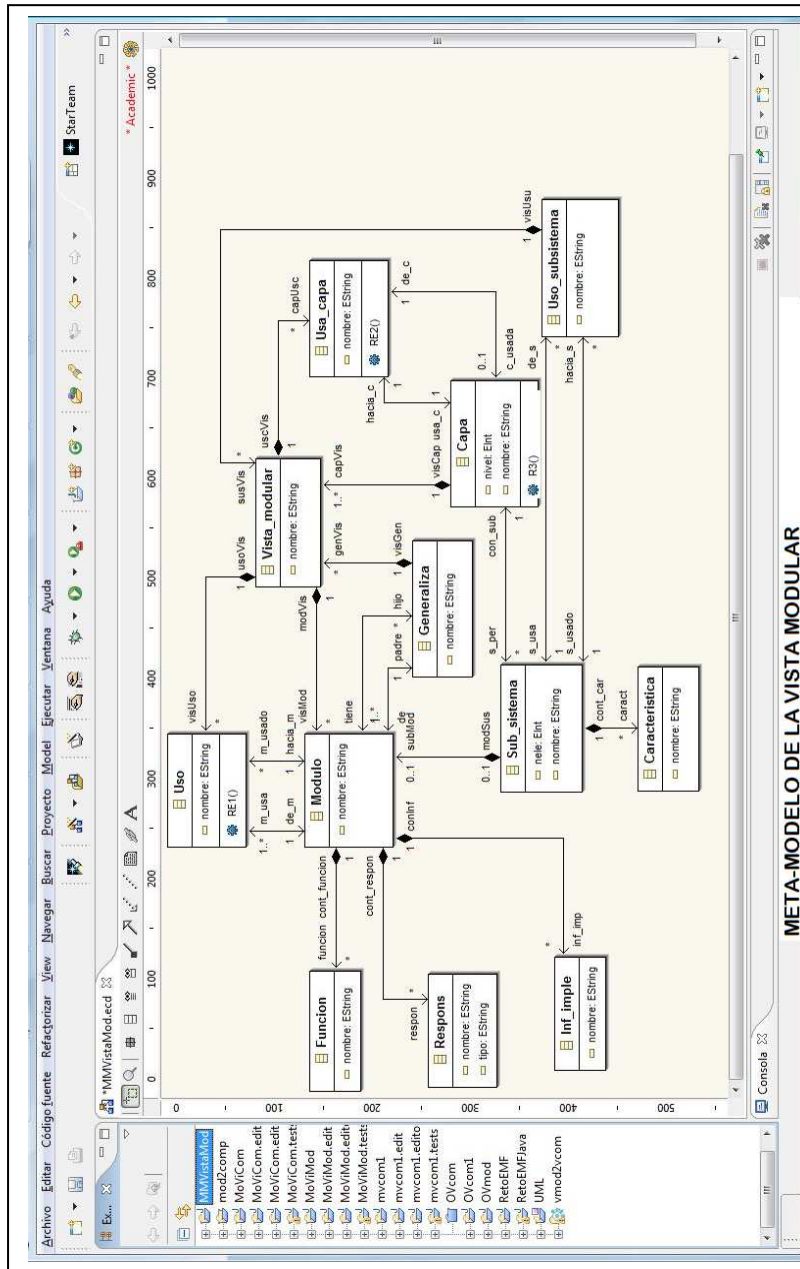
hace una adaptación al lenguaje QVT-operacional que se usa dentro de la herramienta tecnológica de ECLIPSE.

Este lenguaje impone que la implementación del meta-modelo siga una estructura de un árbol, donde la raíz la constituye la identificación del meta-modelo, en este caso el de cada vista. De ella se derivan los elementos más importantes, y de estos se desprenden otros y así sucesivamente. Con los elementos derivados se establecen relaciones de asociación mediante enlaces de navegación que permiten transitar de una meta-clase a otra, estas pueden ser en un sólo sentido o bidireccionales.

Las relaciones de generalización que existen son descompuestas en forma simple, ya que su implementación no permite la comunicación de la información en este tipo de relación. Así los atributos que se incluyen en la clase padre de una generalización son incorporados en las clases hijas. La razón es que en una transformación no se guarda el enlace que los vincula.

Tomando en cuenta estas consideraciones de implementación, se crearon los meta-modelos de cada vista, cuyos diagramas de clase se muestran a continuación¹:

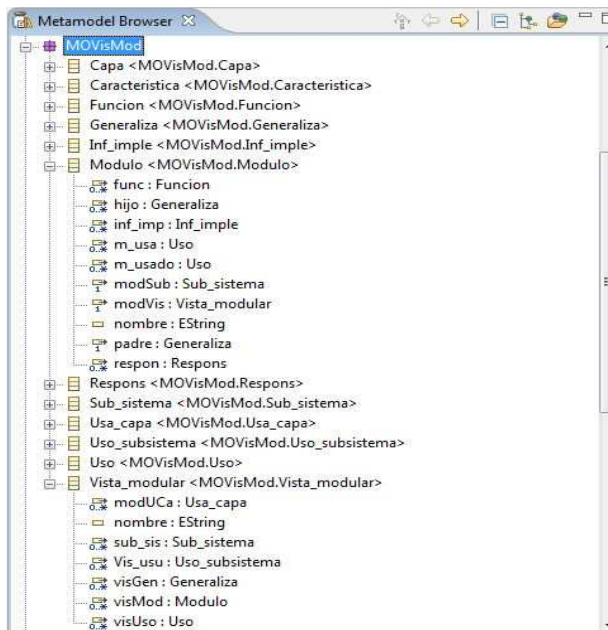
¹ Se usó como herramienta a OMONDO-ECLIPSE





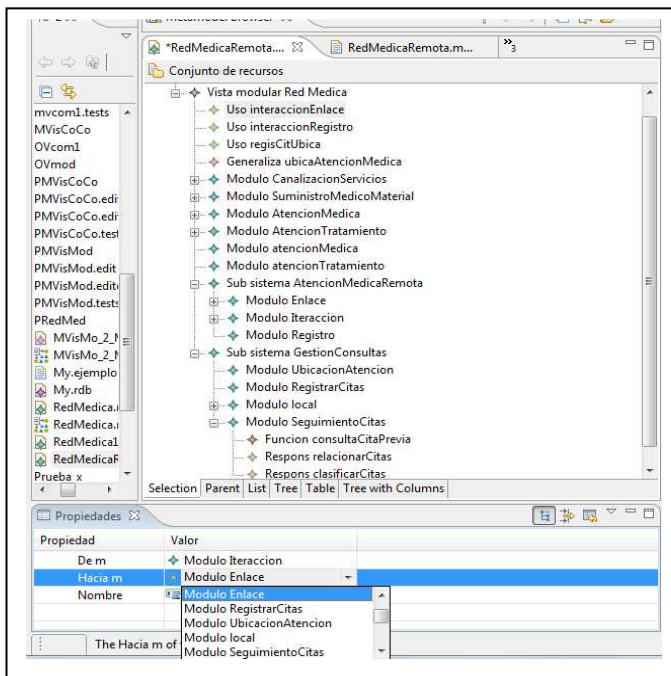
Una vez que se disponen de los meta-modelos de cada vista, se pueden crear modelos a partir de ellos mediante la generación de conectores que almacenan al meta-modelo para usarlo como patrón en la creación de modelos.

A continuación se ilustra el patrón del meta-modelo de la vista de modular, en él se aprecia la estructura de árbol mencionada anteriormente con sus meta-clases correspondientes. Dentro de cada una de ellas se tiene la información detallada correspondiente a sus atributos, enlaces con las otras clases y sus relaciones. En el meta-modelo mostrado se despliega el detalle de dos meta-clases, la de Modulo y de Vista_modular. La relaciones se indican siguiendo el patrón “enlace: Metaclase”. Así por ejemplo “func : Funcion”, indica que el enlace func establece el vínculo de la meta-clase Modulo hacia la meta-clase Funcion . Estas especificación corresponde exactamente al diseño de su meta-modelo en el diagrama de clases presentado anteriormente (de ahí que se omite el despliegue de las demás meta-clases).



Con estos patrones se crean los modelos requeridos para producir las transformaciones. En el caso del sistema Red-Medica se crea su modelo correspondiente de la vista modular usando al patrón anteriormente especificado, el modelo creado se muestra a continuación en forma jerárquica, teniendo como raíz el nombre del modelo (Vista modular Red Medica), de la cual dependen los demás elementos. Al inicio de este modelo se muestran cuatro relaciones (tres de uso y la otra de generalización), y en seguida todos los demás elementos constituidos por módulos y sub-sistemas, de estos últimos se despliegan los módulos ahí contenidos, y el último de ellos despliega su detalle.

Cada elemento tiene asociadas sus propiedades, en la parte inferior del modelo se ilustra cómo la relación “Uso interaccionEnlace” tiene como propiedades a los módulos que vincula (De m: Interacción, Hacia m: Enlace). Estas propiedades se van especificando conforme el modelo se diseña.



El modelo de la vista modular creado es usado a su vez como entrada para llevar a cabo la transformación a su modelo de la vista de componentes-y-conectores correspondiente, de acuerdo al proceso indicado en el Capítulo V, y a su implementación vista en el Capítulo VI.

APENDICE B

Codificación de las relaciones de correspondencias en QVT-Relaciones

Las relaciones de correspondencias aplicadas para la transformación de modelos de la vista modular a modelos de la vista de componentes-y- conectores, representada en notación gráfica en el capítulo V, se sustentan en el lenguaje QVT-Relaciones con el que se formulan las reglas que los meta-modelos correspondientes utilizan para llevar a cabo este proceso.

Antes de iniciar con la descripción del código de las relaciones se presentan las siguientes consideraciones:

- Aquí se le designa a los objetos de la clase Modular como fuentes(identificados por la clausula chekonly) , ya que ellos se usan como entrada para crear a los objetos de la clase Componentes-y-Conectores (identificados por la clausula enforce), estos últimos serán designados como objetos destino.
- Los nombres usados para las relaciones, meta-classes y de las instancian de estas son los mismos que se tienen en sus diagramas respectivos, las cuales están resumidas en la Tabla 5.2 del capítulo V. Su código será presentado en el mismo orden que el mostrado en la Tabla referida, describiendo lo más relevante de éste.

Relación: $\text{Modulo} \xrightarrow{R} \text{Componente}$

Esta relación aunque es básica es fundamental, porque es empleada en casi todas las demás transformaciones. En ella un objeto Modulo (m) sólo verifica que esté en su dominio(mod). Si éste existe, se crea un objeto Componente (c) en su dominio correspondiente (com) asignándole el mismo identificador (nombre) concatenándole el prefijo 'C', a la vez que se activan la relaciones funcionAPropiedades y responsabilidadAServicio especificadas en la clausula where, comunicándole las instancias de las meta-classes correspondientes (m,c) para propagar sus cambios.

```
top relation moduloAComponente
{ nombre: String;
checkonly domain mod m: Modulo{nombre = nom};
enforce domain com c: Componente{nombre =
    'C'+nom};
    where{
        funcionApropiedades(m,c)
        responsabilidadAServicio(m,c)
    }
}
```

Relacion: $\text{Funcion} \xrightarrow{R} \text{Propiedades}$

Al ser una relación complementaria, está usa el atributo Funcion del objeto Modulo y le comunica su nombre a un atributo Propiedad del objeto Componente creado, especificando que el tipo de esta propiedad es función, la cual se utiliza cuando se quiere regresar al modelo modular.

```
relation funcionApropiedades
{ nombre, tipo: String;
checkonly domain mod m: Modulo{
    func = fun: Funcion {
        nombre=nom
    }
enforce domain con c: Componente{
    prop = pro: Propiedad {
```

```

                                nombre = nom,
                                tipo =
'funcion'
                                }
                                }

```

Relacion: *Responsabilidad* \xrightarrow{R} *Servicio*

Como en el caso anterior se usan los atributos de los objetos de las clases comunicadas, pero ahora se crean otras propiedades en el objeto destino (los puertos), activándose una función que sirve para determinar su tipo, el puerto será de entrada si el tipo de la función es 'Local', y el puerto será de salida si la función obtiene un resultado, estos tipos se pueden modificar cuando un componente sea refinado.

```

relation responsabilidadAServicio
{ nom, ntip, nser, tipoPto: String;
  checkonly domain mod m: Modulo{
    func = fun: Funcion {
      nombre=nom,
      tipo= tip : Tipo {nombre = ntip}}
    }
  enforce domain con c: Componente{
    servicio = ser : Servicio {
      nombre =
      nser},
      puerto= pue : Puerto { nombre=
      'pto'+nser,
      tipo =
      tipoPto
      }
    }
  Where {
    tipoPto= tipoPuerto(ntip)
  }

function tipoPuerto(tipoFun:String):String;
{ if (tipoFun = 'LOCAL') then 'Entrada'
  else if (tipoFun = 'RESULTADO') then 'Salida'
  endif
  endif;
}

```

}

Relacion: $Subsistema \xrightarrow{R} Componente\ Compuesto$

En esta relación cada uno de los elementos que constituyen a un subsistema (módulos) llegan a generar sus respectivos componentes, además de crear un componente de tipo compuesto.

```

top relation subsistemaAcompoCom
{ nc, nct, nco, npn, nps, ncomp : String;
checkonly domain rcomp rc: Composicion {
    nombre=nc,
    prop = mp:Modulo { nombre= npn}
    sub = ms:Modulo { nombre=nps}
}
enforce domain cc: Componente
{ name = nct,   contenedor = cc;
  componente = c : Componente {
    nombre = nco }
}
}

```

Relacion: $usa \xrightarrow{R} conector$

Una relación usa entre módulos generará un conector que vincula a los componentes que son producidos previamente por una conversión de módulos a componentes. Por lo tanto, se tiene como requisito que los módulos vinculados con usa existan, este requisito se cumple al incluir a los dos objetos de esta clase (nombrados aquí como usa y usado) dentro de la verificación del dominio Usa.

Por otro al generarse un elemento conector, también se crean los componentes que se vinculan mediante él, para lo

cual se establece como condición que antes de llevar a cabo su creación dichos componentes se generen usando a los componentes incluidos en la verificación, esto se logra llamando a la relación de correspondencia moduloAComponente (una por cada componente) dentro de la clausula when que actúa como precondition.

Al generarse el conector y los componentes respectivos, se crea también un rol conector y un puerto por cada componente, mediante los cuales se establece su enlace.

```

top relation usaAConector
{
  ser_1, ser_2 :Servicio;
  comp_1, comp_2:Componente;
  nma,nmo,ncon,nfa,nfo,tfa,tfo : String.
  checkonly domain reluse ru: Usa
  { nombre = nuse,
    usa = am: Modulo
      {
        nombre = nma,
        fun= fa : Funcion { nombre = nsa
                           tipo =tfa
                           }
      }
    usado = om : Modulo
      {
        nombre = nmo,
        sfun = fo: Funcion {
                           nombre = nfo,
                           tipo= tfo
                           }
      }
    enforce domain cn : conector {
      nombre= ncon,
      propiedad = ra : Rol { nombre = ncon+'rol',
                             puerto pa : Puerto
      }
    { nombre= npa, tipo= tfa }
  }
  enforce domain pb : Puerto {
    Nombre = npb, tipo = tfo,

```

```

ncon+'rol'
                                Rol= rb: Rol { nombre =
                                tipo = tfb
                                }
                                }
when {
  moduloAComponente (am,comp_1);
  moduloAComponente (om,comp_2);
  funcionAServicio (fa,serv_1);
  funcionAServicio (fb,serv_2);
}
}

```

Con la especificación de estas relaciones de correspondencia se formaliza cada una de las transformaciones de los elementos entre una vista modular y la vista de componentes-y- conectores, sin embargo para producir la transformaciones se usa el lenguaje QVT-operacional (anexo c).

APENDICE C

Codificación de las transformaciones entre modelos de las vistas en QVT-Operacional

Para poder llevar a cabo cada una de las transformaciones entre los modelos de las vistas implicadas se usa el lenguaje QVT-Operacional(OMG 2006). Este lenguaje de aproximación híbrida (combina lo declarativo con lo imperativo) es propuesto por MDA con el propósito de implementar dichas transformaciones mediante diversas operaciones.

Las operaciones de transformación permiten crear o actualizar un modelo usando a otro como fuente, donde el modelo fuente es equivalente al que se especifica en la clausula checkonly dentro del lenguaje de relaciones, y el modelo creado es equivalente al que se especifica con la clausula enforce del lenguaje referido. Una correspondencia se realiza tratando a los modelos como colecciones de elementos (conjuntos simples, conjuntos ordenados), en forma semejante a como se hace el lenguaje OCL 2.0. En este caso la vista modular es el conjunto de elementos fuente que será mapeado para crear o actualizan elementos y/o relaciones de la vista de componentes-y-conectores.

Existen varias herramientas de software con las que se puede implementar la especificación de lenguaje QVT-Operacional para la transformación entre las vistas arquitectónicas. Aquí se ha usando la misma herramienta con la que se han diseñado los meta-modelos de las vistas, es decir la plataforma ECLIPSE, de esta manera los meta-modelos podrán ser ocupados como los dominios de referencia para hacer la aplicación.

Para realizar el proceso de transformación lo primero que hay que especificar son los meta-modelos a usar (MOVisMod'; MVisCoCO'), los cuales son importados tal como se muestra en las líneas 2 y 3 del código mostrado a continuación:

```
1 transformation MOVisMod_To_MVisCoCO;
2 metamodel 'http://MOVisMod';
3 metamodel 'http://MVisCoCO';
```

La transformación que desencadena a todas las demás transformaciones (main) inicia con los elementos de la vista modular que tienen un mayor nivel de agregación para que a través de ellos se vayan creando los elementos del mismo nivel de agregación de la otra vista. En la líneas de código 7,9 y 11 se transforman a los módulos (visMod), subsistemas (sub_sis), y a las relaciones de uso (visUso) para crear sus correspondientes elementos de la vista componentes-y-conectores como colecciones ordenadas de elementos: componentes (viscom), componentes-compuestos (viscomc), y a los conectores (viscon).

```
4 mapping main(in model:
  MOVisMod::Vista_modular):
  MVisCoCO::VistaCom_Con {
5   object {
6     nombre:= 'v-c-c'+model.nombre;
7     viscom := model.visMod->
8       collect(emod|emod.mod2com(' '))-
9       >asOrderedSet();
10    viscomc := model.sub_sis->
11      collect(esus|esus.sus2coc())-
12      >asOrderedSet();
13    viscon := model.visUso->
14      collect(eusa|
        eusa.uso2con(eusa.nombre))->asOrderedSet();
15  }
```

La construcción de los demás elementos se va propagando en forma anidada llamando a sus correspondientes mapeos: EL siguiente código es el detalle de cada uno de los mapeos:

```
mapping
MOVisMod::Sub_sistema::sus2coc():MVisCoCO::Com_com
p
{
    nombre := 'C_'+nombre;
    atributo:= if(nelem>=0)then

'elemento_'+nelem.oclAsType(String)
        else ''
        endif;
    comccom := self.subMod->

collect(emod|emod.mod2com(nombre))-
>asOrderedSet();
}

mapping
MOVisMod::Modulo::mod2com(nom:String):MVisCoCO::Co
mponen
{
    nombre:= 'C_'+nom+nombre;
    compen:= self.respon->
        collect(eus|eus.res2pen())-
>asOrderedSet();
    compsa:= self.respon->
        collect(esu|esu.res2psa())-
>asOrderedSet();
}

mapping
MOVisMod::Uso::uso2con(nom:String):MVisCoCO::Conec
tor
{
    nombre := 'co_'+nombre;
    conrol := self.de_m.respon->
        collect(eres|eres.rol2res())-
>asOrderedSet();
}
```



```

mapping MOVisMod::Respons::rol2res():MVisCoCO::Rol
{
    nombre:= 'rol_'+nombre;
    tipo := 'rol_'+tipo;
    rolpen := self->collect(eres|eres.unirptoecon())->asOrderedSet();
}

mapping
MOVisMod::Respons::res2pen():MVisCoCO::PuertoEnt
{
    nombre := if(tipo='requerir')
                then 'pto_e'+nombre
                else ''
            endif;
}

mapping
MOVisMod::Respons::res2psa():MVisCoCO::PuertoSal
{
    nombre := if(tipo='provee') then
                'pto_s'+nombre
            else ''
            endif;
}

mapping
MOVisMod::Respons::unirptoecon():MVisCoCO::PuertoEnt
{
    init{
        var ptoe := self.resolve(MVisCoCO::PuertoEnt);
        var come := self.resolve(MVisCoCO::Componen);
        var role := self.resolve(MVisCoCO::Rol)->asOrderedSet();
    }
    nombre := 'C_'+nombre;
    penrol := role;
}

```

Además de las operaciones de mapeo existen otras operaciones que no modifican el estado de los elementos

(no crean, ni actualizan), solo son ocupadas para determinar el estado de en que se encuentran los elementos del modelo. Por ejemplo, saber si se cuenta con las condiciones para llevar a cabo un mapeo y por tanto ejecutar una transformación, su resultado sirve como un filtro para llevar otras acciones. Aquí las consultas servirán para conocer si las trazas de una transformación se han realizado y con ello poder establecer crear relaciones entre estos elementos. A continuación se ilustra una de ellas que sirve para conocer el tipo primitivo de un elemento.

```
query Primitive2Primitive(in name : String) :
String {
    if name = 'String' then 'varchar' else
        if name = 'Boolean' then 'int' else
            if name = 'Integer' then 'int'
        else
            name
        endif
    endif
endif
}
```

Por último, existen un tipo especial de operaciones que se ocupan dentro de un mapeo, estas son identificadas como resolve. Mediante una operación tipo resolve es posible acceder a elementos que han sido creados en una reciente transformación, estas son ocupadas aquí para establecer las relaciones entre los elementos de la vista de componentes y conectores. A continuación se ilustra cómo se asigna la característica de role a un conector que enlaza a dos o más componentes.

```
mapping
MOVisMod::Respons::unirptoecon():MVisCoCo::PuertoE
nt
{
    init{
```

```

        var                ptoe                :=
self.resolve(MVisCoCO::PuertoEnt);
        var                come                :=
self.resolve(MVisCoCO::Componen);
        var role := self.resolve(MVisCoCO::Rol)-
>asOrderedSet();
    }
    nombre := 'C_'+nombre;
    penrol := role;
}

```

REFERENCIAS

- Abi-Antoun et al., 2005 Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, Tony Tseng. Improving System Dependability by Enforcing Architectural Intent, ACM SIGSOFT Software Engineering Notes, Proceedings of the 2005 workshop on Architecting dependable systems WADS '05, ACM Press, Volume 30 Issue 4, 2005.
- Abi-Antoun et al., 2005 Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl. Modeling and Implementing Software Architecture with Acme and ArchJava, ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA. ACM 1-58113-963-2/05/0005, 2005.
- OMG QVT, 2005 MOF 2.0 Query/Views/Transformations, Revised Submission. QVT-Merge Group, OMG Document: Version 2.0, ad/03-08-05
- Ali Babar et al., 2004 Muhammad Ali Babar, Liming Zhu, Ross Jeffery. A Framework for Classifying and Comparing Software Architecture Evaluation Methods, IEEE, Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), 2004, pp 309
- Ali Babar, 2004 Muhammad Ali Babar. Scenarios, Quality Attributes, and Patterns: Capturing and Using their Synergistic Relationships for Product Line Architectures, Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004.
- Allen, 1997 Robert J. Allen. A Formal Approach to Software Architecture, PhD Thesis, Carnegie Mellon Univ., CMU Technical Report CMU-CS-97-144, May 1997.
- Araújo et al 2004 Araújo, Jon Whittle, Dae-Kyoo Kim. Modeling and Composing Scenario-Based Requirements with Aspects, 12th IEEE International Requirements Engineering Conference (RE'04) - Volume 00, Pages: 58-67
- Atkinson y Kühne, 2003 Colin Atkinson y Thomas Kühne. Model-Driven Development: A Metamodeling Foundation, IEEE software [0740-7459] vol:20 iss:5, 2003, p. 36
- ATLAS, 2006 ATLAS group LINA & INRIA Nantes, Atlas Transformation Language, ATL User Manual - version 0.7, Feb- 2006.

http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf .(abril-2008)

- | | |
|--------------------------|--|
| Bachmann et al., 2000 | Bachmann, F.; Bass, L.; Chastek, G.; Donohoe, P.& Peruzzi, F. The Architecture Based Design Method.CMU/SEI-2000-TR-001 ADA375851. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000 |
| Bachmann et al., 2003 | Bachmann F., Bass L., Klein M. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design, Technical Report: CMU/SEI-2003-TR-004, Pittsburgh, PA: Software Enginnering Institute, Carnegie Mellon University, 2003. |
| Bachmann et al., 2003-b. | Felix Bachmann, Len Bass, Mark Klein. Preliminary Design of ArchE: A Software Architecture Design Assistant, Technical report CMU/SEI-2003-TR-021, Software Eng. Inst. Carnegie Mellon Univ, 2003 |
| Baldwin y Clark, 2000 | Baldwin, C.,Clark, K. Design Rules: Volume 1. The Power of Modularity. Cambridge, MA: The MIT Press. 2000. |
| Baniassad y Clarke, 2004 | Baniassad, E. y Clarke, S., Theme: An Approach for Aspect-Oriented Analysis and Design, IEEE, 26th International Conference on Software Engineering (ICSE'04), 2004, pp. 158-167 |
| Baragry y Reed, 2001 | Jason Baragry y Karl Reed. Why We Need A Different View of Software Architecture, Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001, pp. 125-134 |
| Barbacci et al., 1998 | Barbacci Mario R., Carriere S. Jeromy, Feiler Peter H., Kazman Rick, Klein Mark H., Lipson Howard F., Longstaff Thomas A.Weinstoc, Charles B. Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis, Carnegie Mellon University Software Engineering Institute Pittsburgh, PA15213-3890., 1998. |
| Bass et al., 2001 | Bass L., Klein M., and Bachmann. "Quality Attribute Primitives and the Attribute Driven Design Method," in 4th International Workshop on Software Product-Family, Engineering, F. van der Linden, Ed. Springer, Berlin Heidelberg, 2001, pp. 169 - 186 |
| Bass et al., 2001 | Len Bass, Mark Klein, Gabriel Moreno. Applicability of General Scenarios to the Architecture Tradeoff Analysis Method, Technical Report,CMU/SEI-2001-TR-014,ESC-TR-2001-014, Carnegie Mellon University. 2001. |

- Bass et al., 2003 Bass L., Clements P., Kazman R. Software architecture in practice, Addison-Wesley, 2ª Ed. , ISBN 0-321-14595-9, 2003
- Bass et al., 2004 Len Bass, Mark Klein and Linda Northrop. Identifying Aspects Using Architectural Reasoning, Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, 2004 pp. 51-57
- Bass et al., 2005 Bass L.B224:B248, Klein M., Bachmann Felix. Quality Attribute Design Primitives and the Attribute Driven Design Method, IEE-Proceedings-Software. Aug. 2005; 152 (4), 2005, pp. 153-165 .
- Bengtsson et al., 2002 Bengtsson PerOlof, Lassing Nico, Bosch Jan, van Vliet Hans, The Journal of Systems and Software 69, Elsevier (2004), 2004, p. 129-147
- Bernardo et al., 2002 Marco Bernardo, Paolo Ciancarini y Lorenzo Donatiello. Architecting families of software systems with process algebras, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11 , Issue 4 (October 2002), 2002, P. 386 - 426
- Beydeda et al., 2005 Sami Beydeda, Matthias Book, Volker Gruhn, Model-Driven Software Development, Springer, 1 edition (September 1, 2005), 354025613X
- Boasson, 1996 Maarten Boasson. The Role of Architecture in the Development of Software Systems, IEEE, Proceedings of the COMPSAC '96 - 20th Computer Software and Applications Conference, 1996, pp. 350-351
- Boehm y Ross, 1989 Boehm B. and Ross R., Theory W Software Project Management: Principles and Examples, IEEE Trans. Software Eng., July 1989, pp. 902-916
- Boer et al., 2004 F.S. de Boer, M. M. Bonsangue, J. Jacob, A. Stam, L.van der Torre. A Logical Viewpoint on Architectures, Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004), 2004
- Booch y IBM, 2007 Grady Booch, IBM. The Irrelevance of Architecture , IEEE, Software, May/June 2007 (Vol. 24, No. 3), 2007, pp. 10-11
- Bosch, 2000 Bosch Jan. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley Professional , 2000
- Boucké et al., 2008 Nelis Boucké, Danny Weyns, Rich Hilliard, Tom Holvoet, and Alexander Helleboogh. Characterizing Relations between Views, por aparecer en ECSA 2008 (articulo aceptado)
- Bratthall y Runeson, 1999 L. Bratthall and P. Runeson, A Taxonomy of Orthogonal Properties of Software Architecture, Proc. Second Nordic Software Architecture Workshop (NOSA '99), 1999.
- Bratthall y Wohlin, 2000 Lars Bratthall y Claes Wohlin. Understanding Some Software Quality Aspects from Architecture and Design Models, IEEE,8th International Workshop on Program Comprehension (IWPC'00), 2000.

- Breemen y Feijs, 2003 A.J.N. van Breemen L.M.G. Feijs. Architecture Evaluation Of An Agent-Based Music Gathering Application, ACM, AAMAS'03, Melbourne, Australia, 2003.
- Brito y Moreira 2004 Isabel Brito and Ana Moreira. Integrating the NFR framework in a RE model, Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, 2004, pages 28-33
- Brown et al., 2005 Alan W. Brown, Jim Conallen, and Dave Tropeano, Model-Driven Software Development, Springer, 1 edition (September 1, 2005), pp. 1-16 354025613X
- Brown et al., 2005 Alan W. Brown, Jim Conallen, Dave Tropeano, Introduction: Models, Modeling, and Model-Driven Architecture (MDA), Model-Driven Software Development, Springer-Verlag Berlin Heidelberg 2005, pp 1-16
- Bruin y Vliet, 2001 Hans de Bruin, Hans van Vliet. Scenario-Based Generation and Evaluation of Software Architectures, Proceedinf of the third Symposium on Genenerative and component-Based Software Engineerng (GCSE'2001), Erfurt Germany, Volume 2186 of Lecture Notes In Comuter Science(LNCS), Berlin Germany, 2001, pp. 128-139,
- Bruin y Vliet, 2002 Bruin, H. de, and Vliet H. van. Top-Down Composition of Software Architectures, Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02), 2002, p. 0147
- Capilla et al., 2006 Rafael Capilla, Francisco Nava, Sandra Pérez, Juan C. Dueñas. A Web-based Tool for Managing Architectural Design Decisions, ACM SIGSOFT Software Engineering Notes Volume 31 , Issue 5, 2006
- Caplat y Sourrouille, 2002 G. Caplat and J.L. Sourrouille. Model mapping in MDA. In J. Bezivin and R. France, editors, Proceedings of UML 2002 Workshop in Software Model Engineering, Dresden, Germany, 2002.
- Caporuscio et al., 2004 Mauro Caporuscio, Paola Inverardi, Patrizio Pelliccione. Compositional Verification of Middleware-Based Software Architecture Descriptions, Proceedings of the 26th International Conference on Software Engineering, 2004.
- Chang et al., 2001 Chang Carl K. , Cleland-Haung Jane, Hua Shiyan, Kuntzmann-Combelles Annie , Function-Class Decomposition: A Hybrid, Software Engineering, IEEE Software, pp. 87-93, Dec. 2001 Method
- Chang y Kim, 2004 Chang Carl K. and Kim Tae-hyung, Distributed Systems Design using Function-Class Decomposition with Aspects, Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004 pp. 148-153

- Cheng et al., 2005 Shang-Wen Cheng, Robert L. Nord, Judith A. Stafford. WICSA Wiki WAN Party: capturing experience in software architecture best practices, ACM SIGSOFT Software Engineering Notes Page 1 January 2005 Volume 30 Number 1, 2005, pp. 1-3
- Clements et al., 2002 Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond, Addison-Wesley Professional; 1st edition (September 26, 2002), isbn: 0201703726, // as tutorial it is 25th International Conference on Software Engineering (ICSE'03), IEEE, 2003.
- Clements et al., 2002-b Clements P., Kasman R., and Klein M., Evaluating Software Architectures : Methods and Case Studies. Addison-Wesley, 2002
- Clements et al., 2002-b Paul Clements, Rick Kazman and Mark Klein, Evaluating Software Architectures: Methods and Case Studies, ADDISON-WESLEY, 12.10.2002
- Courtois, 1985 P.-J. Courtois. On Time and Space Decomposition of Complex Structures, Communication of the ACM, Volume 28 Number 6, pages 590-603, 1985.
- Csert'án et al., 2002 Csert'án, G., G. Huszerl, I. Majzik, Z. Pap, A. Pataricza and D. Varr'ó. VIATRA - visual automated transformations for formal verification and validation of UML models, in: Proc. 17th Int'l Conf. Automated Software Engineering (2002), pp. 267-270.
- Cuesta et al., 2004 Carlos E. Cuesta, M. Pilar Romay, Pablo de la Fuente, Manuel Barrio-Solórzano. Reflection-based, Aspect-Software Architecture, First European Workshop on Software Architecture (EWSA'2004), 2004.
- Cuesta, 2002 Carlos E. Cuesta. Arquitectura de Software Dinámica Basada en Reflexión, Tesis doctoral , Uni. De Valladolid España, 2002.
- Denford et al., 2003 Mark Denford, Tim O'Neill, John Leaney. Architecture-Based Design of Computer Based Systems, Proceedings of the 10 th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), 2005.
- Deursen et al., 2004 Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, Claudio Riva. Symphony: View-Driven Software Architecture Reconstruction, Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04) , 2004.
- Deursen, 2004 Arie van Deursen. *Software Architecture Reconstruction, IEEE, Proceedings of the 26th International Conference on Software Engineering (ICSE'04), 2004, pp.745-746*
- Di Ruscio et al., 2006 Davide Di Ruscio, Henry Muccini, Alfonso Pierantonio, y Patrizio Pelliccione, Towards Weaving Software Architecture Models, Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for irvasive and Embedded

- Software (MBD/MOMPES'06), 2006, pp. 103-112
- Dijkman, 2004 Remco M. Dijkman Dick A.C. Quartel Luís Ferreira Pires Marten J. van Sinderen. A Rigorous Approach to Relate Enterprise and Computational Viewpoints, Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004), 2004, pp.187-200
- Dijkstra, 1979 Programming Considered as a Human Activity. Classic in Software engineering. New York, NY: Yourdon Press, p. 5
- Dobrica y Niemelä, 2002 Liliana Dobrica and Eila Niemelä. A survey on software architecture analysis methods , IEEE Transactions On Software Engineering, Vol. 28, No. 7, 2002
- Duddukuri y Prabhakar, 2005 Rambabu Duddukuri, Prabhakar T.V. On Archiving Architecture Documents, Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), 2005.
- Dueñas y Capilla, 2005 Juan C. Dueñas y Rafael Capilla. The decision view of software architecture, Proceedings of EWSA 2005, LNCS 3527, 2005, pp. 222-230
- Dueñas y Capilla, 2005 Juan C. Dueñas, and Rafael Capilla. The decision view of software architecture, EWSA 2005, LNCS 3527, 2005, pp. 222-230
- Eden y Kazman, 2003 Amnon H. Eden, Rick Kazman. Architecture, Design, Implementation, 7695-1877-X/03 \$17.00 © 2003 IEEE, 2003.
- Egyed et al, 2000 Alexander Egyed, Nikunj Mehta and Nenad Medvidovic. Software Connectors and Refinement in Family Architectures, Proceedings of 3rd International Workshop on Development and Evolution of Software Architectures for Product Families (IWSAPF), Las Palmas de Gran Canaria, Spain, 2000.
- Egyed y Hilliard, 2000 Alexander Egyed, Rich Hilliard. Architectural Integration and Evolution in a Model World, Proceedings of 4th International Software Architecture Workshop (ISAW) co-located with ICSE 2000, Limerick, Ireland, June 2000
- Egyed y Medvidovic, 2000 Alexander Egyed and Nenad Medvidovic. A Formal Approach to Heterogeneous Software Modeling, disponible en <http://citeseer.ist.psu.edu/egyed00formal.html>
- Egyed, 2000 Alexander Franz Egyed. Heterogeneous View Integration and its Automation, Faculty of the Graduate School University of Southern California, 2000.
- Ehrig et al., 2006 H. Ehrig , K. Ehrig , U. Prange , G. Taentzer. Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer science. An EATCS Series), Springer, ISBN-10: 3540311874, 2006.
- Elkharraz et al., 2004 Mili Hafedh, Elkharraz y Mccheick Hamid. Understanding separation of concerns, Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, 2004
- Emery y Hilliard, 2008 David Emery y Rich Hilliard. Updating IEEE 1471:

- architecture frameworks and other topics, in Seventh Working IEEE/IFIP Conference on Software Architecture, pp. 303-306, 2008
- Everitt et al., 2004 Tristan Everitt, Roseanne Tesoriero Tvedt, John D. Tvedt. Compositional Verification of Middleware-Based Software Architecture Descriptions+B247, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 417-421
- Feijs et al., 1998 L. Feijs, R. Krikhaar y R. Van Ommerring. A Relational Approach to Support Software Architecture Analysis, SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(4), 371–400 (10 APRIL 1998)
- Filman et al., 2004 Filman Robert, Elrad Tzilla, Aksit Mehmet, Clarke Siobhan. Aspect-Oriented Software Development, Addison Wesley Professional ISBN 0321-21976-7, 2004
- Finkelstein et al., 1993 A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications, Proceedings of 4th European Software Engineering Conference (ESEC '93), Garmisch-Partenkirchen, Germany, September 1993, LNCS 717, Springer-Verlag, 1993, pp. 84-99
- Fradet et al., 1999 Pascal Fradet, Daniel Le Métayer, Michaël Périn. Consistency checking for multiple view software architectures, Proceedings of the ESEC/FSE-7: ACM SIGSOFT Software Engineering Notes, Volume 24 Issue 6, 1999, pp. 410-428
- France et al., 2004 R. France, I. Ray, G. Georg and S. Ghosh. Aspect-oriented approach to early design modelling, IEE Proc.-Softw., Vol. 151, No. 4, August 2004, pp. 173-185
- Franch y Pere, 1998 Franch Xavier y Botella Pere. Putting Non-Functional Requirements Software Architecture, IEEE, 9th International Workshop on Software Specification & Design, 1998, p. 60
- García, 2003 Milagros García Barbero. La telemedicina como herramienta de formación continuada, experiencias y perspectivas de futuro, Revistaesalud.com , Vol. 2, Número 6 (2006) – II Trimestre, disponible en línea : <http://www.insp.mx/bidimasp/documentos/6/1a%20telemedicina%20como%20herramienta.pdf> , ultimo acceso 21/07/2008
- Garlan et al., 1995 David Garlan, Walter Tichy, Frances Paulisch. Seminar 9508 on Software Architecture in Dagstuhl
- Garlan et al., 2005 David Garlan, William K. Reinholtz, Bradley Schmerl, Nicholas D. Sherman, Tony Tseng. Bridging the Gap between Systems Design and Space Systems Software, Available on line in: <http://www-2.cs.cmu.edu/~able/publications/polaris/> , paper sender to 29th Annual, 2005.
- Garlan y Shaw, 1993 David Garlan y Mary Shaw. An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Vol. I, Word Cientific, 1993, pp. 1-39

- Garlan, 1995 David Garlan. Research Directions in Software Architecture, ACM Computing Surveys, vol 27 no. 2, 1995, p.p. 257-261
- Garzás y Piattini, 2005 Javier Garzás, Mario Piattini. An Ontology for Microarchitectural Design Knowledge, IEEE, Software March/April 2005 Vol. 22, No. 2, 2005, pages. 28-33
- Gašević et al., 2006 Dan Gašević, Dragan Djurić y Vladan Devedžić. Model Driven Architecture and Ontology Development, Springer, 2006, ISBN: 3-540-32180-2
- Gomaa et al., 2001 Gomaa, H., Menascé, Daniel A., Shin Michael E. Reusable Component Interconnection Patterns for Distributed Software Architectures, ACM, Proceedings of the 2001 symposium on Software reusability, 2001, p.p. 69-77
- Gomaa y Farrukh, 1998 H. Gomaa, G. A. Farrukh. Composition of Software Architectures from Reusable Architecture Patterns, Proceedings of the third international workshop on Software architecture, 1998.
- Gomaa y Shin, 2002 Hassan Gomaa y Michael Eonsuk Shin. Multiple-View Meta-Modeling of Software Product Lines, Proceedings of the Eighth IEEE international Conference on Engineering of Complex Computer Systems (ICECCS'02), 2002.
- Gorton, 2006 Gorton Ian. Essential Software Architecture , Springer; 1 edition, ISBN-10: 3540287132, 2006
- Grunske et al., 2005 Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using Graph Transformation for Practical Model-Driven Software, Sami Beydeda, Matthias Book, Volker Gruhn, Model-Driven Software Development, Springer, 1 edition (September 1, 2005) pp. 91-117
- Guo, 2004 Jiang Guo. An Approach for Modeling and Designing Software Architecture, Proceedings of the 10 th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), 2003
- Guoqing et al., 2004 Guoqing Xu , Zongyuan Yang , Haitao Huang. A Basic Model for Components Implementation of Software Architecture, ACM SIGSOFT Volume 29 , Issue 5 , 2004, pp 1-11
- Hagget y Chorley, 1967 Richard J. Chorley, Peter Haggett. Models, paradigms and new geography, in Models in Geography, Methuen, London, 1967
- Hall et al., 2002 Jon G. Hall, Michael Jackson, Robin C., Laney Bashar, Nuseibeh Lucia Rapanotti. Relating Software Requirements and Architectures Using Problem Frames, Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02), 2002. pp. 137.
- Herbert y Ando, 1961 Herbert A. Simon, Albert Ando, Aggregation of Variables in Dynamic Systems, Econometrica, Vol. 29, No. 2 (Apr., 1961), pp. 111-138

Herzum y Sims, 1999	Peter Herzum and Oliver Sims. Business Component Factory, Wiley, ISBN 0471327603, 1999
Hochstein y Lindvall, 2005	Lorin Hochstein, Mikael Lindvall. Combating architectural degeneration: a survey, Information and Software Technology 47 (2005) 643–656, Elsevier B.V., 2005, p.p. 643-657
Hofmeister et al., 1999	C. Hofmeister, R. L. Nord, D. Soni. Describing Software Architecture with UML, Proceedings of the First Working IFIP Conference on Software Architecture. 1999 IFIP. Published by Kluwer Academic Publishers, 1999.
Hofmeister et al., 2005	Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, Pierre America. Generalizing a Model of Software Architecture Design from Five Industrial Approaches, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5). Pittsburgh, PA, November 6-9, 2005. Named one of the five best papers of the conference, 2005.
Hofmeister et al., 2007	Christine Hofmeister, Philippe Kruchten, Robert L. Nord , Henk Obbink ,Alexander Ran , Pierre rica. A general model of software architecture design derived from five, industrial approaches, Journal of Systems and Software,Elsevier, Volume 80, Issue 1 , January 2007, Pages 106-126
Hofmeister, 2005	Christine Hofmeister. Reexamining the Role of Interactions in Software Architecture, Quality of Software Architectures and Software Quality, Volume 3712/2005, LNCS, 2005, pag. 1
Holt, 2006	R. C. Holt. Binary Relational Algebra Applied to Software Architectural, CSRI Technical Report 345, Computer Systems Research Institute, University of Toronto, 2006.
Hopkins, 2000	Hopkins, J. “Component Primer,” Communications of the ACM, Vol. 43, No. 10, 2000, pp. 27-30
Hu y Gorton, 1997	Lei Hu and Ian Gorton. B108A Performance Prototyping Approach to Designing Concurrent Software Architectures, IEEE, 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97),1997, p- 270
Huang, 2007	Gang Huang. Post-Development Software Architecture, ACM SIGSOFT Software Engineering Notes, Volume 32 , Issue 5, 2007, pp. 1-9.
IEEE Std 1471-2000	IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std. 1471, Product No. SH94869-TBR: 2000
Ionita et al,	Mugurel T. Ionita, Dieter K. Hammer, Henk Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview
ISO 9126, 1991	International Standard Organization. Information Technology Software Product Evaluation - Quality Characteristics and Guide lines for their Use, ISO/IEC IS 9126, Geneva, Switzerland, 1991

- ISO/IEC 42010, 2007 ISO. ISO/IEC 42010 Systems and Software Engineering — Architectural Description, 2007
- ISO/IEC DIS 10746-3, 1993 ISO/IEC DIS 10746-3. “Basic Reference Model of Open Distributed Processing - Part 3: ISO/IEC DIS 10746-3, “Basic Reference Model of Open Distributed Processing - Part 3:Prescriptive Model”, February 1994.
- Issarny et al., 1998 Valhie Issarny, Titos Saridakis, Apostolos Zarras. Multi-View Description of Software Architectures, Proceedings of the third international workshop on Software architecture , ACM Press, 1998.
- Ivers et al., 2004 James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, Jaime Rodrigo Oviedo Silva. Documenting Component and Connector Views with UML 2.0, School of Computer Science, Carnegie Mellon University, TECHNICAL REPORT CMU/SEI-2004
- Jazayeri et al., 2000 Jazayeri, M., Ran Alexander, Van der Linden F. Software Architecture for Product Families. Addison-Wesley ISBN-10 0201699672, 2000
- Jinalben et al., 2007 Patel Jinalben, Lee Roger, Kim Haeng-Kon. Architectural View in Software Development Life-Cycle Practices, Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on 11-13 July 2007, pp. 194 - 199
- Jouault y Kurtev, 2006 Frédéric Jouault e Ivan Kurtev. Transforming Models with ATL, Satellite Events at the Models 2005 Conference, LNCS Volume 3844/2006ol. Volume 3844, 2006, pp. 128-138
- Kalfoglou, 2001 Yannis Kalfoglou. Exploring Ontologies, Handbook of Software Engineering and Knowledge Engineering, Vol 1 Fundamentals , ed S.K. Chang, World Scientific Publishing, pp. 863-887
- Kandé y Strohmeier, 2000 Mohamed Mancona Kandé y Alfred Strohmeier. On The Role of Multi-Dimensional Separation of Concerns in Software Architecture. In OOPSLA'2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems (ASoC), October 2000
- Kandé y Strohmeier, 2000 Mohamed Mancona Kandé, Alfred Strohmeier. Towards a UML Profile for Software Architecture Descriptions, Proceedings of UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, 2000
- Kandé, 2003 Mohamed Mancona Kandé. A Concern-Oriented Approach to Software Architecture. PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.
- Kang y Choi, 2005 Sungwon Kang, Yoonseok Choi. Designing Logical Architectures of Software Systems, Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05), IEEE, 2005, pp. 330-337.

- Katara y Katz, 2003 Mika Katara, Shmuel Katz. Architectural views of aspects, Proceedings of the 2nd international conference on Aspect-oriented software development, 2003.
- Kazman et al., 1996 R. Kazman, L. Bass y P. Clements. Scenario Based Analysis of Software Architecture, Software, IEEE Volume 13, Issue 6, Nov 1996 pp.47-55
- Kazman y Carrière, 1998 Rick Kazman, S. Jeromy Carrière. View Extraction and View Fusion in Architectural Understanding, Fifth International Conference on Software Reuse ICSR'98), IEEE, 1998, pp. 290-299
- Keller y Wendt, 2003 Frank Keller, Siegfried Wendt. FMC: An Approach Towards Architecture-Centric System Development, Proceedings of the 10 th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), 2003, pp. 173-182
- Kerth y Cunningham, 1997 Norman L. Kerth, Ward Cunningham. Using Patterns to Improve Our Architectural Vision, IEEE Software, 1997, pp. 417-421
- Khammaci et al., 2006 Tahar Khammaci, Adel Smeda, Mourad Oussalah. Mapping COSA Software Architecture Concepts into UML 2.0, Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS, 2006.
- Kiczales et al., 1997 a Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. XEROX PARC Technical Report, SPL97-008 P9710042. February 1997
- Kiczales et al., 1997 b Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. Aspect Oriented Programing, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp 220-242
- Kim et al., 2004 Feongwook Kim, Jintae Kim, Sooyong Park, Vijayan Sugumaran. A multi-view approach for requirements analysis using goal and scenario, Industrial Management & Data Systems, Volume 104, Number 9, 2004, pp. 702-711(10)
- Kim et al., 2006 Young-Gab Kim, Jin-Woo Kim, Sung-Ook Shin and Doo-Kwon Baik. Managing Variability for Software Product-Line, Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06), IEEE, 2006.
- Kirova y Rossak, 1996 Vassilka Kirova and Wilhelm Rossak. ASPECT - An Architecture Specification Technique A Report on Work in Progress, IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), 1996
- Kishi y Noda, 2001 Tomoji Kishi and Natsuko Noda. Aspect-Oriented Analysis for Architectural Design, Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, pp. 126-129

- Kruchten, 1995 Philippe Kruchten. The 4+1 View Model of Architecture, Paper published in IEEE Software Vol.12, No.6, 1995, pp. 42-50
- Kruchten, 2000 Philippe Kruchten. The Rational Unified Process: An Introduction, Addison-Wesley Longman Publishing Co., Inc. ISBN 0-201-70710-1, 2000
- Kruchten, 2004 P. Kruchten. An ontology of architectural design decisions in software intensive systems, In 2nd Groningen Workshop on Software Variability, 2004, pp. 54–61
- Krüger y Mathew, 2004 Ingolf H. Krüger y Reena Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures, IEEE, Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), 2004, p. 177
- Kyaruzi y Katwijk, 2000 John J. Kyaruzi, Jan van Katwijk. Concerns on Architecture-Centered Software Development: A Survey, Transactions of the Society for Design and Process Science (SDPS), 2000, Vol. 4, No. 3, pp.13-35
- Land, 2002 Rikard Land. A Brief Survey of Software Architecture, Research Center (MRTC) Report, Mälardalen University, Västerås, Sweden, Feb 2002
- Lero et al., 2007 Liam O'Brien Lero, Paulo Merson, Len Bass. Quality Attributes for Service-Oriented Architectures, IEEE Computer Society, Proceedings of the International Workshop on Systems Development in SOA Environments (SDSOA'07), 2007.
- Lesaint y Papamargaritis, 2004 David Lesaint, George Papamargaritis. Aspects and Constraints for Implementing Configurable Product-Line Architectures, IEEE/IFIP Conference on Software Architecture (WICSA'04), 2004, p. 135
- Li et al., 2005 Wanchun Li, Peter Eades, Seok-Hee Hong. Navigating Software Architectures Constant Visual Complexity, the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), 2005, pp. 225-232
- Limon y Ramos, 2006 Rogelio Limon Cordero, Isidro Ramos Salavert, Using Styles to Improve the Architectural Views Design, Proceedings of the International Conference on Software Engineering Advances (ICSEA'06), 2006
- Limon y Ramos, 2006-b Limon C. R., Ramos S.I. The Design of Software Architecture Views Driven by Architectural Styles, 13 Th International Congress on Computer Science Research. Tampico, Mex. Nov. 2006.
- Limon y Ramos, 2007 Limon Cordero R, y Ramos Salavert I. Analyzing Styles of the Modular Software Architecture View, ECSA 2007, LNCS 4758, pp. 275 – 278, 2007
- Lopes y Fiadero, 2003 Lopes A. y Fiadero J.L. Higher-Order Architectural Connectors, ACM Transactions on Software Engineering and Methodology, Vol 12, No. 1 , January 2003, Pages 64-104
- Lung et al, 1997 Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan, Rick Kazman. An approach to software architecture analysis

- for evolution and reusability, ACM, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, p. 15 , 1997
- Manola, 1999 Manola Frank. Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems, Report prepared by Object Services and Consulting, Advanced Information Technology Services Architecture, under contract DASW01 94 C 0054 for the Defense Advanced Research Projects Agency, 1999. <http://www.objs.com/aits/9901-iquos.html#character> . último acceso: oct/2007
- Maranzano et al., 2005 Joseph F. Maranzano, Sandra A. Rozsypal and Gus H. Zimmerman, Guy W. Warnken and Patricia E. Wirth, David M. Weiss. Architecture Reviews:Practice and Experience, IEEE Postmodern software design, March/April 2005 (Vol. 22, No. 2), 2005, pp 34-43
- Marcio et al., 2000 Marcio S. Dias Marlon E. R. Vieira. Software Architecture Analysis based on Statechart Semantics, IEEE Proceedings of the Tenth International Workshop on Software Specification and Design (IWSSD'00), 2000.
- Martínez et al., 2003 Javier Jaén Martínez, Isidro Ramos Salavert. A Conceptual Model for Context-Aware Dynamic Architectures, IEEE, Proceedings of the 23 rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03), 2003
- Matinlassi, 2004 Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods:COPA, FAST, FORM, KobrA and QADA, ACM, Proceedings of the 26th International Conference on Software Engineering, 2004.
- Mattsson et al, 2009 Anders Mattsson, Member, IEEE, Bjoörn Lundell, Member, IEEE, Brian Lings, and Brian Fitzgerald, Linking Model-Driven Development and Software Architecture: A Case StudyLinking, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 35, NO. 1, JANUARY/FEBRUARY,2009,pags. 83-93
- McBride, 2007 McBride Matthew R. The Software Architect, ACM Communications, Vol. 50, Num, 5, pp.75-81, 2007
- Medvidovic et al., 2002 Nenad Medvidovic, David S. Rosenblum y David D. Redmiles. Modeling software architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology (TOSEM), 2002.
- Medvidovic et al., 2003 *Nenad Medvidovic, Marija Mikic-Rakic, Nikunj R. Mehta, Sam Malek. Software Architectural Support for Handheld Computing, IEEE Computer, 2003, pp. 66-73*
- Medvidovic y Taylor, 2000 Nenad Medvidovic y Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 26, NO. 1, JANUARY 2000, pages 70-93
- Medvidovic, 1999 Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor, Rohit Khare, Michael Guntersdorfer. An Architecture-

Centered Approach to Software Environment Integration

Medvidovic, 2002 Nenad Medvidovic. Component-based software engineering: On the role of middleware in architecture-based software development, Proceedings of the 14th international conference on Software engineering and knowledge engineering, 2002.

Medvidovic, 2003 Nenad Medvidovic, Paul Gruenbacher, Alexander F. Egyed Barry, W. Boehm. Bridging Models across the Software Lifecycle, The Journal of Systems and Software, vol. 68, 2003, pp. 199-215.

Mehta et al., Nikunj R. Mehta, Ramakrishna Soma, Nenad Medvidovic. Style-Based Software Architectural Compositions as Domain-Specific Models

Mehta et al., 2000 Mehta N. R., Medvidovic N., Phadke S. Towards a Taxonomy of Software Connectors, ACM, 22nd International Conference on Software Engineering (ICSE'00), 2000, pp 178

Mehta y Medvidovic, Nikunj R. Mehta, Nenad Medvidovic. Distilling Software Architectural Primitives from Architectural Styles, disponible en <http://citeseer.ist.psu.edu/544573.html>

Mehta, 2003 Nikunj R. Mehta. JavaBeans and Software Architecture, Technical Report, 2003.

Mehta, 2004 Nikunj Rohitkumar Mehta. Composing Style-Based Software Architectures From Architectural Primitives, A Dissertation Presented to the Faculty Of The Graduate School University of Southern California In Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy (Computer Science), 2004.

Mellor et al., 2003 Mellor, S.J. Clark, A.N. Futagami, T., Model-driven development , IEEE Software, Volume: 20 Issue: 5, Sept.-Oct. 2003 pp. 14-18

Métayer, 1998 Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars, IEEE Transactions on Software Engineering, 1998, Vol. 24, No. 7, pp. 521-533

Mettala y Graham, 1992 Erik Mettala, Marc H. Graham. The Domain-Specific Software Architecture Program, Technical Report CMU/SEI-92-SR-009, 1992.

Metzger, 2005 Andreas Metzger. A Systematic Look at Model Transformations, Sami Beydeda, Matthias Book, Volker Gruhn, Model-Driven Software Development, Springer, 1 edition (September 1, 2005), 354025613X, pp. 19-33

Miao y Liu, 2005 Huaikou Miao, Jing Liu. Modeling Architecture Based Development in UML, the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), 2005, pp. 56-65

Mikic-Rakic, 2004 Marija Mikic-Rakic. Software Architectural Support for Disconnected Operation in Distributed Environments, A Dissertation Presented to the Faculty of the Graduate School University of Southern California In Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy (Computer Science), 2004.

- Monroe y Garlan, 1996 Robert T. Monroe y David Garlan. Style-Based Reuse for Software Architectures, IEEE Fourth International Conference on Software Reuse, Orlando, FL, 1996, p. 84
- Moreira et al., 2002 A. Moreira, J. Araújo, I. Brito. Crosscutting Quality Attributes for Requirements Engineering, ACM, Proceedings SEKE'02, Vol 7, 2002, pp: 167-174
- Moriconi et al., 1995 Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement, IEEE Transactions on Software Engineering, 1995, Vol. 21, No. 4, pp. 356-3
- Moriconi y Qian, 1994 Mark Moriconi, Xiaolei Qian. Correctness and composition of software architectures, ACM SIGSOFT Software Engineering Notes, Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, 1994.
- Muccini et al., 2004 Henry Muccini, Antonia Bertolino, Paola Inverardi. Using software architecture for code testing, IEEE Transactions on Software Engineering, vol. 30, no. 3, 2004, pp 160-171
- Muskens et al., 2004 Johan Muskens, Michel Chaudron, Christian Lange. Investigations in Applying Metrics to Multi-View Architecture Models, 30th EUROMICRO Conference (EUROMICRO'04), IEEE, 2004, pp. 372-379
- Muskens et al., 2005 J. Muskens, R.J. Bril and M.R.V. Chaudron. Generalizing Consistency Checking between Software Views, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005, pp. 169-180
- Navarro et al., 2003 Elena Navarro, Isidro Ramos & Jennifer Pérez. Software Requirements for Architected Systems, Proceedings of the 11th IEEE International Requirements Engineering Conference, 2003.
- Navasa et al., 2002 A.Navasa, M.A.Pérez, J.M. Murillo. Using an ADL to Design Aspect Oriented Systems1, University of Extremadura. Spain, 2002
- Navasa et al., 2002 A. Navasa, M.A. Pérez, J.M. Murillo, J. Hernández. Aspect Oriented Software Architecture: a Structural Perspective1, In Proceedings of the Aspect-Oriented Software Development, The Netherlands, 2002
- Navasa et al., 2005 Amparo Navasa, Miguel Angel Pérez, and Juan Manuel Murillo, Aspect Modelling at Architecture Design, EWSA 2005, LNCS 3527, pp. 41-58
- Nentwich et al., 2003 C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. ACM Trans. Softw. Eng. Methodol., 12(1):28–63, 2003.
- Nitto y Rosenblum, 1999 Elisabetta Di Nitto, David Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures, Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, May 16–22, 1999, pp 13-22

- Nock, 2003 Clifton Nock. Data Access Patterns: Database Interactions in Object-Oriented Applications, Addison Wesley, ISBN: 0131401572, 2003
- Nord et al., 2000 Robert L. Nord, Daniel J. Paulish, Dilip Soni. Using Software Architecture Estimation
- Nord et al., 2001 Robert L. Nord, Daniel J. Paulish, Robert W. Schwanke, and Dilip Soni. Software Architecture in a Changing World: Developing Design Strategies that Anticipate Change, ACM, ESEC/FSE 2001, Vienna, Austria, 2001, pp. 309, 310
- Noro y Kumazaki, 2002 Masami Noro, Atsushi Kumazaki. On Aspect-Oriented Software Architecture: It Implies a Process as Well as a Product, IEEE, Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSEC'02), 2002, p 276.
- Nuseibeh, 2001 Nuseibeh Bashar. Weaving Together Requirements and Architectures, IEEE, Computer, vol 34, n 3, 2001, pp. 115-119
- Ohlenbusch y Heineman, 1998 Helgo M. Ohlenbusch, George T. Heineman. Composition and interfaces within software architecture, Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, 1998.
- Oliveira y Wermelinger, 2001 Cristov o Oliveira, Michel Wermelinger. Architectural Views for Computation, Coordination and Distribution
- OMG MDA/2003-06-01, 2003 Object Management Group. MDA guide version 1.0.1. Technical Report OMG/2003-06-01, OMG, July 2003
- OMG MOF, 2002 OMG Meta Object Facilit (MOF) Specification v1.4, OMG Document Formal 702-04-03. Disponible en linea: <http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pfd> . Se accedi  el 23 de nov de 2005
- OMG MOF2, 2003 OMG Meta Object Facilit (MOF) Specification 2.0 Core Specification Version 2: Final Adopted Specification Document ptc/03-10-04. Disponible en linea: <http://www.omg.org/cgi-bin/apps/doc?formal/03-10-04.pfd>, Se accedi  el 20 nov de 2007
- OMG QVT RFP, 2002 MOF 2.0 Query/Views/Transformations RFP, OMG Document AD/2002-04-10. <http://www.omg.org/>, 2002.
- OMG QVT2, 2005 MOF QVT Final Adopted Specification
- OMG UML 2.0, 2004 The Object Management Group, Unified Modeling Language: Superstructure, Version 2.0, OMG document formal/05-07-04, 2004
- Oquendo, 2004 Flavio Oquendo. Formally Refining Software Architectures with π -ARL: A Case Study, ACM SIGSOFT Software Engineering Notes Page 1 September 2004 Volume 29 Number 5, 2004.
- Oreizy, 2000 Peyman Oreizy. Open architecture-Software: A Flexible approach to decentralized Software Evolution, University California Irvine, 2000.
- Pack et al., 1997 Pack R.T., Wilkes D. Mitchell and Kawamura K., A software architecture for integrated service robot development, Proceedings Systems, Man, and

- Cybernetics, 1997. 'Computational Cybernetics and Simulation', 1997 IEEE International Conference on. vol.4. pp. 3774-3779
- Pal, 1998 Cris Pal. A Technique for Illustrating Dynamic Component Level Interactions Within a Software Architecture, ACM, Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, 1998.
- Pantoquilha y Moreira, 2004 Marta Pantoquilha, Ana Moreira. Aspect-Oriented Logical Architecture Design A Layered Perspective Applied to Data Warehousing, Aspect-Oriented Logical Architecture Design - A Layered Perspective Applied to Data Warehousing. Desarrollo de Software Orientado a Aspectos (DSOA'2004), Sponsored by AOSD-Europe, in conjunction with JISBD 2004, Malaga, 2004.
- Parnas, 2001 David Parnas. Software fundamentals: collected papers, Addison-Wesley, ISBN 9780201703696 , Apr 01, 2001
- Paulson y Wand, 1992 Paulson Dan y Wand Yair, An Automated Approach to Information Systems Decomposition, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO. 3, MARCH pag. 174-189
- Perez et al., 2003 Jennifer Pérez, Isidro Ramos, Javier Jaén, Patricio Letelier. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures, IEEE, Proceedings of the Third International Conference On Quality Software (QSIC'03), 2003
- Perez, 2006 J. Perez. PRISMA. PhD Departamento de Sistemas y Computación, Universidad Politecnica de Valencia, España, 2006.
- PerOlof et al., 2002 Bengtsson PerOlof, Lassing Nico, Bosch Jan, van Vliet Hans. Architecture-level modifiability analysis (ALMA), The Journal of Systems and Software 69, Elsevier (2004) pag. 129-1472
- Perry y Wolf, 1992 Dewayne E. Perry y Alexander L. Wolf. Foundations for the Study of Software Architecture, ACM SIGSOFT SOFTWARE ENGINEERING NOTES vol 17 no 4, 1992, pp. 40-52
- Perry y Wolf, 1995 D. E. Perry & A. L. Wolf. The "4+1" View Model, ACM Software Foundations for the Study of Software Architecture, Engineering Notes, 1995, p17, 4,40-52
- Poulin, 1994 Jeffrey S. Poulin, Norm Kemerer, Mike Freeman, Tim Becker, Kathy Begbie, Cheryl D'Allesandro, Chuck Makarsky. A Reuse-Based Software Architecture for Management Information Systems, IEEE Fourth International Conference on Software Reuse, Orlando, FL, April 1998, p 94
- Purhonen et al., 2004 Anu Purhonen, Eila Niemelca, Mari Matinlassi. Viewpoints of DSP software and service architectures, Journal of Systems and Software Volume 69, Issues 1-2, 2004, pp. 57-73
- Rakic y Medvidovic, 2001 Marija Rakic and Nenad Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software

Connector-Based Approach, ACM SIGSOFT Software Engineering Notes , Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, Volume 26 Issue 3, 2001, p. 11-18

Rapanotti et al., 2004 Lucia Rapanotti, Jon G. Hall, Michael Jackson, Bashar Nuseibeh. Architecture-driven Problem Decomposition, 12th IEEE International Requirements Engineering Conference (RE'04), 2004

Rashid et al., 2002 Awais Rashid, Peter Sawyer, Ana Moreira and João Araújo. Early Aspects: a Model for Aspect-Oriented Requirements Engineering, IEEE Joint International Conference on Requirements Engineering (RE'02), 2002, pp. 199

Rashid et al., 2003 Awais Rashid, Ana Moreira, João Araújo. Modularisation and composition of aspectual requirements, Proceedings of the 2nd international conference on Aspect-oriented software development, ACM pages 11-20, 2003

Remco, 2003 Remco M. Dijkman Dick A.C. Quartel Luís Ferreira Pires Marten J. van Sinderen. An Approach to Relate Viewpoints and Modeling Languages, Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), 2003. P. 14

Reza y Grant, 2004 Hassan Reza and Emanuel Grant. Model Oriented Software Architecture, IEEE, 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts (COMPSAC'04), 2004, pp. 4-5

Reza, 2003 Hassan Reza. Pattern-Based Software Architecture: A Case Study

Rosch, 2007 Peter Rosch. User Interaction in a Multi-View Design Environment, Visual Languages. Proceedings., IEEE Symposium on 3-6 Sept. 1996, pp. 316 - 323

Rosenblum y Natarajan, 2000 D.S. Rosenblum y R. Natarajan. Supporting architectural concerns in component-Interoperability standards, IEEE Proceedings-Software, 2000, p. 147(6).

Roshandel et al., 2004 Roshanak Roshandel, Bradley Schmerl, Nenad Medvidovic, David Garlan, Dehua Zhang. Understanding Tradeoffs among Different Architectural Modeling Approaches, Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), 2004, pp. 47- 56

Roshandel et al., 2004 Roshanak Roshandel, Andr'E Van Der Hoek, Marija Mikic-Rakic y Nenad Medvidovic. Mae---a system model and environment for managing architectural evolution, ACM, TOSEM, Volume 13, Issue 2, 2004, Pages: 240 - 276

Roshandel y Medvidovic, Rumbaugh et al., 2005 b Roshanak Roshandel and Nenad Medvidovic. Multi-View Software Component Modeling for Dependability Rumbaugh, Jacobson, and Booch, UML Process, Addison Wesley, 2005

Ryoo y Saiedian, 2006 Jungwoo Ryoo a, Hossein Saiedian. AVDL: A highly adaptable architecture view description language, Journal

of Systems and Software Elsevier 79 (8), 2006, p.p. 1180-1206

- Sagardui et al., 2005 Gouiuria Sagardui, Gentzane Aldekoa, Leire Etxeberria. The ADOV Method: an Experience in Selecting the Relevant Views of an Architecture in a SME, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005.
- Sangal et al., 2005 Sangal N., Jordan, E. Sinha, V., Jackson D. Using Dependency Models to Manage Complex Software Architecture, Proceedings of the 20th annual ACM SIGPLAN, 2005, Pages: 167 – 176
- Sarkar y Thonse, 2004 Santonu Sarkar, Srinivas Thonse. EAML- Architecture Modeling Language for Enterprise Applications, Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-East'04), 2004, pp 40-47
- Savolainen y Tuomo, 2002 Juha Savolainen, Tuomo Vehkomäki. An Integrated Model for Requirements Structuring and Architecture Design, 7th Australian Workshop on Requirements Engineering (AWRE'2002) , 2002
- Savolainen, 2003 Juha Savolainen. The Role of Ontology in Software Architecture, A Position Paper for OOPSLA Workshop on How to Use Ontologies and Modularization to Explicitly Describe the Concept Model of a Software Systems Architecture, 2003.
- Schmerl y Garlan, 2004 Bradley Schmerl and David Garlan. AcmeStudio: Supporting Style-Centered Architecture Development, In Proc. 2004 International Conference on Software Engineering, Edinburgh, Scotland, May 2004.
- Schmidt, 2001 Douglas C. Schmidt
David A. Schmidt. David A. Schmidt. Should UML Be Used for Declarative Programming?, Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming, 2001.
- Scratchley, 2000 W. Craig Scratchley. Evaluation and Diagnosis of Concurrency Architectures, PhD Thesis, Department of Systems and Computer Engineering Carleton University, Jul 2000
- Seidewitz, 2003 Ed Seidewitz. What models mean, IEEE Software Volume: 20 Issue: 5 Date: Sept.-Oct. 2003, pags. 26- 32
- Selic, 2003 Selic, B., The pragmatics of model-driven development, IEEE Software Volume: 20 Issue: 5 Date: Sept.-Oct. 2003, pags. 19- 25
- Sendall y Kozaczynski, 2003 Sendall, S., Kozaczynski, W., Model transformation: the heart and soul of model-driven software development, IEEE Software, Volume: 20 Issue: 5 Date: Sept.-Oct. 2003, pp. 42- 45
- Shaw y Garlan, 1996 Shaw Mary y Garlan David. Software Architecture Perspectives on an Emerging Discipline, Prentice Hall, ISBN 8120314700, 1996

- Shaw et al., 1995 Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, Daniel V. Klein, Theodore L. Ross, Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, vol.21, No. April. 1995, pp. 314-335
- Shaw y Clements, 2006 Mary Shaw y Paul Clements. The Golden Age of Software Architecture, IEEE Software, Volume: 23 Issue: 1, 2006, pp 31-39
- Shaw, 1984 Mary Shaw. Abstraction Techniques in Modern Programming Languages, IEEE Software vol. 1(4), p. 10
- Shaw, 1989 Shaw. Larger Scale Systems Require Higher-Level Abstractions
- Shaw, 1990 Mary Shaw , Toward higher-level abstractions for software systems, Data & Knowledge Engineering, Volume 5 , Issue 2 (July 1990), North Holland: Elsevier Science Publishers B.V. 1990, pp. 119-128
- Shaw, 2001 Mary Shaw. The Coming-of-Age of Software Architecture Research, Proc.. 23rd Int'l Conf. Software Eng. IEEE CS Press, 2001, pp. 656-664a.
- Shaw, 2003 M. Shaw. Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status, Workshop on Studies of Software Design, 1993.
- Shi, 2003 Tianjun Shi, Xudong He. A Methodology for Dependability and Performability Analysis in SAM, International Conference on Dependable Systems and Networks (DSN'03). IEEE, 2003, p. 679-688
- Smolander, 2003 Smolander, K. On the Role of Architecture in Systems Development. PhD Thesis, Lappeenranta University of Technology, 2003
- Soni et al., 1995 Soni, D., Nord, R.L., and Hofmeister, C. Software Architecture in Industrial Applications, in Proceedings of the 17th International Conference on Software Engineering, Seattle, WA, 1995
- Sousa et al., 2004 Geórgia Sousa, Sérgio Soares, Paulo Borba and Jaelson Castro. Separation of Crosscutting Concerns from Requirements to Design: Adapting an Use Case Driven Approach, Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, 2004 pages. 96-107
- Stahl et al., 2006 Stahl Thomas, Völter Markus, Bettin Jorn, Haase Arno, Helsen Simon. Model-Driven Software Development, John Wiley & Sons, 2006
- Steen et al., 2004 M.W.A. Steen, D.H. Akehurst, H.W.L. ter Doest, M.M. Lankhorst. Supporting Viewpoint-Oriented Enterprise Architecture, Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004), 2004, pp. 201-211
- Stevenson et al., 2008 Duncan Stevenson, Jane Li, Jocelyn Smith and Matthew Hutchins. A Collaborative Guidance Case Study, Proc. 9th Australasian User Interface Conference (AUI2008), Wollongong, Australia, January

- 2008., pags. 33-42
- Stoermer et al., 2004 Christoph Stoermer, Liam O'Brien, Chris Verhoef. Architectural Views through Collapsing Strategies, 12th IEEE International Workshop on Program Comprehension (IWPC'04), 2004
- Stoermer et al., 2006 Stoermer C, Rowe A, O'Brien L, Verhoef C. Alborz: An Interactive Toolkit to Extract Static and Dynamic Views of a Software System, Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), 0-7695-2601-2/06 2006
- Stoermer et al., 2006 Christoph Stoermer, Anthony Rowe, Liam O'Brien and Chris Verhoef. Model-centric software architecture reconstruction, SOFTWARE-PRACTICE & EXPERIENCE 36 (4): 333-363 APR 10 2006, Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.699, 2006.
- Storey y Muller, 1995 M.-A.D. Storey, H.A. Muller. Manipulating and documenting software structures using SHrIMP views, Proceedings of 11th International Conference on Software Maintenance (ICSM'95)IEEE, 1995, p. 275,
- Subramanian y Chung, 2003 Nary Subramanian, Lawrence Chung. Process-Oriented Metrics for Software Architecture Evolvability, Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03), 2003, pp 65.
- Sutton Jr y Rouvellou, 2000 Sutton Jr., S. M. and Rouvellou I. , Concerns in the Design of a Software Cache In Workshop on Advanced Separation of Concerns (OOPSLA), Mineapolis, 2000 http://www.research.ibm.com/AEM/pubs/caching_oopsla2000.pdf, último acceso: 19/10/2007
- Svahnberg, 2002 Mikael Svahnberg, Claes Wohlin, Lars Lundberg, Michael Mattsson. A Method for Understanding Quality Attributes in Software Architecture Structures, ACM International Conference Proceeding Series; Vol. 27, Pages: 819 - 826, Proceedings of the 14th international conference on Software engineering and knowledge engineering, 2002
- Swapna et al., 2002 Swapna S., Gokhale Kishor, S. Trivedi. Reliability Prediction and Sensitivity Analysis Based on Software Architecture, IEEE, Proceedings of the 13 th International Symposium on Software Reliability Engineering (ISSRE'02), 2002, pp 64.
- Tarr et al., 1999 P.Tarr, H. Ossher, W. Harrison. and S.M. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns", Proc. of the 21st International Conference on Software Engineering, Los Angeles, CA, USA, May 1999, pp. 107-119.
- Tekinerdoğan et al., 2004 Bedir Tekinerdoğan, Ana Moreira, João Araújo and Paul Clements, Workshop Report on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture

- Design Workshop, Lancaster, 2004
- Tekinerdogan, 2000 Bedir Tekinerdogan. Synthesis-Based Software Architecture Design, PhD thesis University of Twente, Enschede, The Netherlands. - With index, -ref. With summary in Dutch. ISBN 90-365-1430-4, 2000.
- Tekinerdogan, 2004 Bedir Tekinerdogan. ASAAM: Aspectual Software Architecture Analysis Method, IEEE/IFIP Conference on Software Architecture, Oslo, 2004, p 5
- Thompson, 1998 C. Thompson (ed.), OMG-DARPA Workshop on Compositional Software Architectures, Monterey, California, January 6-8, 1998, Workshop Report, February 15, 1998, <http://www.objs.com/workshops/ws9801/report.html>.
- Tracz, 1995 Will Tracz. Domain-Specific Software Architecture (DSSA) Pedagogical Example, ACM SIGSOFT Software Engineering Notes vol 20 no 3, 1995, p. 49-62
- Tsantalis et al., 2006 Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides. Design Pattern Detection Using Similarity Scoring, IEEE Transactions on Software Engineering, 2006, Vol. 32, No. 11, pp. 896-909.
- Tu y Godfrey, 2001 Qiang Tu and Michael W. Godfrey. The Build-Time Software Architecture View, 17th IEEE International Conference on Software Maintenance (ICSM'01), 2001, pp. 398-407
- Tyree y Akeman 2005 Jeff Tyree and Art Akerman, Architecture Decisions: Demystifying Architecture, IEEE Postmodern Software Design, March/April 2005 (Vol. 22, No. 2), 2005, pp 19-27
- Tyson, 1998 K. Tyson (ed.). "Reference Model Extension Green Paper", Object Management Group, Object and Reference Model Subcommittee of the Architecture Board, OMG Document
- Videira y Bajracharya, 2005 Cristina Videira Lopes and Sushil Krishna Bajracharya. An Analysis Of Modularity In Aspect Oriented, ACM ,Proceedings of the 4th international conference on Aspect-oriented software development Design, 2005 pages 15-26
- Wallnau et al., 2001 Kurt Wallnau, Judith Stafford, Scott Hissam, Mark Klein. On the Relationship of Software Architecture to Software Component Technology, In Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP6), in conjunction with the European Conference on Object Oriented Programming (ECOOP), Budapest, Hungary, 2001.
- Walter y Dos Santos, 2006 Walter A. Dos Santos. An MDA Approach for a Multi-Layered Satellite On-Board Software Architecture, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2006
- Wang et al., 2004 Guijun Wang, Casey K. Fung. Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems, IEEE, 37th Hawaii International

- Conference on System Sciences - 2004, pp. 90272a
- Westfechtel y Conradi, 2004 Bernhard Westfechtel y Reidar Conradi . Software Architecture and Software Configuration Management, Norwegian University of Science and Technology, Advances in Software Engineering and Knowledge Engineering. Volume 2., 2004.
- Wieringa, 1998 R. Wieringa. "A Survey of Structured and Object-Oriented Specification Methods and Techniques," ACM Computing Surveys, Dec. 1998, pp. 459-527.
- Williams y Smith, 2002 Williams L. G., Smith C. U. PASASM: A Method for the Performance Assessment of Software Architectures, ACM, the third international workshop on Software and performance , SESSION: Performance evaluation of software architecture Pages: 179 - 189 Year of Publication: 2002 ISBN:1-58113-563-7, 2002.
- Woods y Rozansk, 2005 Eoin Woods y Nick Rozansk. Using Architectural Perspectives, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005.
- Woodside, 2001 C. M. Woodside. *Software Resource Architecture and Performance Evaluation of Software Architectures, Proceedings of the 34th Hawaii International Conference on System Sciences, 2001.*
- Xu et al., 2004 Guoqing Xu, Zongyuan Yang, Haitao Huang. JCMP: Linking Architecture with Component Building, Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004, pp 292-299
- Xu et al., 2005 Lihua Xu Hadar Ziv Debra Richardson. An Architectural Pattern for Non-functional Dependability Requirements, ACM SIGSOFT Software Engineering Notes , Proceedings of the 2005 workshop on Architecting dependable systems WADS '05, Volume 30 Issue 4, 2005, pp.1-6
- Yahiaoui Nesrine Yahiaoui, Bruno Traverson, Nicole Levy. Adaptation management in multi-view systems, disponible en http://wcat05.unex.es/Documents/Yahiaoui_et_al.pdf
- Yeh et al., 1997 Alexander S. Yeh, MA, David R. Harris, Melissa P. Chase. Manipulating Recovered Software Architecture Views , 19th International Conference on Software Engineering (ICSE'97) IEEE, 1997, pp. 184-194
- You-Sheng, 2003 Zhang You-Sheng He Yu-Yun. Architecture-Based Software Process Model, ACM SIGSOFT Software Engineering Notes vol 28 no 2 March 2003, P. 1-4
- Yujian et al., 2006 Yujian Fu, Zhijiang Dong, Xudong He. A method for realizing software architecture design, Proceedings of the 6th International Conference on Quality Software 2006
- Zdun y Avgeriou, 2005 Uwe Zdun y Paris Avgeriou, Modeling Architectural Patterns Using Architectural Primitives, OOPSLA'05, 2005, pags. 133-146
- Zhu et al., 2005 Liming Zhu, Aybü Ke Aurum, Ian Gorton, Ross Jeffery.

Tradeoff and Sensitivity Analysis in Software
Architecture Evaluation Using Analytic Hierarchy
Process, Springer Science Business Software Quality
Journal 13, 2005, pp.357-375