Tesis de Ingeniería

Un modelo integrado de dependencias

Documentación de arquitectura más allá de las vistas

Facultad de Ingeniería

Universidad de Buenos Aires

Alumno: Diego Fontdevila

Tutor: Lic. Sergio Villagra

Cotutora: Lic. Andrea Morales

Indice

I.Organización del documento..5

II.Introducción..7

III.Objetivo de la Tesis..11

1.Alternativas a la solución propuesta..12

IV.Problemas de documentación integradora..15

1.Una primera aproximación...15

V.Documentación basada en modelos...19

1.Especificación vs. Descripción...19

2.El problema del acuerdo entre el modelo y el sistema......................20

VI.Diagramas de dependencia..23

VII.Documentación de arquitectura en la práctica.....................................25

1.Reglas de buena documentación..27

VIII.IDM: Una herramienta de control de dependencias.............................33

1.Cómo se usa IDM..35

2.Escenarios de uso de IDM...37

3.Evaluación del sistema en estudio en base al modelo......................39

4.Análisis de modelos basado en metadatos..40

5.Historia de una implementación...41

5.1.Implementación de IDM Núcleo...43

5.2.Implementación de IDM Vista..44

6.Arquitectura de IDM..47

6.1.Vista de Casos de Uso..47

6.2.Vista lógica...48

6.2.1.Modelo de solución de IDM..48
6.2.2.Estructura de subsistemas de IDM..50
6.2.3.IDM Núcleo...51
6.2.4.IDM Vista...55
6.3.Vista de Componentes...57

IX.Conclusiones y perspectivas...59

X.Agradecimientos..61

XI.Anexo I: Ejemplo de modelo integrado de dependencias.......................63

XII.Anexo II: Ejemplo de modelo para una aplicación real.........................65

XIII.Indice de Figuras...67

XIV.Glosario...69

XV.Referencias...71

XVI.Bibliografía...73

I. Organización del documento

Este documento está estructurado en dos secciones principales bien diferenciadas. La primera parte presenta la problemática y sienta las bases teóricas para la solución propuesta, mientras que la segunda parte presenta la herramienta implementada como prueba de concepto para las ideas planteadas.

La primera va desde el capítulo I hasta al VII y presenta la problemática central del trabajo y la solución propuesta. El capítulo II introduce el problema del que se ocupa el trabajo, el capítulo III presenta los objetivos de la Tesis, el capítulo IV profundiza en los problemas de documentación integradora, el capítulo V sienta las bases para el uso de modelos en la evaluación de un sistema en desarrollo, el capítulo VI describe el uso de diagramas de dependencias en los modelos, y el capítulo VII profundiza sobre la práctica de la documentación.

La segunda parte incluye los capítulos VIII y IX, en el primero se presenta la herramienta y el proceso llevado a cabo para su desarrollo, y en el siguiente se presentan las conclusiones y alternativas de trabajo a futuro.

En todo el trabajo las citas están realizadas en el idioma de original de la bibliografía. Sin embargo, se presenta nuestra traducción para cada una a fin de aclarar nuestra interpretación.

II. Introducción

En los distintos tipos de proceso de desarrollo de software, las divisiones de tareas y roles marcan profundamente la estructura del producto resultante. Así, las tareas de análisis, diseño, construcción, implantación, etc., definen diferentes perspectivas sobre el sistema, se asocian en general con personas distintas y se llevan a cabo en momentos diferentes del proceso de desarrollo.

Este concepto es muy utilizado para organizar los modelos y las tareas de un proyecto de software y se conoce como vista (*view*): Una representación parcial de los elementos del sistema y sus relaciones asociadas ([Clements 2003], pág. 13). La división en vistas es muy importante ya que afecta especialmente la organización de los modelos, la definición del vocabulario de dominio del problema y la especificación de los requerimientos, entre otros.

Un ejemplo muy común de organización en vistas es la que incluye el UML (*Unified Modeling Language*), originada en el trabajo de Philippe Kruchten "*The "4+1" View Model of Software Architecture*" (Las cuatro vistas son: Lógica, Desarrollo o Implementación, Física o Distribución y Proceso, la vista +1 es la vista de Escenarios, Casos de uso, o Análisis, ver [Kruchten 1995]), pero otros autores han propuesto otras clasificaciones de vistas para describir un sistema, por ejemplo Soni, Nord y Hofmeister, de Siemens (citados por [Clements 2003], pág. 17), proponen las siguientes vistas: Conceptual, Interconexión de módulos, Ejecución y Código.

Aunque asumir la perspectiva de una vista simplifica la comprensión del sistema, existen relaciones entre elementos de vistas distintas que muchas veces quedan fuera de toda perspectiva. Esa información refleja muchas veces aspectos importantes del sistema, es difícil de documentar y tiende a estar desactualizada, porque hay que trabajar doble, sobre el sistema y sobre la documentación.

En las metodologías basadas en documentación, como el RUP (*Rational Unified Process*), el intercambio de información entre los participantes está definido por el modelo de proceso de desarrollo, de manera tal que cada rol tiene relación más cercana con alguna parte de la documentación. Por ejemplo, el arquitecto mantiene una visión de alto nivel de las diferentes estructuras del sistema, el diseñador una visión más centrada en la estructura lógica del sistema, y el

programador una basada en el código fuente y los archivos de configuración. Sin embargo, a la hora de comprender el sistema, desde el punto de vista del programador, no es suficiente el código. En general, es necesario más que sólo una porción de la información para obtener una comprensión del sistema. Por ejemplo, cuando estudiamos la estructura de clases de una aplicación, la información sobre el flujo de control parece fácilmente extraíble del código; sin embargo, la relación de esas clases con otros elementos del sistema no es tan clara, y puede ser muy significativa a la hora de comprender el comportamiento global del mismo.

Esta característica se acentúa con el aumento de tamaño de los módulos componentes de los sistemas (Clements y Shaw [Clements 1996], presentan esta característica como uno de los tres patrones que explican la historia de la ingeniería de software) y su distribución, que tienen como consecuencia la proliferación de *frameworks* y software de base independiente (como las bases de datos, las herramientas de desarrollo de aplicaciones web o los servidores de aplicación Java EE). En esos casos, el código de la aplicación interactúa cada vez más con componentes y elementos predefinidos, los cuales no se definen en el mismo momento ni participan del mismo ciclo de desarrollo que los componentes de aplicación, si no más bien se integran como parte del proceso de instalación o despliegue (*deployment*).

Por ejemplo, si consideramos un aplicación web java típica, nos encontramos con que el estándar JEE (*Java Enterprise Edition*) define que el flujo de control de una aplicación es controlado por *servlets* (clases java capaces de recibir y responder pedidos realizados a un servidor web). Sin embargo, a la hora de aplicar esta definición, en la práctica no se utiliza nunca código Java para definir este flujo, si no que se configura en archivos externos no compilables. El ejemplo más claro en ese ámbito y el estándar de facto hasta hace pocos años (hasta la definición de JSF, *Java Server Faces*, como parte del estándar JEE) es el *framework Struts*, que esencialmente implementa un marco para control de flujo de aplicaciones web configurado sobre XML. En *Struts*, los desarrolladores crean sus propias clases que implementan la lógica de la aplicación, pero el control de la navegación de la aplicación se define en la configuración de *Struts* (struts-config.xml). En este caso, para saber si una porción de código se ejecuta en ciertas condiciones, se mira primero la configuración en xml y a partir de ella, la clase correspondiente. En otras palabras, no alcanza el compilador ni el código

para comprender el comportamiento del sistema.

Esto tiene como resultado la disminución de la información que puede manejarse con claridad desde el código, y el aumento de la necesidad de utilizar otras herramientas para poder desarrollar en forma eficiente. Esto incluye herramientas de instalación (*deployment*), prueba, edición gráfica de archivos descriptores y de configuración en general (como el caso de la vista de estados de MyEclipse para struts-config.xml), bases de datos y *web services* entre otros.

Para un ejemplo independiente de la web, basta considerar las bases de datos relacionales, para las cuales no contamos con soporte nativo en los lenguajes de propósito general (Microsoft .NET ha incorporado en los últimos años alguna forma de integración de código SQL a sus lenguajes orientados a objetos, pero manteniendo la heterogeneidad de las expresiones y sin unificar el lenguaje , ver http://msdn2.microsoft.com/es-es/library/ms254963(VS.80).aspx), y necesitamos todavía hoy incluir código SQL en el código de aplicación (aunque en el mejor de los casos esté oculto en un transformador objeto relacional u *object-relational mapper*). Las mismas consideraciones valen para el acceso remoto a objetos. En la mayoría de los casos, el acceso remoto o distribuido a objetos sigue siendo no orientado a objetos, es decir sin soporte de polimorfismo (para una clara comparación entre sistemas distribuidos no orientados a objetos, basados en objetos y orientados a objetos, ver [SUN 2003]). Basta considerar la tan moderna forma de integración de servicios con *web services* (que no es orientada a objetos) y el soporte que esta tecnología tiene en los lenguajes estándar de desarrollo que fuerza a utilizar librerías de terceros o herramientas específicas (ver el *Java Enterprise Edition Tutorial* en http://java.sun.com/javaee/5/docs/tutorial/doc/ y la herramienta Apache Axis en http://ws.apache.org/axis/).

En todos estos casos, cuesta comprender un elemento en forma independiente de otros elementos, y cuesta apreciar sus relaciones, porque corresponden a modelos distintos y las herramientas para manejarlas no están estandarizadas en los lenguajes de programación. Podríamos decir que, para esas arquitecturas, no es fácil cumplir el Criterio de Comprensibilidad Modular como lo enuncia Bertrand Meyer en *Object-Oriented Software Construction*:

> *"A method favors Modular Understandability if it helps produce*

software in which a ***human reader can understand each module without having to know the others****, or, at worst, by having to examine only a few of the others."*

"Un método favorece la Comprensibilidad Modular si ayuda a producir software en el cual un ***lector humano puede entender cada módulo sin tener que conocer los otros****, o, en el peor de los casos, teniendo que examinar sólo unos pocos de los otros."*

[Meyer 1985, pág. 43, el resaltado es nuestro. Método significa metodología de desarrollo en este contexto]

Es decir, no es fácil comprender un componente sin tener en cuenta y conocer varios otros componentes, y este proceso se hace más difícil en la medida en que componentes muy dispares aparecen muy relacionados.

En el capítulo siguiente describimos los objetivos de la tesis en base a la problemática planteada.

III. Objetivo de la Tesis

Como vimos en el capítulo anterior, la problemática de describir y comprender módulos pertenecientes a vistas distintas es compleja y difícil de tratar. Nuestra propuesta consiste en utilizar un modelo integrado de dependencias para describir dichas relaciones. Usaremos diagramas de dependencia para relacionar esos elementos y evaluaremos esa información para decidir si se corresponde con el sistema construido.

El ejemplo clásico de dependencia entre elementos de distinta vista es la trazabilidad de requerimientos o *traceability*, descrita en el UML como una dependencia de estereotipo *trace (*para la definición ver [Booch 1999], página 468). La misma describe una relación histórica entre dos elementos, en particular un requerimiento y los paquetes cuya responsabilidad es proveer la funcionalidad requerida. Esta dependencia es interesante porque es una de las pocas que el UML define para relacionar elementos de vistas distintas (ver [Booch 1999], página 423).

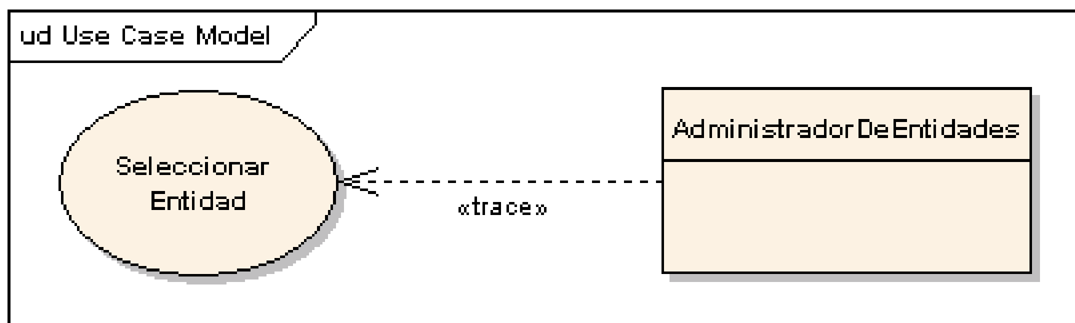

Figura 1: Ejemplo de trazabilidad, una clase que participa de la Implementación de un caso de uso

En una perspectiva más moderna y afín a la de nuestro trabajo, en ***Documenting Software Architectures, Views and Beyond***, el equipo de arquitectura del SEI propone encarar el tema con una visión más integradora, que se percibe desde el título, **vistas y más allá**, y se plasma desde el principio de su propuesta:

> *"Documenting software architecture is a matter of documenting the relevant views and then adding **documentation that applies to more than one view**."*

> *“Documentar la arquitectura de software es cuestión de documentar las vistas relevantes y luego agregar* ***documentación que aplica a más de una vista.****”*
>
> *([Clements 2003], pág. 13, el resaltado es nuestro).*

Consideramos este estudio de las dependencias significativo no sólo por la importancia de una buena documentación, si no porque ésta afecta el **análisis de impacto**, es decir el análisis de las estructuras y componentes de un sistema que se ven afectados por un cambio. El análisis de impacto es interesante porque su resultado depende de la estructura interna del sistema y de la flexibilidad del diseño realizado. Como plantean Clements et al.:

> *“Impact analysis requires a certain degree of design completeness and integrity of the module description. In particular,* ***dependency information has to be available and correct*** *in order to create good results.”*
>
> *“El análisis de impacto requiere cierto grado de completitud del diseño e integridad de la descripción de los módulos. En particular,* ***la información de dependencias tiene que estar disponible y ser correcta*** *para crear buenos resultados”*
>
> *([Clements 2003], pág. 47, el resaltado es nuestro).*

Además, este tipo de análisis es la base de la definición del costo de un requerimiento de modificación, y en gran medida del costo de mantenimiento, que no son en general menores en el ciclo de vida completo de un sistema.

A partir de estas ideas, avanzamos en el estudio de las dependencias entre elementos de distintas vistas. En la búsqueda de una aplicación práctica para las mismas, profundizamos en el uso de modelos para la descripción de las dependencias y su documentación, y presentamos como prueba de concepto una herramienta para la evaluación automática de diagramas y el control de las dependencias documentadas.

1. Alternativas a la solución propuesta

Una solución alternativa a nuestra propuesta basada en la documentación de las dependencias es la de unificar en el lenguaje de programación el soporte para

describir todas estas relaciones (planteada por David Garland en la edición de 2006 de la JIDIS, Jornada de Investigación y Desarrollo de Ingeniería de Software, en Buenos Aires). Bertrand Meyer define una idea análoga para los módulos como el "Principio de Unidades Lingüísticas Modulares" (*Linguistic Modular Units Principle*), que se enuncia así:

> *"Modules must correspond to syntactic units in the language used."*
>
> *"Los módulos deben corresponder a unidades sintácticas en el lenguaje utilizado."*
>
> *([Meyer 1985], pág. 53)*

La ventaja sería contar con un lenguaje capaz de expresar mucho mejor estas arquitecturas que son hoy tan comunes. La dificultad inmediata está en la definición de ese lenguaje y la complejidad de lograr múltiples implementaciones del mismo (actualmente las herramientas de desarrollo para estas arquitecturas tienen en general una parte definida de acuerdo al estándar y una parte específica propia del proveedor del producto, lo cual agrega un elemento más de variabilidad y complejidad).

El soporte de anotaciones en Java y C# permite agregar metadatos que exceden el lenguaje y pueden describir características de arquitectura. Por ejemplo, en la especificación del estándar EJB 3.0, Enterprise Java Beans, de JEE, las anotaciones permiten describir comportamiento transaccional, publicación de un componente como servicio web, dependencias entre un componente de lógica de negocio y un recurso compartido, por ejemplo un servidor de mail.

Otro ejemplo de extensión de los lenguajes en esta dirección es AspectJ, implementación original de AOP (Aspect Oriented Programming) que permite expresar aspectos transversales a múltiples módulos mediante una sintaxis específica (en la sección de perspectivas trataremos una aplicación de AOP a nuestra herramienta).

En una perspectiva completamente diferente, la de infraestructura en el modelo ITIL (*Information Technology Infrastructure Library*, ver http://www.itil.co.uk/), la administración de los cambios en la perspectiva de mantenimiento requiere un control muy detallado de los componentes afectados y de las consecuencias que

el cambio vaya a tener en el servicio (para ITIL, cambio es todo aquello que pueda afectar al nivel del servicio prestado, desde un cambio en la aplicación hasta un cambio en el hardware de infraestructura). Por ejemplo, cuando un nuevo requerimiento genera cambios en componentes distribuidos, los diferentes cambios a realizar deben coordinarse cuidadosamente para disminuir el efecto en el servicio prestado. En este punto, el proceso de desarrollo y los procesos de entrega y soporte se vinculan íntimamente. Es así que es necesario contar con información detallada de las dependencias, y mantener esa información actualizada y accesible. En el modelo de ITIL, esto se implementa mediante una CMDB (*Configuration Management Database*), que mantiene el control de estado y versión de todos los componentes de sistemas de la organización. Para nuestra perspectiva, los diagramas de dependencias integrados aportan mayor consistencia a la información de administración de configuración y permiten evaluar constantemente el estado de los componentes sujetos a cambios.

Estas alternativas siguen en general un camino diferente al elegido por nosotros. Aunque ponemos muchas esperanzas en el desarrollo y mejora de los lenguajes y la administración de infraestructura, mantenemos centrado nuestro trabajo en la tarea de documentación.

En el capítulo siguiente desarrollamos en detalle la problemática planteada en la introducción para formalizar las bases de nuestro trabajo.

IV. Problemas de documentación integradora

El problema que dio origen a este trabajo puede expresarse así: Si tenemos que documentar un módulo A, trataremos de mantener su documentación cercana al módulo A (para que se mantenga actualizada y sea fácil de encontrar), y si tenemos que documentar otro módulo B, haremos lo mismo, más allá del tipo de módulo de A o de B. Ahora bien, si queremos documentar la relación entre A y B, es un poco más complicado decidir dónde y cómo hacerlo.

En el caso particular en que los componentes son del mismo tipo, los términos disponibles para describir su relación se pueden obtener en forma inmediata del contexto y son fácilmente administrables mediante herramientas. En cambio, si los componentes son de distinto tipo, es más difícil documentar su relación.

Por ejemplo, si tenemos que decir algo sobre una clase, lo documentamos en un comentario en el código, si tenemos que decir algo sobre una tabla, lo registramos en su descripción. Ahora, si tenemos que decir algo de la relación entre esa clase y esa tabla, no es tan claro dónde expresarlo.

1. Una primera aproximación

A primera vista, podríamos plantear que el problema se resuelve describiendo la relación en la clase o en la tabla, o en ambas. En palabras de Bertrand Meyer, cumplimos con la regla de “Interfaces Explícitas” si:

> *“Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.”*
>
> *“Cuando dos módulos A y B se comunican, esto debe ser obvio a partir del texto de A o B o ambos”. [Meyer 1985, pág. 50]*

Sin embargo, no es tan simple aplicar esta regla por varios motivos. En las secciones siguientes se describen algunos de estos problemas:

- **Dependencia centrada en la implementación**

  Cuando la dependencia está centrada en la implementación de los módulos y por lo tanto no es aparente en su interfaz. Esto corresponde a un error de diseño cuando se trata de elementos de la misma vista, pero entre elementos de distinta vista el concepto de interfaz, o parte pública,

es mucho menos fuerte.

- **Dependencia basada en acoplamiento indirecto**

  Por otro lado, la dependencia puede estar dada por el acoplamiento indirecto basado en un tercer componente:

  “One of the problems in applying the Explicit Interfaces rule is that there is more to intermodule coupling than procedure call; data sharing, in particular, is a source of indirect coupling.”

  “Uno de los problemas para aplicar la regla de Interfaces Explícitas es que el acoplamiento entre módulos puede no deberse solamente a llamadas a procedimientos; los datos compartidos, en particular, son una fuente de acoplamiento indirecto.” [Meyer 1985, pág. 50]

- **Dependencia entre componentes de vistas distintas**

  Finalmente, es posible que la dependencia aparezca entre elementos de vistas distintas (en general con distinto nivel de abstracción). Es decir, cuando se vuelve un problema de arquitectura y no de diseño. En este caso, es mucho más difícil describir una relación, y determinar dónde registrar la documentación de la misma. El problema es que cada vista define las relaciones válidas entre sus elementos, con lo cual cuando trabajamos entre vistas nos encontramos en principio con un “vacío” de vocabulario, una carencia de términos para expresar la relación. Además, es más difícil establecer dónde y cómo documentarla.

  Por ejemplo, si una clase de objeto utiliza un archivo de configuración para definir su estado inicial, documentar la dependencia en el código fuente de la clase haría que ese comentario quede excesivamente lejos del modelo de componentes del cual forma parte el archivo. De la misma manera, si agregamos un comentario en el archivo, la referencia a la clase que podamos incluir ahí está lejos del modelo de código fuente y más aún del modelo lógico en el que la clase se origina. Además, es poco probable que la clase y el archivo aparezcan juntos en el modelo de la vista de componentes, puesto que la clase probablemente esté oculta dentro de un componente mayor que la contenga.

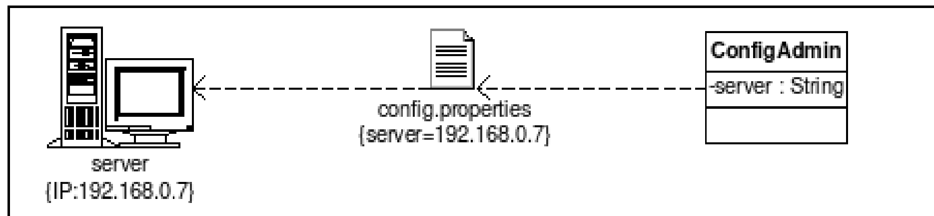

Figura 2: Diagrama de dependencias entre elementos de distinta vista. Una Clase (elemento de la vista lógica), un archivo de configuración que fija los valores de sus propiedades (elemento de la vista de componentes) y un servidor físico (elemento de la vista de distribución).

Esta última aproximación es la que utilizaremos como base para el resto de nuestro trabajo, asumiendo una perspectiva de arquitectura para el estudio del sistema en desarrollo. Nuestra propuesta es entonces trabajar en el uso de modelos de dependencia entre elementos de distintas vistas. En el capítulo siguiente formalizamos el rol de los modelos en la documentación de arquitectura para establecer la base teórica para la prueba de concepto desarrollada.

V. Documentación basada en modelos

Un modelo es una representación abstracta del sistema, es decir que deja afuera alguna información sobre el mismo. Naturalmente, la elección de la información a incluir es fundamental, de manera que sea suficiente para comprender lo documentado y no sea demasiado compleja. La idea de representación incompleta inherente al concepto de modelo es una que desarrollaremos en las secciones siguientes y utilizaremos como característica central de nuestra propuesta de documentación.

Ahora bien, más allá de la selección de la información incluida en el modelo, el problema está en la interpretación del mismo. La interpretación, como se planteó en la introducción, depende mucho del rol del lector dentro del proyecto y del momento en que se realiza. Formalmente, la interpretación consiste en asignar significado al modelo, en un contexto variable, y esencialmente en los términos presentes en la cabeza del lector. Esos términos subjetivos, a su vez, tienen significado en relación al sistema, es decir, cada uno representa algún elemento o característica del sistema.

Los problemas con este proceso son múltiples:

- No es fácil determinar el significado de los elementos del modelo.
- El significado de cada término depende de la perspectiva o rol de cada uno.
- En el caso ideal, un mismo elemento aparece transformado en cada vista distinta. En la realidad, cada elemento puede transformarse en un conjunto de otros elementos, y esa transformación ser más compleja que cada uno de los elementos por separado.

1. Especificación vs. Descripción

La documentación basada en modelos tiene dos funciones posibles: La especificación o la descripción de un sistema. Las diferencias entre ambas son profundas, pero esencialmente pueden reducirse a dos: La diferencia histórica y la de alcance.

La diferencia histórica está en que la especificación debe realizarse antes que el sistema, mientras que la descripción puede hacerse durante o después. La

diferencia de alcance está en que para estar completa, una especificación requiere abordar todos los aspectos para un nivel de detalle dado, mientras que la descripción puede tomar una perspectiva parcial y enfocada, resaltando algún aspecto particular.

2. El problema del acuerdo entre el modelo y el sistema

Más allá de las diferencias, en ambos tipos de modelo puede considerarse el problema del acuerdo entre el modelo y el sistema. En el caso de la especificación, el sistema debe cumplirlas para ser **correcto**. En el caso descriptivo, ambas alternativas son posibles, si hay desacuerdo entre el modelo y el sistema puede considerarse que el modelo es incorrecto, o que el sistema lo es. En el capítulo 5, dedicado a IDM, nuestra herramienta de control integrado de dependencias, desarrollaremos en profundidad este concepto desde la perspectiva de su aplicación práctica. En [Seidewitz 2003] se define como **correcto** al modelo que describe a un sistema, y como **válido** al sistema que adhiere a su especificación (nosotros usaremos el término **correcto** puesto que válido tiene connotaciones relacionadas con la aceptación por parte de los usuarios):

> *"A model is a set of statements about some system under study. Here, statement means some expression about the SUS that can be considered true or false (although no truth value has to necessarily be assigned at any particular point in time).*
>
> *We can use a model to describe a SUS. In this case, we consider the model correct if all its statements are true for the SUS."*
>
> *"Un modelo es un conjunto de enunciados sobre algún sistema en estudio. Aquí, enunciado significa una expresión sobre el sistema en estudio que puede ser considerada verdadera o falsa (aunque no es necesario asignar un valor de verdad en ningún momento en particular).*
>
> *Podemos usar un modelo para describir un sistema en estudio. En este caso, consideramos al modelo correcto si todos sus enunciados son verdaderos para el sistema en estudio." [Seidewitz 2003, pág. 27]*

En este trabajo nos concentraremos estrictamente en modelos descriptivos, dejando de lado el problema de la especificación y explorando las aplicaciones prácticas de los modelos descriptivos de dependencias. En el próximo capítulo se detalla la función de los diagramas de dependencia para describir la idea básica aplicada en la prueba de concepto.

VI. Diagramas de dependencia

Una dependencia entre dos entidades A y B es una relación (A,B) tal que si B sufre un cambio, A puede sufrir un cambio. Esta definición evidencia el alto nivel de abstracción de la relación: **Cualquier cambio en B puede provocar un cambio en A**.

Este alto nivel de abstracción le imprime a la dependencia algunas características importantes: No implica una connotación estructural (la relación no requiere que un elemento contenga a otro ni que sea de la misma vista), no describe necesariamente un cambio de forma (es decir que los cambios pueden no ser aparentes o estar en la implementación), y puede representar aspectos puramente semánticos (A y B pueden seguir vinculados pero con la esencia de esa vinculación alterada).

Por estas razones, más allá de la afinidad intuitiva con el concepto, la relación de dependencia nos resulta apropiada para enfrentar el problema que nos ocupa.

El uso de diagramas, sin embargo, no es necesariamente la única forma de mostrar las dependencias documentadas. Hay múltiples alternativas posibles, entre ellas la descripción textual, las listas, las tablas y las matrices (con los módulos como filas y columnas y las celdas registrando una dependencia). Aunque estas herramientas pueden ser apropiadas en otros contextos, consideramos que en el caso de elementos de distintas vistas, y dada la relativamente pequeña cantidad de información a ser mostrada, los diagramas son la forma más aconsejable de plasmar la información de dependencias. Particularmente, es recomendable utilizar iconos y ayudas gráficas claras para destacar de qué tipo de componente de trata, dado que no vamos a contar con un contexto acotado que ayude a identificar a los componentes.

El otro problema a tener en cuenta es el de darle significado a las relaciones. Por ejemplo, estamos acostumbrados a ciertas dependencias entre elementos de una misma vista, como por ejemplo, el código fuente A “usa” el código fuente B, o un elemento de la tabla A es identificado por un elemento de la tabla B. Algunas dependencias, como “usa”, tienen significado en un contexto mayor al habitual, pero eso no es necesariamente cierto para todas (por ejemplo, la dependencia entre objetos donde A “instancia” B no es aplicable entre servidores físicos). Por lo tanto, es importante aplicar las dependencias a un

nivel de abstracción apropiado a los componentes que relaciona.

En el capítulo que sigue se presentan buenas prácticas de documentación en general y particularmente aplicables a la descripción de arquitecturas a través de un conjunto de vistas.

VII. Documentación de arquitectura en la práctica

Como planteamos en la introducción, documentar una arquitectura de software requiere documentar las vistas y luego documentar la información que aplica a más de una vista. Como plantean Clements et al en *Documenting Software Architecture*:

> *"The basic principle of documenting an architecture as a set of separate views brings a divide-and-conquer advantage to the task of documentation, but if the views are irrevocably different, with no relationship to one another, nobody would be able to understand the system as a whole."*
>
> *"El principio básico de documentar una arquitectura como un conjunto de vistas aporta una ventaja de divide y vencerás a la tarea de documentación, pero si las vistas son irrevocablemente diferentes, sin relación la una con la otra, nadie podría entender el sistema como un todo." [Clements 2003, pág. 200]*

La relación entre vistas es trivial cuando los elementos de vistas distintas describen el mismo concepto dentro del sistema, por ejemplo entre una Clase de la vista lógica y la Clase como archivo compilado de la vista de componentes. Cuando la transformación no es "uno a uno", o cuando hay información específica que es necesario agregar a la transformación, es más complejo documentar esa arquitectura.

Nuestra propuesta de usar dependencias entre elementos de distintas vistas en un modelo o vista integrada de dependencias se corresponde directamente con lo planteado por Clements et al:

> *"Sometimes, the most convenient way to show a strong relationship between two views is to collapse them into a single combined view"*
>
> *"A veces, la forma más conveniente de mostrar una relación fuerte entre dos vistas es colapsarlas en una única vista combinada" [Clements 2003, pág. 200]*

Ahora bien, para definir esta vista combinada existen varias alternativas:

- Describir la relación de transformación entre las vistas
- Crear una vista mixta con los elementos originales de ambas vistas que tienen relación entre si.

Nuestra propuesta consiste en utilizar la segunda opción, utilizando relaciones de dependencia para vincular elementos de vistas distintas en una única vista integrada. Las razones para nuestra elección ya fueron presentadas, pero podemos expresar este razonamiento como sigue, en otras palabras:

> *“Do this if the relationship between the two views is strong, the mapping from elements to elements in the other is clear, and documenting them separately would result in too much redundant information.”*
>
> *“Haga esto si la relación entre las dos vistas es fuerte, la correspondencia de los elementos con los elementos de la otra vista es clara, y documentarlos separadamente resultaría en demasiada información redundante” [Clements 2003, pág. 201]*

Para crear esta vista combinada es necesario cumplir con algunas condiciones, esencialmente definir qué componentes y qué relaciones son válidos dentro de esa vista. En particular, es necesario aclarar la notación a utilizar y las propiedades que es aceptable utilizar para describir cada tipo de elemento.

Para definir esta vista combinada existen dos alternativas claras (descritas en [Clements 2003, pág 201]):

- Una vista superpuesta (*overlay*), es decir una superposición de dos o más vistas, pero conservando los elementos y relaciones presentes en las vistas originales.
- Una vista híbrida, es decir una vista nueva basada en ambas vistas pero con elementos propios, acompañada de una guía para construir modelos dentro de esa vista.

En esta clasificación, nuestra propuesta se corresponde directamente con una vista superpuesta, puesto que las relaciones de dependencia, dado su nivel de abstracción, aparecen en múltiples vistas y no necesitan ser redefinidas, y los elementos relacionados mantienen la forma y propiedades de la vista original.

Como consecuencia, es más fácil adaptarse a utilizar esta vista sin tener que acceder a una guía de estilo o redefinir la sintaxis del lenguaje de modelado.

Aunque no es necesario utilizarlo, se desprende de lo expresado que utilizar un lenguaje de modelado de amplia difusión como el UML complementa las ventajas descriptas aportando una sintaxis básica reconocida y facilitando el acceso de los desarrolladores a los modelos. Como se verá en la sección siguiente, el UML aporta también mayor facilidad para la integración con herramientas de modelado existentes y la distribución de los modelos.

1. Reglas de buena documentación

Las reglas que se describen a continuación están tomadas de [Fontdevila 2003], basadas a su vez en [Clements 2003], y aportan una visión práctica de la tarea de documentación.

Para cada regla se expresa su enunciado, los comentarios y los aspectos en que se aplica a nuestro caso particular.

Regla 1: Escriba la documentación desde el punto de vista del lector

"Póngase en el lugar del lector, no use vocabulario específico, sea cortez y recuerde que el documento se leerá muchas más veces que las que fue escrito."

En nuestra propuesta, es fundamental mantener la perspectiva del desarrollador, los diagramas de dependencia integrados que concebimos no tienen sentido alejados de los elementos componentes fundamentales del sistema.

Regla 2: Evite la repetición innecesaria

"Cuide no decir las cosas más de una vez a menos que sea necesario. Intente referir a otras partes del documento u otros documentos. Aunque a veces promueve una más profunda comprensión o transmite énfasis, la repetición con variaciones puede ser confusa."

Mantener la documentación de la dependencia en un diagrama integrador evita tener que redundar la información de dependencia en el texto (definición formal) de ambos elementos, y permite observar la

dependencia desde cualquiera de los elementos relacionados. Recordar que la regla de interfaces explícitas de Bertrand Meyer exigía que la dependencia fuera evidente al observar por lo menos uno de los elementos, pero no lo exigía de ambos (ver capítulo II, pág. 13).

Regla 3: Evite la ambigüedad, explique la notación

"Utilice las mismas palabras para referirse a los mismo conceptos, no busque dar matices para mejorar una exposición, elija los aspectos a desarrollar y concéntrese en ellos.

En cada diagrama, agregue una clave con una referencia de la notación utilizada, no confíe ciegamente en estándares para garantizar que el lector comprenda la notación."

La utilización de un subconjunto muy limitado de elementos del UML (relaciones de dependencia como flechas de línea punteada y entidades como clases, componentes y procesadores físicos) permite mantener una notación simple y accesible, que puede ser fácilmente descripta por una clave de referencia.

Regla 4: Use una organización estándar

"Mantenga una organización estándar de la documentación, ayudará a los lectores a encontrar lo que buscan y le servirá para asegurarse de que está completa y correcta."

Organizar los diagramas de dependencia en un directorio estándar del modelo, que puede corresponder a la vista de dependencias integradas, facilita el acceso a los modelos y ayuda a crearlos estableciendo para ellos un lugar específico predefinido.

Regla 5: Registre las razones

"Deje constancia de las razones que motivaron las decisiones tomadas y de las alternativas descartadas."

Para registrar las razones de cada decisión es difícil utilizar diagramas, aunque pueden agregarse notas aclaratorias (Ver [Fontdevila 2003] para una propuesta de Log de decisiones como forma de registrar las razones detrás del proceso de definición de la arquitectura).

Regla 6: Mantenga la documentación actualizada pero no

demasiado

“Mantenga la documentación actualizada, de lo contrario no será consultada. Cuando reciba una consulta, controle la documentación y refiera a la persona al documento. Evite poner demasiado detalle en la documentación para no redundar información entre el software y la documentación.”

Utilizar una vista simple como la de dependencias integradas mantiene una mínima redundancia con el resto de los modelos y el código, puesto que deja de lado toda información de tipo estructural.

Regla 7: Revise si la documentación cumple su propósito

“Consiga que los lectores revisen y den su aprobación a la documentación, esa es la prueba última de su utilidad.”

Como todo herramienta de comunicación, la vista integrada de dependencias debe ser revisada por lectores para confirmar su capacidad de transmitir la información que almacenan.

Sin embargo, en el caso de nuestra propuesta, la herramienta que describimos en el siguiente capítulo permite contrastar los diagramas con el sistema y evaluar su capacidad de describir el mismo.

Regla 8: Publique eficientemente la documentación

“Es importante lograr que la información esté disponible para todos los lectores en forma rápida y a un bajo costo, para fomentar su uso. Preguntar al que sabe es siempre preferible a usar documentación difícil de acceder.”

La herramienta desarrollada permite ver y validar los diagramas dentro del entorno de desarrollo, manteniéndolos cerca del código del sistema que documentan.

Regla 9: Utilice herramientas

“Tanto el formato como las herramientas de edición seleccionadas deben promover la comodidad y permitir el acceso de todos los lectores. Elija herramientas que en lo posible permitan fácilmente vincular documentos entre sí y buscar en el repositorio de

información.”

La herramienta que proponemos se integra con otras herramientas UML y con el entorno de desarrollo.

Regla 10: Tenga en cuenta los objetivos del proyecto

“Al redactar la documentación, el autor debería tener en mente los objetivos estratégicos del proyecto y buscar imprimir a la documentación la dirección correspondiente.”

Como se describió en la introducción, una vista integrada de dependencias ayuda a realizar el análisis de impacto de los cambios y mejora la flexibilidad del sistema. No es recomendable para sistemas con ciclo de vida muy acotado y baja probabilidad de cambios, ya que el esfuerzo de desarrollar esta documentación puede no traer beneficios significativos. Por otro lado, en metodologías ágiles puede ser apropiado porque puede mantenerse cerca del código y automatizarse el control del sistema en base a la documentación (incluso realizarse pruebas continuas de la documentación mediante herramientas como la presentada en el siguiente capítulo).

La aplicación práctica de los modelos integrados de dependencias facilita documentar información para la cual no tenemos un espacio definido. Esa característica aporta un valor especial: Facilita la construcción de la documentación ofreciendo un espacio para ubicar la información y recíprocamente aporta una organización definida de la documentación que permita buscar y acceder con facilidad a la misma.

En las metodologías orientadas al plan y los documentos, este modelo se organiza directamente como una sección de la documentación de arquitectura y diseño, en particular en los modelos orientados a vistas como los que describimos se puede agregar como una vista más o como un apéndice.

En las metodologías ágiles, el uso de diagramas simples y capaces de ser validados automáticamente para ofrecer información de consistencia del sistema pueden ser agregados valiosos a la información contenida en el código. En este caso, el uso de una herramienta integrada en el entorno de desarrollo facilita mantener cerca del código la información del sistema.

El uso del modelo integrado de dependencias, ya sea como una vista extra en el modelo de arquitectura o como simple herramienta de control de corrección del sistema en desarrollo, puede aportar valor al proceso de empaquetar la información que finalmente termina componiendo un sistema de software.

En el capítulo que sigue se describe la herramienta IDM y cómo procesa los modelos de dependencias para evaluar el sistema que describen.

VIII. IDM: Una herramienta de control de dependencias

IDM (*Integrated Dependencies Model*) es la herramienta desarrollada como prueba de concepto para las ideas planteadas en este trabajo. Es capaz de procesar diagramas de dependencia escritos en UML e informar si esas dependencias efectivamente se cumplen en la realidad.

IDM está formado por los siguientes módulos (ver figura 3):

IDM Vista: Un *plugin* (módulo agregado al entorno de desarrollo) de Eclipse para la evaluación gráfica de los modelos. La funcionalidad de IDM Vista está integrada naturalmente con Eclipse, de manera tal que un archivo .xmi (*XML Metadata Interchange*, que contiene un diagrama UML en formato XML) es abierto con IDM Vista por Eclipse. Así, basta abrir un archivo .xmi para ejecutar los controles y ver el resultado.

IDM Consola: Una aplicación de consola capaz de realizar los controles sobre los diagramas sin interfaz gráfica de usuario.

IDM Núcleo: Un conjunto de librerías que describen el modelo de dependencias entre elementos de distintas vistas. Incluye también una plantilla para simplificar la creación de modelos de dependencia en UML.

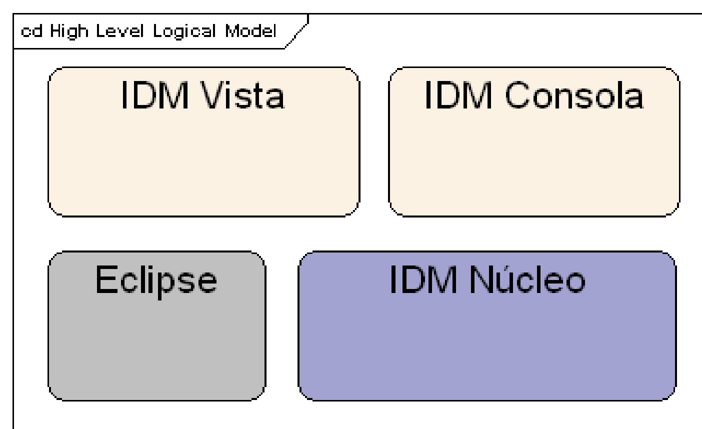

Figura 3: Estructura de módulos de IDM. En gris están los módulos preexistentes, en violeta las librerías y en beige las aplicaciones.

La herramienta está diseñada para acompañar el proceso de desarrollo, con las siguientes características:

- Define una vista mixta para documentar relaciones entre elementos de distintas vistas.

- Promueve la documentación de información específica tradicionalmente difícil de ubicar.
- Convierte los diagramas en entidades activas, con comportamiento, agregando una nueva dimensión a la información registrada.
- Agrega un atractivo más para motivar a los desarrolladores a generar la documentación.
- Permite ejecutar múltiples veces la evaluación de un diagrama, por ejemplo antes y después de realizado un cambio, ayudando a mantener la consistencia del sistema en desarrollo.
- Utiliza estándares de la industria como UML y XMI.
- Está diseñada para integrarse al proceso de desarrollo sin reemplazar a otras herramientas, si no agregando su funcionalidad.
- No requiere información muy detallada en los diagramas para funcionar, a diferencia de otras herramientas estilo CASE.

1. Cómo se usa IDM

La utilización de la herramienta IDM está pensada como soporte a las tareas de desarrollo y documentación, buscando no modificar en lo posible las prácticas y herramientas tradicionales del proceso.

El desarrollador crea el modelo de dependencias para registrar información específica a medida que realiza el diseño o la construcción del sistema, con una herramienta estándar de UML. Luego, procede a utilizar IDM para evaluar si el sistema en desarrollo se corresponde con el modelo.

El proceso, entonces, tiene tres partes fundamentales:

1.- Documentar las dependencias con una herramienta UML.

2.- Desarrollar total o parcialmente el sistema.

3.- Evaluar si el sistema desarrollado se corresponde con el modelo ejecutando para ello IDM sobre el modelo UML generado y el sistema en estudio.

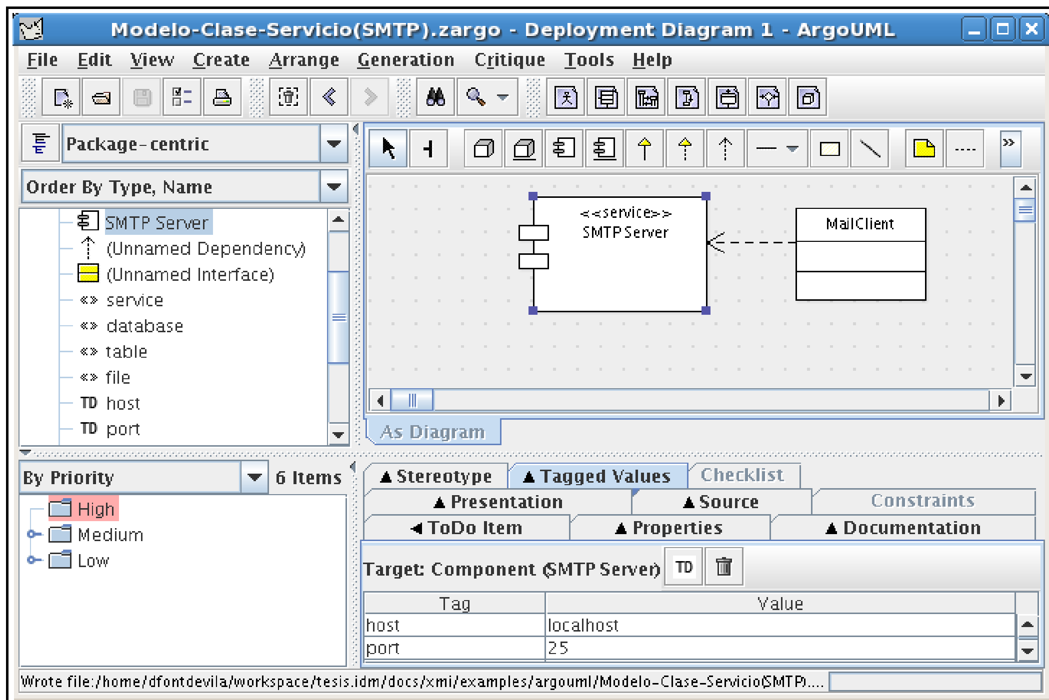

Figura 4: Creación de un modelo con una herramienta UML

Por ejemplo, para documentar que una clase del sistema en desarrollo, MailClient, depende de un servicio de envío de mail SMTP, el proceso sería el siguiente:

1. Crear un modelo de dependencias con una herramienta UML. La figura 4 muestra el uso de ArgoUML para crear un diagrama.
2. Desarrollar total o parcialmente el sistema (este paso también puede realizarse antes de documentar las diferencias).
3. Evaluar el modelo de dependencias mediante IDM para ver si el sistema desarrollado se corresponde con él. La figura 5 muestra como las dependencias pueden evaluarse mediante IDM Vista desde Eclipse.

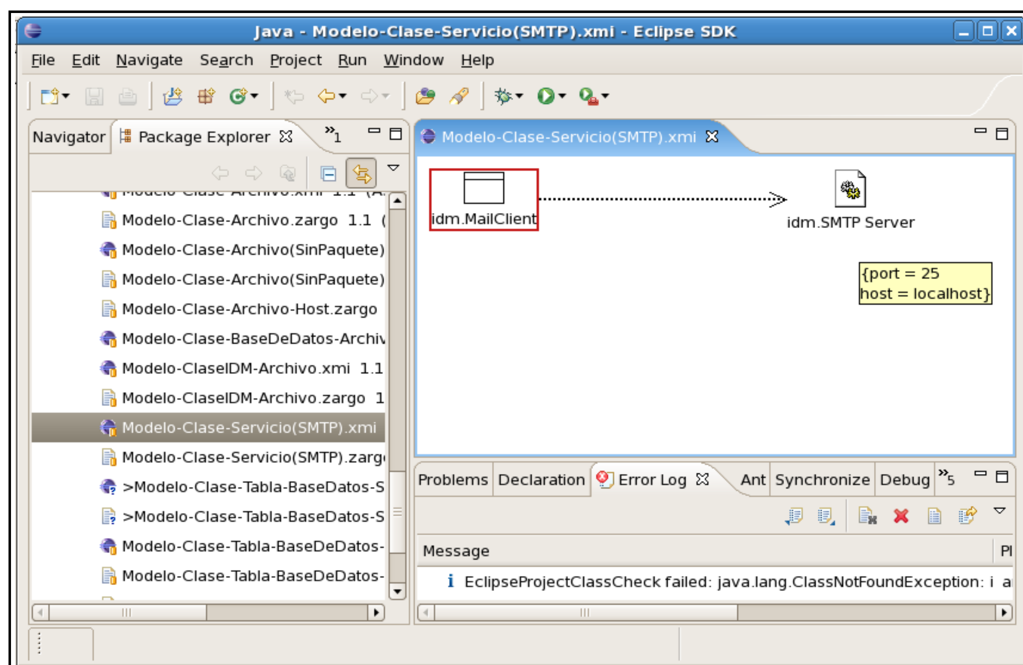

Figura 5: Ejemplo de uso de IDM Vista

En este caso, IDM muestra que hay algún problema con el modelo (la clase idm.MailClient aparece destacada en rojo), aunque el servicio de mail SMTP no presenta problemas. En este caso esto se debe a que la clase MailClient no ha sido encontrada en el sistema en desarrollo.

Entonces, desde el punto de vista del desarrollador, nuestra propuesta tiene dos componentes: Primero, plantear una forma de documentar información que no es fácil de registrar, y segundo, agregar valor a esa documentación (motivando también a que sea generada) mediante su evaluación automática para decidir si el sistema en desarrollo cumple o no con la documentación.

2. <u>Escenarios de uso de IDM</u>

Formalmente, el uso de IDM consta de dos casos de uso principales como muestra el modelo:

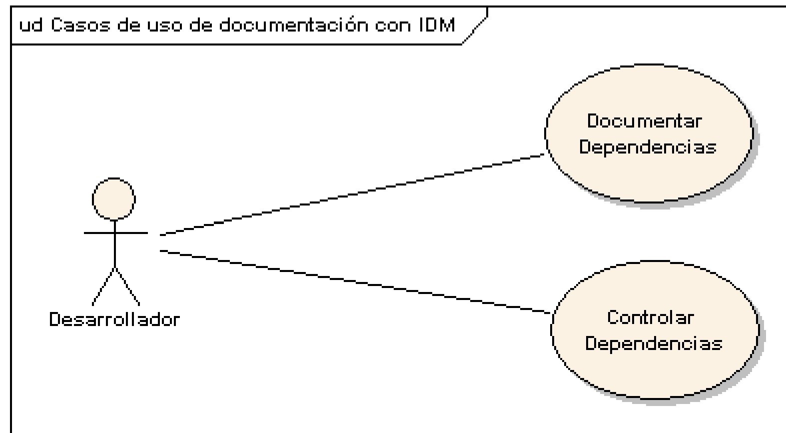

Figura 6: Modelo de casos de uso de documentación con IDM

El primer paso consiste en documentar las dependencias mediante diagramas, esta primera actividad del proceso es la que incorpora el conocimiento de los participantes sobre el sistema en desarrollo.

Proceso: Documentar dependencias

Insumos: Conocimiento del sistema en desarrollo.

Productos: Modelo integrado de dependencias en UML en formato XMI.

Luego, un desarrollador puede ejecutar la herramienta sobre el modelo y generar un gráfico o reporte de log que le mostrará el diagrama resaltando las dependencias no satisfechas en el sistema desarrollado.

Proceso: Controlar dependencias

Insumos: Modelo integrado de dependencias en UML en formato XMI.

Productos: Informe de corrección del sistema en desarrollo con respecto al modelo.

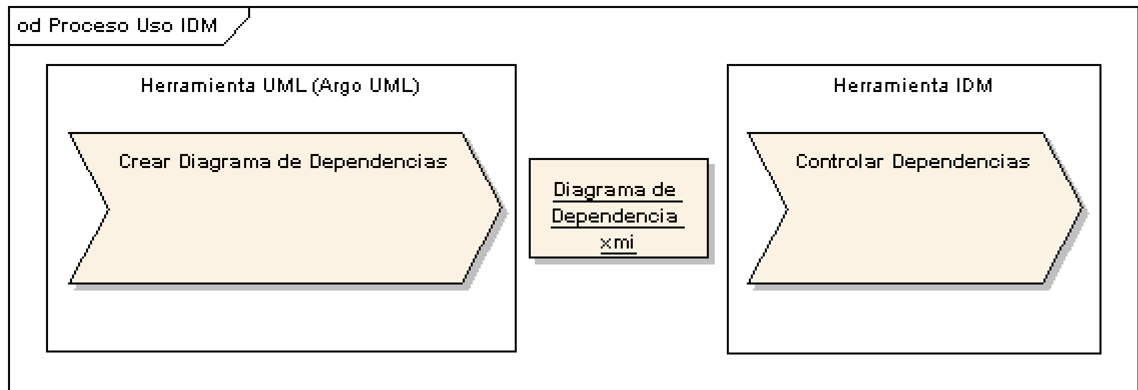

Figura 7: Proceso de control de dependencias usando IDM

3. Evaluación del sistema en estudio en base al modelo

En la introducción se planteó la cuestión de la interpretación de un modelo descriptivo como una relación entre el modelo y el sistema en estudio. Se describió el concepto de **corrección** como el acuerdo entre el modelo y el sistema en estudio. Usualmente, si el modelo es descriptivo, se considera que interpretar el modelo consiste en ver si el modelo es correcto en relación al sistema, no si el sistema es correcto en relación al modelo. En nuestro caso, sin embargo, puesto que los modelos aportan información sobre el sistema en estudio y los supondremos correctos, consideraremos que es el sistema en estudio el que no es correcto si no cumple con lo especificado en el modelo.

Usamos entonces el término **corrección** para describir la propiedad del sistema en estudio de satisfacer las dependencias documentadas para él. La utilidad de la herramienta es la de dar información sobre el sistema en estudio, interpretando si las dependencias documentadas en el diagrama están satisfechas en el mismo.

IDM puede considerarse como una herramienta de “ejecución de diagramas” o “running UML”, puesto que genera y ejecuta controles para todas las dependencias documentadas. A diferencia de una herramienta de ingeniería directa (forward engineering), no se aplica a generar el sistema en estudio sino a analizarlo a posteriori.

En segunda instancia, la idea es que IDM sirva como elemento motivador para promover en los desarrolladores la generación de los modelos, puesto que su utilización permite agregar valor a los diagramas, que pueden ejecutarse múltiples veces para controlar el estado del sistema en desarrollo de acuerdo a la documentación. Además, es probable que contar con una herramienta gráfica también resulte más atractivo que tener simplemente un reporte en formato de texto con los resultados.

Finalmente, hay que destacar que la herramienta se utilizará durante el desarrollo del sistema para estudiar sus dependencias y evaluarlo a medida que se construye y se instala, es decir que su utilización depende fuertemente de los procesos de administración de cambios (promoción entre ambientes, administración de configuración, etc.).

4. Análisis de modelos basado en metadatos

Como se describió en la introducción, el objetivo de este trabajo está centrado en el uso de modelos descriptivos; el problema está en asignar un significado a los modelos. Darle significado a un modelo es establecer una relación entre los elementos del modelo y los elementos del sistema en estudio que nos permita interpretar cada elemento del modelo (ver [Seidewitz 2003], pág. 27).

Para interpretar un modelo y decidir si es **correcto** es necesario establecer una relación o transformación entre los elementos del modelo y los elementos del sistema en estudio, tal que si existe para cada elemento del modelo un elemento par correspondiente en el sistema en estudio, el modelo y el sistema se consideran consistentes, o cada uno correcto en relación al otro.

Ahora bien, para interpretar el modelo y darle significado, es necesario analizar el modelo y descomponerlo, estableciendo cuales son sus componentes y transformando cada uno para buscar y obtener el componente par correspondiente en el sistema de estudio. Este proceso implica un aumento de nivel de abstracción, es decir pasar del nivel del modelo al del metamodelo, que actúa como un metalenguaje que establece la sintaxis para el modelo y nos permite analizarlo. Por ejemplo, si el modelo describe una clase de objetos C, es necesario evaluar a C como instancia de la metaclase Clase con sus atributos particulares, esencialmente el nombre de la clase C, que será probablemente "C", y con esa información buscar en el sistema en desarrollo una clase de objetos correspondiente.

En el caso de la herramienta de análisis de dependencias, esto es particularmente difícil porque no solamente hay que obtener la información necesaria sobre los componentes sino también sobre las relaciones.

5. Historia de una implementación

Para construir la herramienta, se planteaban dos alternativas: Definir un metamodelo ad hoc para los diagramas y construir un editor para los mismos, o utilizar un lenguaje de modelado estándar para los diagramas con su correspondiente metamodelo y utilizar un editor capaz de intercambiar diagramas en base a ese metamodelo.

Optamos por la segunda opción, que permite no sólo integrar IDM en forma natural en el proceso de desarrollo, si no que favorece una separación clara entre la tarea de documentación (llevada a cabo con una herramienta de modelado tradicional) y la tarea de evaluación automática del modelo (llevada a cabo con IDM).

En este contexto, decidimos utilizar el lenguaje de modelado UML y el estándar XMI (XML Metadata Interchange) para el intercambio de modelos por las siguientes razones:

- Amplia aceptación del UML en la industria y el ámbito académico.
- Existencia del estándar XMI para intercambio de modelos UML sobre XML.
- Disponibilidad como software libre de incipientes implementaciones de APIs para análisis de modelos que cumplan el estándar XMI.
- Disponibilidad de herramientas UML de software libre y propietarias capaces de exportar a XMI (aunque con diferencias notables que requirieron la selección de una única herramienta UML, en contra del espíritu del estándar).

Es necesario destacar que el estándar XMI en la versión 1.1 no incluye información de metadatos sobre los diagramas del modelo, si no solamente sobre el modelo mismo. Esto impone algunas consideraciones especiales para la construcción de los modelos a intercambiar, particularmente en el tamaño de los mismos, pero no afecta los aspectos centrales de nuestro trabajo. En general, cuando decimos diagrama, nos referimos a una porción lógica del modelo que se organiza para mostrar gráficamente el modelo, pero no vamos a contar con información sobre los diagramas en XMI. Esto tiene como resultado que parte del contenido documentado (ubicación y tamaño de los elementos, disposición

de las flechas de dependencia, colores, etc.) se pierda al exportar el modelo a XMI y por lo tanto no esté disponible para IDM. Se espera que en XMI 2.0 se agregue al estándar información sobre los diagramas.

Para el análisis de los modelos en formato XMI fue necesario seleccionar una API que implementara el estándar y permitiera el acceso al contenido de los diagramas. El JCP (Java Community Process) define un estándar Java para la implementación de repositorios de metadatos llamada JMI (Java Metadata Interchange, identificado como Java Specification Request, JSR, 000040), cuyas implementaciones deben soportar XMI.

Se evaluaron dos implementaciones de software libre del estándar JMI:

- Unisys XMI (implementación de referencia de JMI)
- MDR, NetBeans Metadata Repository (implementación de SUN Microsystems)

Se seleccionó la implementación de NetBeans (MDR) por ser la más accesible y claramente publicada.

A pesar de la promesa del estándar de contar con un mecanismo de intercambio de modelos basado en el formato XMI, en la práctica no fue posible utilizar archivos generados por distintas herramientas. Como el estándar no define cómo representar los diagramas ni la disposición gráfica de los elementos, cada herramienta de UML representa a su manera esa información. Aunque la misma no es estrictamente necesaria para el análisis automático de las dependencias, el análisis los archivos .xmi generados por algunas herramientas falla a causa de la existencia de información no estándar.

Entre las herramienta evaluadas se encuentran:

- Magic Draw (propietaria, No Magic Inc., http://www.magicdraw.com)
- Enterprise Architect (propietaria, SparxSystems, http://www.sparxsystems.net)
- ArgoUML (software libre, Proyecto Tigris, http://argouml.tigris.org)

Todas cuentan con la capacidad de exportar a XMI, pero ArgoUML resultó la única capaz de generar archivos .xmi que el MDR de NetBeans pudiera analizar.

Además, siendo ArgoUML un proyecto de software libre de origen académico (ver [Robbins 2000]), resultaba más afín a este trabajo.

Como se dijo al comienzo de este capítulo, IDM cuenta con tres módulos principales. A continuación se detallan los principales aspectos relacionados con la implementación de cada uno.

5.1. Implementación de IDM Núcleo

El núcleo de IDM está compuesto por un conjunto de librerías Java que definen el modelo de solución básico de IDM:

- Modelo de dominio: Elementos, dependencias y controles de dependencia.
- Analizador de diagramas: Basado en XMI.
- Recolector de información: Basado en la herramienta de registro de eventos (*logging*) log4j.

La primera función del núcleo de IDM es definir el modelo de dominio bajo estudio: Elementos y relaciones de dependencia entre ellos, más un modelo de Controles de dependencia a ser ejecutados sobre los mismos. Además, y no menos importante, implementa los controles de dependencia soportados. Como se ve en la sección de arquitectura, algunos controles tienen varias implementaciones alternativas, mientras que otros sólo cuentan con una implementación básica.

En segundo lugar, el núcleo de IDM contiene las herramientas para analizar modelos UML representados en XMI y extraer la información de los mismos para que pueda ser procesada. En particular, la falta de soporte en XMI 1.1 para diagramas obliga a utilizar modelos acotados cuyo tamaño corresponda aproximadamente al de un diagrama. También impide por ahora que el núcleo administre información sobre la disposición gráfica de los elementos en el modelo.

Para facilitar el proceso de intercambio de modelos, se definió una plantilla para modelos UML que define explícitamente algunos estereotipos estándar de UML que son necesarios para interpretar el modelo (como por ejemplo, *table* para una tabla), además de algunas etiquetas (tags o tagged values, como *port* para

definir un puerto) que permiten especificar propiedades de los elementos en forma estándar. Esta plantilla debe utilizarse con la herramienta de modelado UML, y por ahora, está solamente implementada para ArgoUML.

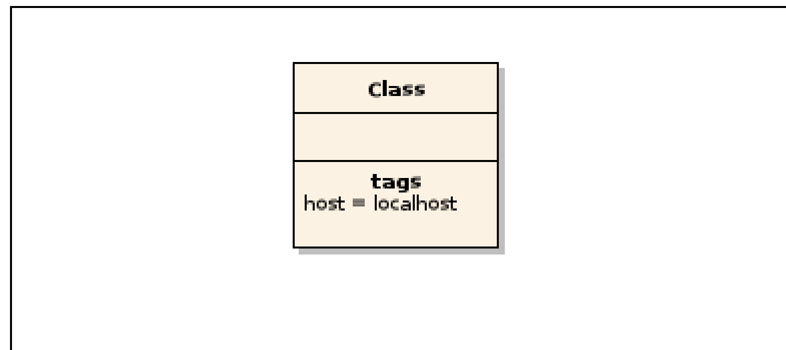

Figura 8: Ejemplo de uso de etiquetas UML (tags o tagged values) desarrollado con Enterprise Architect porque ArgoUML no muestra gráficamente los tags

Por último, el núcleo de IDM define un modelo de recolección de información mediante el registro de eventos (*logging*) basado en la herramienta de software libre **log4j**. Cada evento representa una falla en la ejecución de un control de dependencia, y para cada dependencia puede ocurrir más de una falla (por ejemplo, cuando ambos elementos relacionados no están presentes en el sistema en desarrollo).

5.2. Implementación de IDM Vista

Originado en una de las primeras pruebas de concepto realizadas para IDM, aún antes de definido el uso de XMI, IDM Vista nació con la idea de probar la integración con el proyecto de software libre Eclipse (http://www.eclipse.org).

Eclipse es un proyecto que provee una plataforma muy potente para la construcción de herramientas de desarrollo integradas, y se utilizó en forma extensa su subproyecto GEF (Graphical Editing Framework) que provee la base para crear editores de tipo componente-conector. A partir de esta infraestructura se desarrolló IDM Vista, que permite ver gráficamente los modelos de dependencia, ejecutando automáticamente los controles y resaltando gráficamente las dependencias no satisfechas.

La implementación de IDM Vista planteó múltiples dificultades, algunas propias de las limitaciones de XMI para describir los aspectos gráficos de la presentación del modelo, algunas causadas por la heterogeneidad de las diferentes tecnologías (Netbeans y Eclipse) y otras inherentes al contexto (necesidad de

utilizar el modelo de proyectos Java dentro de Eclipse).

Con respecto a la presentación gráfica de los modelos se aplicó una disposición gráfica genérica a los elementos del modelo, dado que como se explicó anteriormente, XMI no permite representar información sobre los diagramas y consecuentemente ésta no está disponible en IDM Vista. Esta disposición genérica no está optimizada para manejar en forma óptima casos de modelos muy complejos, pero tiende a funcionar bien con modelos medianos.

La integración del MDR de Netbeans (la API de XMI utilizada en IDM Núcleo) dentro de Eclipse presentó un conjunto de problemas inesperados y difíciles de resolver. Básicamente, MDR seleccionaba la clase a utilizar como implementación del analizador (parser) de XMI en forma dinámica, mediante un componente de búsqueda dinámica de implementaciones, y al ser integrado en Eclipse, este modelo simplemente no funcionaba dado que la estructura de *classloaders* (módulos responsables de la carga dinámica de clases) en Eclipse es muy diferente de la utilizada por la máquina virtual Java y por Netbeans. Esta integración requirió entonces realizar una implementación del componente de búsqueda de clases usado por el Netbeans MDR para el contexto de Eclipse. Este componente tuvo que ser configurado para ser utilizado por el IDM Núcleo cuando se utilizada dentro de Eclipse.

Finalmente, para permitir la utilización razonable de IDM dentro de Eclipse, fue necesario reemplazar algunas implementaciones de controles de dependencias por versiones apropiadas para el contexto de Eclipse. El problema de fondo es el siguiente: Si aplicamos un control de dependencia dentro de Eclipse, su ámbito de aplicación no debería ser el espacio de trabajo (workspace) completo de Eclipse, si no el proyecto en el que estamos trabajando. Por ejemplo, si el modelo describe que debe existir la clase A, deberíamos buscarla en el proyecto actual, aquél al que pertenece el modelo, no en cualquier proyecto que forme parte del espacio de trabajo, ni en la máquina virtual Java que actualmente está ejecutando Eclipse.

Esto llevó a reescribir algunas de las implementaciones de controles, aquellas que tenían que ver con la carga de clases, incluyendo no solamente los controles de clases si no los de bases de datos y tablas, que realizaban carga de controladores (drivers) JDBC (Java Database Connectivity) para la conexión con bases de datos. Afortunadamente, el diseño del núcleo de IDM (basado en el

patrón Bridge) preveía realizar sin demasiado esfuerzo el cambio en la implementación de los controles, sin embargo la implementación de los mismos requirió bastante esfuerzo, particularmente sobre la API de control de proyectos Java de Eclipse, JDT (Java Development Tools) y la de *classloaders*.

6. Arquitectura de IDM

La documentación de arquitectura de IDM está organizada de acuerdo al modelo de vistas del UML. Se destaca la vista lógica o de diseño, que es la que mejor describe a IDM, que es en sí misma una herramienta de software.

La arquitectura de IDM está basada en la necesidad de contar con un núcleo básico de funcionalidad accesible para un conjunto extensible y dispar de herramientas de usuario, como IDM Vista e IDM Consola.

6.1. Vista de Casos de Uso

Las responsabilidades del subsistema IDM Núcleo son:

1. Definir el modelo de dominio de IDM.
2. Analizar modelos de dependencia XMI.
3. Implementar las librerías de controles de dependencias.
4. Implementar la recolección de información mediante log4j.

Las responsabilidades del subsistema IDM Vista son:

1. Mostrar en forma gráfica el estado de corrección de las dependencias documentadas en un modelo.

Las responsabilidades del subsistema IDM Consola son:

1. Ejecutar por consola los controles de estado de corrección de las dependencias documentadas en un modelo.

6.2. Vista lógica

6.2.1. Modelo de solución de IDM

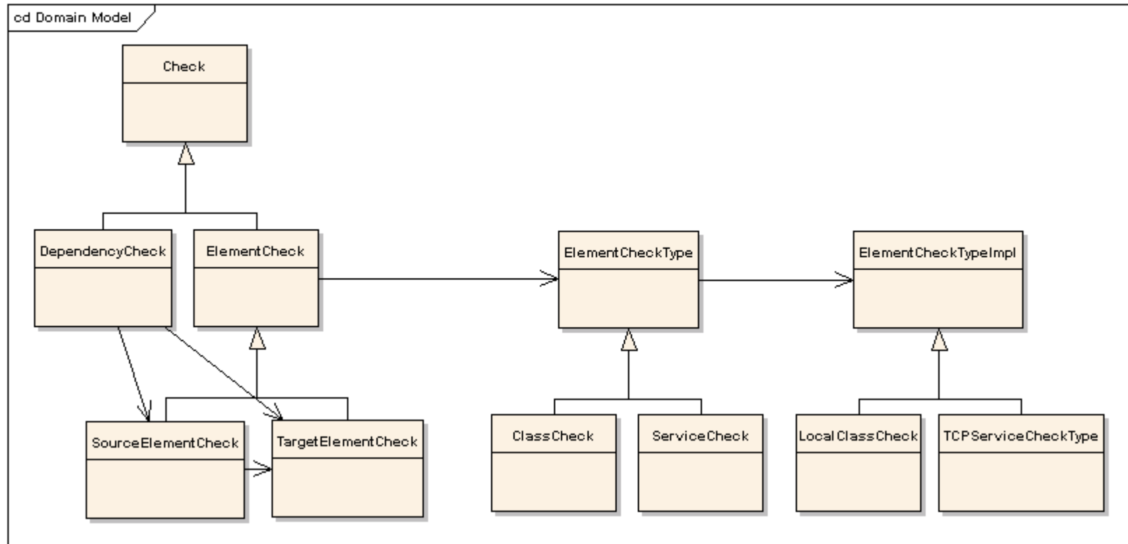

Figura 9: Modelo de solución de IDM

El diagrama muestra el modelo básico de solución de IDM Núcleo. Está basado en un modelo de dominio simple formado por las siguientes entidades:

- Check: Esta entidad describe un control capaz de dar un valor de verdad para una condicionándonos en la relación de dependencia.
- SourceElementCheck (Control de elemento origen): Control de un elemento que es el origen en la relación de dependencia, es decir que el elemento dependiente.
- TargetElementCheck (Control de elemento destino): Control de un elemento que es el destino en una relación de dependencia, es decir el elemento del cual depende el elemento origen.
- DependencyCheck (Control de dependencia): Control de una relación de dependencia. Compuesto por un control de elemento origen y un control de elemento destino. Cuando falla el control para el elemento destino, es necesario que falle el control para el elemento origen.

Más allá del modelo de dominio básico descripto, el modelo de solución está formado por dos jerarquías de clases que describen la implementación de los controles descriptos en el modelo de dominio.

La jerarquía principal del modelo, cuya clase madre es Check (Control), tiene su

comportamiento centrado en la implementación del método check() (controlar). Para permitir variar en forma independiente el algoritmo de control para cada tipo de elemento, las implementaciones del método check() están organizadas mediante el patrón de diseño *Strategy* (Estrategia) en una nueva jerarquía. De acuerdo a este patrón, existe una clase abstracta, en este caso ElementCheckType, que modela un **tipo** de control de elementos, y clases concretas herederas que aportan una implementación específica para el control de cada tipo específico de elemento.

Por ejemplo, para un elemento que representa una Clase de objeto, se asigna un control de tipo ClassElementCheck (que extiende ElementCheck) y que indica que el elemento que debe controlarse es una Clase y que el control debe hacerse en forma acorde.

Ahora bien, para cada tipo de control es necesaria una implementación específica, que a su vez puede variar en forma independiente del tipo de control de que se trate. Por ejemplo, para controlar una Clase puede se intentar cargarla en memoria en el entorno de ejecución de los controles, o buscar su archivo binario en el sistema de archivos. Así, se hace necesario volver a separar responsabilidades distintas, y encapsular el concepto de **implementación de un tipo de control de elemento**. El patrón que sigue esta parte de la solución se conoce como *Bridge* (Puente), porque establece un puente entre una jerarquía que define la interfaz para los tipos de control, y otra que define las implementaciones para los mismos. Esta estructura de la solución permite no sólo contar con múltiples implementaciones para cada control (como se verá más adelante esto es necesario en algunos casos) si no incluso compartir implementaciones entre controles de elementos distintos pero análogos.

6.2.2. Estructura de subsistemas de IDM

La estructura lógica de IDM está dada esencialmente por el patrón Capas (Layer), en este caso con dos niveles de abstracción bien definidos: El nivel superior de herramientas de nivel de usuario, y el nivel inferior de plataforma con servicios básicos prestados a las capas superiores.

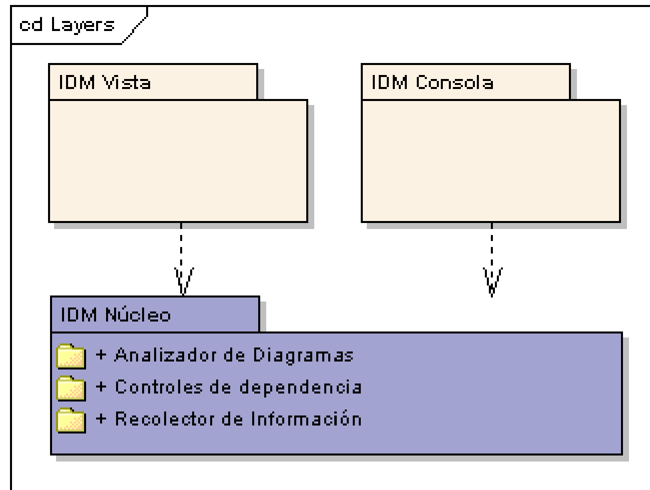

Figura 10: Estructura de subsistemas de IDM

Subsistemas de IDM y sus componentes

IDM Núcleo	IDM Vista	IDM Consola
Analizador de diagramas XMI	Ejecutor de controles	Ejecutor de controles
Controles de dependencia	Visor de diagramas	
Recolector de información	Visor de logs	

6.2.3. IDM Núcleo

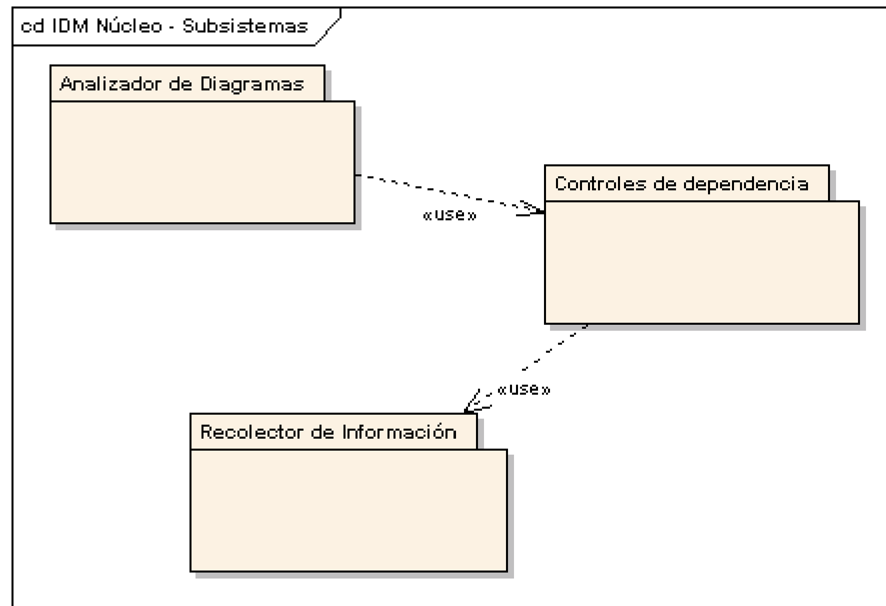

Figura 11: Subsistemas de IDM Núcleo

Analizador de diagramas

El analizador de diagramas está basado en la implementación de XMI del MDR (MetaData Repository) del proyecto NetBeans (Entorno de desarrollo Java de software libre de SUN Microsystems). La función principal del analizador de diagramas es interpretar la información contenida en el archivo .xmi, recuperando la información de cada componente (incluyendo los valores documentados mediante etiquetas UML para sus propiedades significativas) y de las relaciones entre ellos. Para cada relación de dependencia, la herramienta se ocupará de generar un módulo de control como se describe en la sección anterior.

Controles de dependencia

Este módulo se encarga de tomar como información de entrada un modelo de dependencias y generar un módulo de control para cada dependencia descrita en el modelo, los cuales permiten obtener información de corrección sobre el estado actual del sistema en desarrollo.

Los controles son componentes de software específico para cada tipo de elementos relacionados, capaces de brindar información de corrección para una dependencia en particular. Por ejemplo, si la clase A usa el servidor B, el módulo

de control de esa dependencia intenta detectar si el servidor B está accesible, por ejemplo mediante un comando que haga un “echo” (como ping), o estableciendo una conexión.

Los controles de dependencia actualmente implementados son los siguientes:

Tipo de Control	**Implementaciones del control**
Clase	System Classloader: Intenta cargar una clase en la máquina virtual utilizando el System Classloader. Buscador de clases por proyecto Eclipse: Intenta cargar la clase luego de buscarla en la estructura del proyecto dentro de Eclipse al que pertenece el diagrama.
Base de datos	JDBC (Java Database Connectivity): Intenta conectarse a la base de datos mediante un controlador JDBC, que debe estar presente en el sistema. JDBC con classloaders de Eclipse: Intenta conectarse a la base de datos mediante un controlador JDBC, que debe estar presente en el proyecto Eclipse al que pertenece el diagrama.
Tabla de base de datos	JDBC (Java Database Connectivity): Intenta conectarse a la base de datos como se describió más arriba y pide los metadatos de la tabla. JDBC con classloaders de Eclipse: Intenta conectarse a la base de datos como se describió más arriba, y pide los metadatos de la tabla.
Archivo	Sistema de archivos: Controla si existe el archivo en el sistema de archivos.
Archivo .properties	Clase Properties sobre el sistema de archivos: Carga el archivo de properties desde el sistema de archivos.
Servicio	TCP/IP Socket: Intenta conectarse al socket especificado por host y port. Clase URLConnection: Intenta abrir una conexión al url especificado por host y port.

Nota: La ejecución de los controles dentro de Eclipse requirió el desarrollo de nuevas implementaciones de controles puesto que el modelo de classloaders

(cargadores de clases, responsables de la carga de clases y otros recursos) en Eclipse es muy distinto del estándar de Java. Esta situación se agravó por la necesidad de mantener estancos los diferentes proyectos dentro del entorno de desarrollo, puesto que cada proyecto corresponde a un sistema en estudio, que debe ser evaluado en forma independiente.

Recolector de información

El recolector de información registra la información generada por los controles de dependencia. Su función es mantener una interfaz simplificada para permitir a los controles publicar la información por ellos generada.

Aunque no da información sobre el comportamiento del sistema en desarrollo permite confirmar si los componentes principales están instalados, si otros sistemas necesarios para el funcionamiento del sistema en estudio están presentes, ya sea en el mismo procesador o a través de la red, etc.

Desde el punto de vista de la implementación, aunque es factible utilizar interfaces más complejas, los controles trabajan agregando información a un **log o registro de eventos**.

Hay que tener en cuenta que la ejecución de los controles es lo que dispara la generación de información, por lo cual la recolección de información ocurre de acuerdo a la herramienta de usuario utilizada. En el caso de IDM Vista, basta visualizar un modelo para que la información de corrección sea recolectada en el log. En el caso de IDM Consola, es necesario ejecutar la aplicación. Es decir, es el usuario quien define cuándo se debe obtener la información.

Diagrama de paquetes de IDM Núcleo

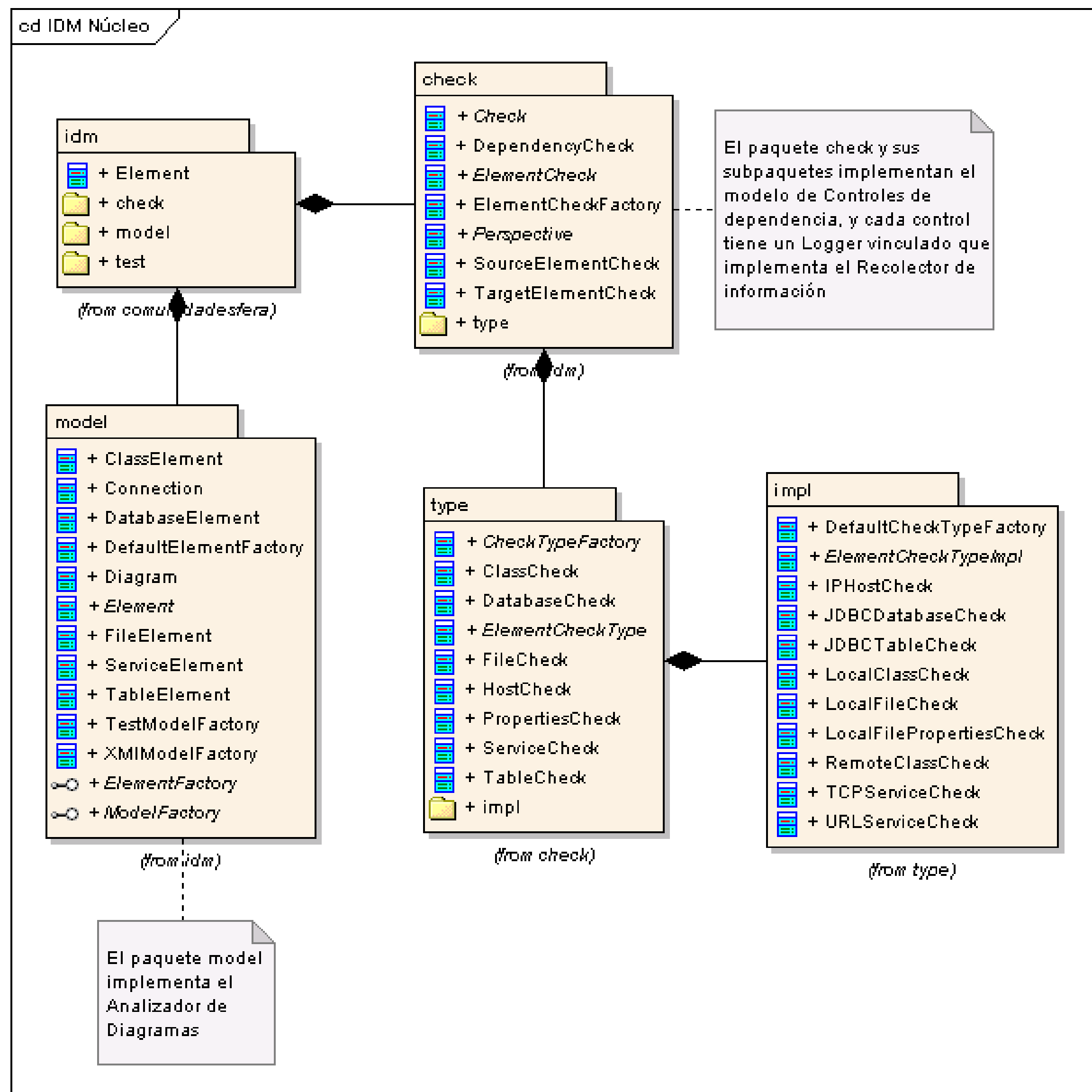

Figura 12: Diagrama de paquetes de IDM Núcleo

6.2.4.IDM Vista

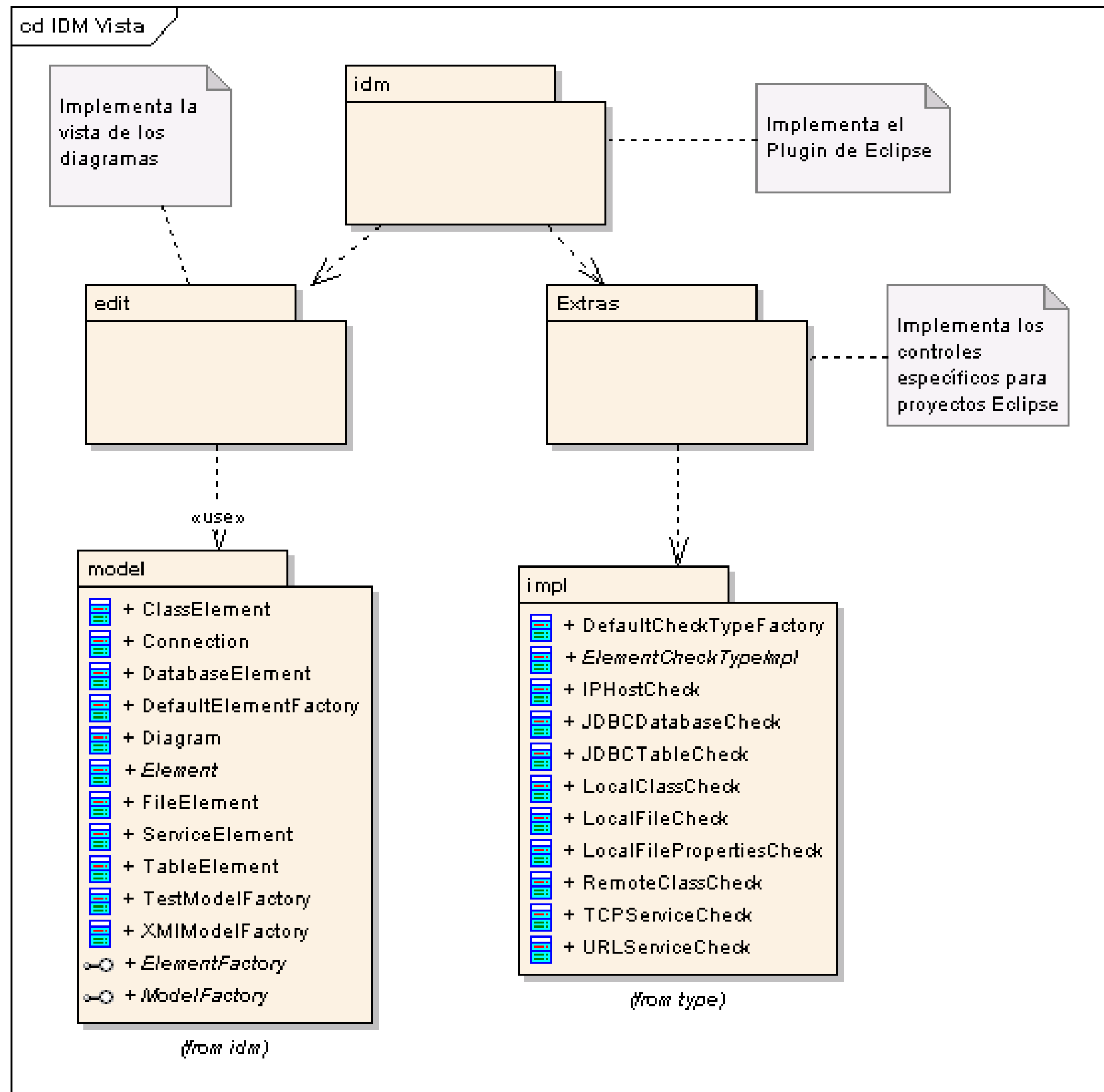

Figura 13: Diagrama de paquetes de IDM Vista

Paquetes de IDM Vista

idm: Implementa el plugin de Eclipse.

idm.edit: Implementa el visor de diagramas dentro del plugin, basado en el *framework* GEF (*Graphical Editing Framework*) de Eclipse.

idm.extras: Contiene las implementaciones de los Tipos de control de dependencias construidas para Eclipse, más algunas clases utilitarias que permiten integrar los controles al entorno de proyectos java de Eclipse.

Los paquetes **idm.model** e **idm.impl** pertenecen al núcleo de IDM.

6.3. Vista de Componentes

La herramienta IDM está dividida en dos proyectos de desarrollo principales:

IDM Núcleo (idm.core.jar): Contiene el subsistema completo de Controles de Dependencia, más el Analizador de diagramas, más los módulos básicos del Recolector de Información. Está empaquetado en un archivo .jar.

IDM Vista (idm.jar): Contiene la clases de presentación gráfica de los modelos, implementado como un plugin de Eclipse, más las implementaciones propias de Eclipse de los controles. Contiene y utiliza el **idm.core.jar** para realizar todas las tareas básicas de análisis de modelos, control de dependencias y recolección de información, presentando los resultados mediante la interfaz gráfica (tanto el diagrama del modelo como el log de los controles de dependencia).

Depende de los siguientes *plugins* de Eclipse: GEF (Graphical Editing Framework), JDT (Java Development Tools), JDT Core y Logging (com.tools.logging).

IDM Consola (idm.core.jar): Incluido en idm.core.jar por simplicidad. Se puede ejecutar directamente desde el jar en línea de comandos.

Todas están desarrolladas como aplicaciones Java puras, sin dependencias de plataforma, aunque IDM Vista requiere integrarse con Eclipse, que es parcialmente dependiente de la plataforma.

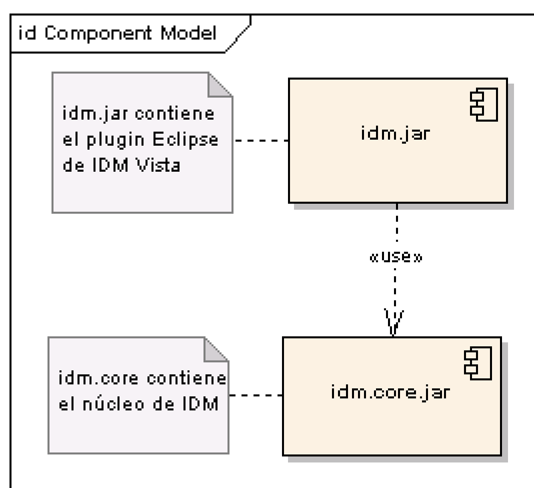

Figura 14: Diagrama de componentes de IDM

IX. Conclusiones y perspectivas

Estamos convencidos de que los problemas planteados en este trabajo relativos al control de las dependencias entre componentes de un sistema son comunes a muchos sistemas modernos. También pensamos que las dependencias entre esos componentes pueden controlarse mejor a través de lenguajes de programación capaces de expresar aspectos de arquitectura.

Existen ya herramientas de modelado de arquitectura de software que permiten trabajar sobre un sistema ya construido e incluso obtener información sobre la arquitectura en forma dinámica a partir de la ejecución del sistema, aunque todavía dependen de ciertos supuestos que deben hacerse explícitas sobre sus componentes (ver [Garlan 1997] y [Garlan 2004]).

Como plantea Frederick Brooks (ver [Brooks 1987]), el uso de lenguajes de más alto nivel permitió en su momento resolver muchos de los problemas no esenciales en el desarrollo de sistemas y resultó en un aumento de la eficiencia de los desarrolladores. Lo mismo podría ocurrir para la problemática de arquitectura, pero también parece difícil evitar que las diferencias arquitectónicas aparezcan cuando se construyen sistemas a partir de componentes de muy distinto origen (ver [Garlan 1995]).

Creemos que un sistema de software es básicamente información empaquetada, ya sea en forma de código, modelos o documentos. Nos relacionamos con esta información de diferentes maneras, como desarrolladores o usuarios, autores o revisores, pero es materia fundamental de la ingeniería de software el hacerla accesible para los interesados de la forma más cercana posible. En este espíritu, nuestra herramienta de control de dependencias busca facilitar la creación de los diagramas y la asignación de significado a los mismos, integrándolos con el código del sistema. Creemos también que no debe minimizarse la importancia de motivar a los participantes a realizar las tareas de documentación, facilitándoles la tarea y mostrándoles resultados inmediatos para su esfuerzo.

Quedan abiertos múltiples caminos de trabajo a futuro. Particularmente, nos parece valioso el desarrollo de una extensión de la herramienta IDM para controlar dependencias en tiempo de ejecución. Esto permitiría contar con información de consistencia durante la ejecución del sistema en estudio, mostrando un perfil dinámico de evolución de las dependencias a través del

tiempo.

Lo interesante de esta modalidad es que detectaría inconsistencias aún cuando no se evidencie una falla del sistema. El control de dependencias en tiempo de ejecución permitiría detectar problemas en forma temprana, para poder prevenir futuras fallas. En particular, permitiría ejecutar estos controles en ambientes productivos mejorando la calidad de los servicios prestados mediante la prevención de fallas.

Para el control en tiempo de ejecución, sería necesario integrar los controles afectando lo menos posible al sistema ya instalado (por ejemplo, sin requerir recompilarlo). Para resolver este problema, nuestra propuesta sería utilizar la tecnología de **Programación Orientada a Aspectos**, fruto de la investigación de Xerox pero cuya implementación original, AspectJ, fue donada y se encuentra actualmente formando parte del proyecto Eclipse. Esta tecnología permite agregar funcionalidad a un programa Java sin modificar su código, por ejemplo agregando la ejecución de los controles de dependencia cada vez que se invoque un método de una clase dependiente en el sistema en estudio (ver [AspectJ 1998]).

Para terminar, nos parece necesario destacar la importancia de mantener la brecha entre los modelos y la realidad. Este salto en el nivel de abstracción que es la esencia de todo modelo debe existir para limitar la cantidad de información presente en ellos. Nos parece fundamental que esa distancia se mantenga entre el modelo y la realidad que lo origina, para así mantener al modelo más cerca de sus autores y lectores. De esta manera, no sólo se humaniza esa información (aunque muchos piensen lo contrario de los modelos), si no que se evita que el modelo quede rápidamente desactualizado. Como dice Borges en “Funes el memorioso”: **“Dos o tres veces había reconstruido un día entero; no había dudado nunca, pero cada reconstrucción había requerido un día entero”** [Borges 1944].

X. Agradecimientos

A mi familia (Fátima, Pimpi, Pablo, Eva, Pole y Elisa); a Sergio Villagra, Osvaldo Carrizo, Marcela Menegotti, Mariano Tugnarelli, Gastón Real, Gabriel Mansi, Pablo de Marcos, Cecilia Ruz, Natalia Lehmann, Claudio Figuerola, Sergio Romano, Sebastián Ismael, Maximiliano Pretzel, Ana Wolosiuk, Carlos Lizarralde, Daniel Sabella Rosa y Rodrigo Campos Alvo.

XI. Anexo I: Ejemplo de modelo integrado de dependencias

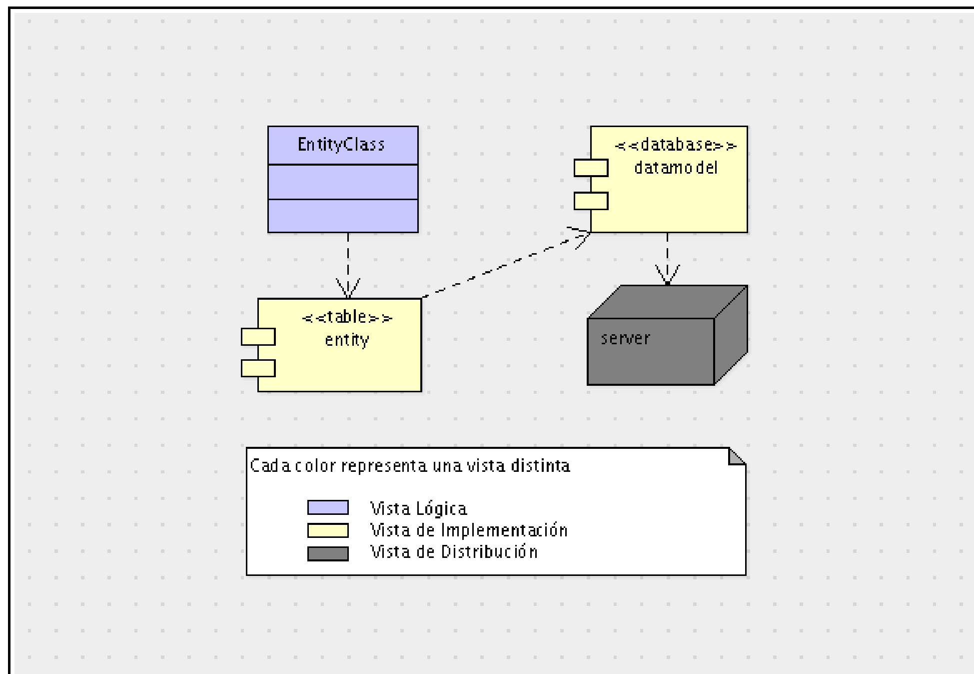

Figura 15: Ejemplo de modelo integrado de dependencias en ArgoUML

XII. Anexo II: Ejemplo de modelo para una aplicación real

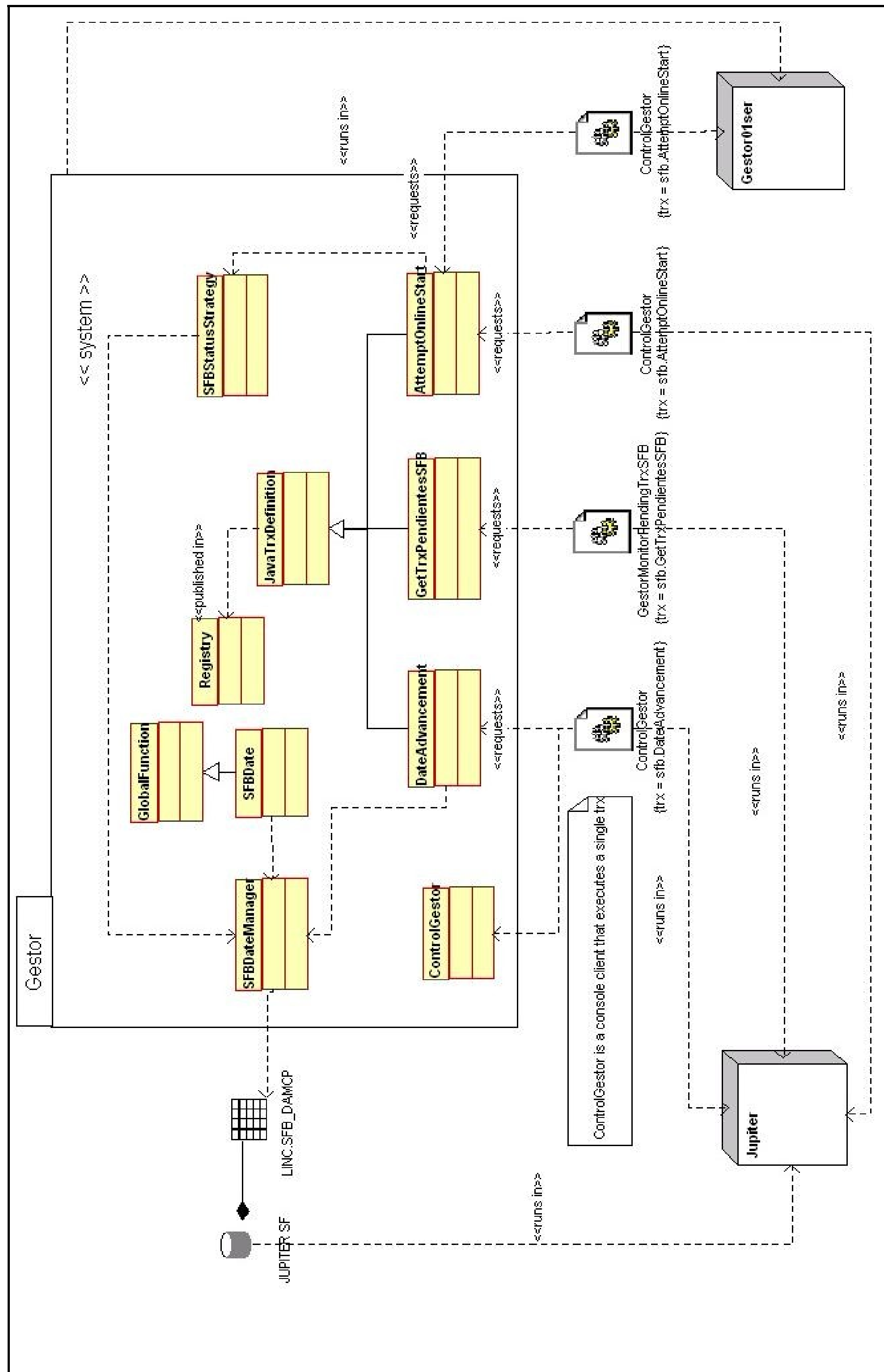

Figura 16: Ejemplo de diagrama de dependencias para una aplicación real

XIII.Indice de Figuras

Figura 1: Ejemplo de trazabilidad, una clase que participa de la Implementación de un caso de uso..11

Figura 2: Diagrama de dependencias entre elementos de distinta vista. Una Clase (elemento de la vista lógica), un archivo de configuración que fija los valores de sus propiedades (elemento de la vista de componentes) y un servidor físico (elemento de la vista de distribución)...17

Figura 3: Estructura de módulos de IDM. En gris están los módulos preexistentes, en violeta las librerías y en beige las aplicaciones..33

Figura 4: Creación de un modelo con una herramienta UML...............................35

Figura 5: Ejemplo de uso de IDM Vista..36

Figura 6: Modelo de casos de uso de documentación con IDM............................37

Figura 7: Proceso de control de dependencias usando IDM................................38

Figura 8: Ejemplo de uso de etiquetas UML (tags o tagged values) desarrollado con Enterprise Architect porque ArgoUML no muestra gráficamente los tags.....44

Figura 9: Modelo de solución de IDM...48

Figura 10: Estructura de subsistemas de IDM..50

Figura 11: Subsistemas de IDM Núcleo..51

Figura 12: Diagrama de paquetes de IDM Núcleo...54

Figura 13: Diagrama de paquetes de IDM Vista..55

Figura 14: Diagrama de componentes de IDM..57

Figura 15: Ejemplo de modelo integrado de dependencias en ArgoUML.............61

Figura 16: Ejemplo de diagrama de dependencias para una aplicación real 62

XIV.Glosario

AOP: *Aspect Oriented Programming,* tecnología que permite el agregado de funcionalidad a un sistema orientado a objetos existente sin modificar su código.

Anotaciones: Mecanismo de extensión del lenguaje Java que permite agregar meta información sobre los componentes de un sistema. Por ejemplo, la anotación @Table permite identificar en una clase a la tabla en la que se persisten las instancias de la misma.

AspectJ: Proyecto original de Xerox donde se desarrolló la teconología AOP. Originalmente implementado como una extensión al lenguaje Java con su propio compilador. Actualmente ofrece también una alternativa Java pura usando anotaciones que no requiere el compilador AspectJ.

Control de Dependencia: Un control que valida una relación de dependencia, en base a validar cada uno de los elementos. Ver control de elemento.

CORBA: *Common Object Request Broker Architecture*, arquitectura estándar de componentes de acceso a objetos remotos multilenguaje, capaz de proveer interconexión basada en objetos entre sistemas heterogéneos.

Dependencia: Una relación entre dos componentes A,B tal que si B sufre un cambio, A puede sufrir un cambio.

Eclipse: Entorno de desarrollo integrado Java, software libre creado por IBM. Incluye el proyecto AspectJ, donado por Xerox.

Elemento: Un componente en un modelo integrado de dependencias que puede estar relacionado con otros mediante dependencias.

Modelo integrado de dependencias: Un modelo de clases o componentes de UML (diagrama del tipo componente-conector) en formato XMI, donde los componentes son elementos de las distintas vistas (tablas, clases, archivos, procesadores lógicos, procesadores físicos, etc.), y las relaciones son de dependencia. Se guardan en archivos XMI.

Netbeans: Entorno de desarrollo integrado Java, software libre creado por SUN Microsystems. Incluye una implementación de XMI.

OMG: *Object Management Group*, consorcio encargado del mantenimiento de los estándares UML, XMI y CORBA.

Tipo de Control de Elemento: Describe el tipo de control que se va a realizar. Para cada tipo de elemento específico existe un tipo de control. Por ejemplo, para un elemento de tipo Clase, existe un control de clase.

UML: Unified Modelling Language

XMI: XML Metadata Interchange, formato estándar del OMG para intercambio de modelos de metadatos, incluyendo modelos de UML.

XV.Referencias

[Aldrich 2004] Aldrich, Jonathan, Garlan, David, Schmerl, Bradley, Tseng, Tony, **Modeling and implementing software architecture with acme and archJava,** Conference on Object Oriented Programming Systems Languages and Applications, Vancouver, Canada, 2004.

[AspectJ 1998] AspectJ, Team, **"Introduction to AspectJ"**, Xerox Corporation, Palo Alto Research Center, 1998.

[Booch 1999] Booch, Grady, Jacobson, Ivar y Rumbaugh, James, **The Unified Modelling Language User Guide**, Addison-Wesley, 1999.

[Borges 1944] Borges, Jorge Luis, **“Funes el memorioso”, en Ficciones (Artificios)**, Sur, Buenos Aires, 1944.

[Brooks 1987] Brooks, Frederick P., **No silver bullet, Essence and accidents of software engineering**, IEEE Computer magazine, Volumen 20, 4º edición, Abril 1987.

[Clements 2003] Clements, Paul, y otros, **Documenting Software Architecture: Views and Beyond**, SEI Series in Software Engineering, Addison-Wesley, 2003.

[Clements 1996] Clements, Paul, Shaw, Mary, **"Three Patterns That Help Explain the Development of Software Engineering"**, Position paper for Dagstuhl Workshop on Software Architecture, 1996.

[Fontedevila 2003] Fontdevila, Diego, Real, Gastón, **“Documentación de arquitectura, Tras la documentación del proceso de decisión”**, Facultad de Ingeniería, Universidad de Buenos Aires, 2003.

[Garlan 1997] Garlan, David, Monroe, Robert, Wile, David, “**Acme: an architecture description interchange language”,** IBM Centre for Advanced Studies Conference archive**.** Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, Page: 7, 1997.

[Kruchten 1995] Kruchten, Philippe, **"Architectural Blueprints - The "4+1" View Model of Software Architecture"**, IEEE Software 12 (6), 1995.

[Meyer 1985] Meyer, Bertrand, **Object-Oriented Software Construction**, Prentice-Hall, 1985, 2da. Edición 1997.

[Robbins 2000] Robbins, Jason E., Redmiles, David F., **Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML**, University of California, Irvine, 2000.

[Seidewitz 2003] Seidewitz, Ed, "**What Models Mean"**, Model-Driven Development, IEEE Software, September/October 2003.

[SUN 2003] SUN Microsystems, **Architecting and Designing J2EE Applications Course Student Manual**, SUN Microsystems Educational Services, 2003.

XVI.Bibliografía

Aldrich, Jonathan, Chambers, Craig, Notkin, David, **ArchJava: connecting software architecture to implementation**, Department of Computer Science and Engineering, University of Washington, Seattle, 2002.

Bachman, Felix, Bass, Len, **Architecture Based Design Method Introduction**, Tutorial presentation, 2000, http://www.sei.cmu.edu/plp/symposium00/ABD/intro/index.htm .

Bader, Atef, Elrad, Tzilla, Filam, Robert E., **"Aspect-oriented programming: Introduction"**, Communications of the ACM, Volume 44, Issue 10, 2001.

Bass, Len, Clements, Paul, Kazman, Rick, **Software Architecture in Practice**, SEI Series in Software Engineering, 1998.

Booch, Grady, Jacobson, Ivar, Rumbaugh, James**, The Unified Modeling Language User Guide**, Addison-Wesley, 1999.

Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, Stal, Michael, **Pattern-Oriented Software Architecture, Volume 1: A System of Patterns**, 1996.

Hessen, J**., Teoría del conocimiento**, Biblioteca contemporánea, Losada, 1938.

Clements, Paul, Shaw, Mary, **"Three Patterns That Help Explain the Development of Software Engineering"**, Position paper for Dagstuhl Workshop on Software Architecture, 1996.

Clements, Paul, et al, **Documenting Software Architecture: Views and Beyond**, SEI Series in Software Engineering, Addison-Wesley, 2003.

Eco, Umberto, **Cómo se hace una tesis: Técnicas y procedimientos de estudio, investigación y escritura**, Biblioteca de educación - Herramientas universitarias, Gedisa, 1977.

Gamma, Eric, Helm, Richard, Johnson, Ralf, Vlissides, John, **Design Patterns, Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995.

Garland, David, Allen, Robert, Ockerbloom, John, **”Architectural Mismatch, or Why it's hard to build systems out of existing parts”**, Computer Science

Department, Carnegie Mellon University, Pittsburgh, 1995.

Garlan, David, Shaw, Mary, **"An Introduction to Software Architecture"**, V. Ambriola and G. Tortora (Ed.), Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, 1993.

Garlan, David, Shaw, Mary, **"Software Architecture: Perspectives on an Emerging Discipline"**, Prentice Hall, 1996.

Grosso, William, **"Aspect-Oriented Programming and AspectJ"**, Dr. Dobbs Journal, 2002.

Kruchten, Philippe, **"Architectural Blueprints- The "4+1" View Model of Software Architecture"**, IEEE Software 12 (6), 1995.

Hannemann, Jan, **"Aspect-Oriented Design Pattern Implementations, Language-dependence of Design Patterns"**, University of British Columbia, 2004.

Hannemann, Jan, Kiczales, Gregor**, "Design Pattern Implementation in Java and AspectJ"**, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.

Hudson, Randy, **“Create an Eclipse based application using the Graphical Editing Framework”**, How to get started with GEF, IBM Developer Works, 2003, http://www-128.ibm.com/developerworks/opensource/library/os-gef/ .

Jacobson, Ivar, et al, **Object Oriented Software Engineering: A Use Case driven approach**, Addison-Wesley, 1992.

Lee, Daniel, **"Display a UML Diagram using Draw2D"**, Eclipse Corner Article, 2003.

Ramnivas, Laddad, **AspectJ In Action**, Manning, 2003.

SUN Microsystems, **Architecting and Designing J2EE Applications Course Student Manual with Instructor notes**, SUN Microsystems Educational Services, 2003.

Ursua, Nicanor, **Filosofía de la ciencia y metodología científica**, Desclee de

Brouwer, 1983.