UNIVERSIDAD PONTIFICIA DE SALAMANCA
Facultad de Informática

TESIS DOCTORAL

Meta-Especificación y Catalogación de Patrones de Software con Lenguajes de Dominio Específico y Modelos de Objetos Adaptativos: Una Vía para la Gestión del Conocimiento en la Ingeniería del Software

Autor: **León E. Welicki**
Directores: **Dr. Juan Manuel Cueva Lovelle**
Dr. Luis Joyanes Aguilar

Madrid 2006

A mi familia

Resumen

En los últimos años, la comunidad de patrones ha crecido a un ritmo vertiginoso, produciendo una gran cantidad de lenguajes y sistemas de patrones aplicables a diversos dominios, con distintos grados de éxito y popularidad. A modo de ejemplo de este crecimiento proponemos un simple pero provocador ejercicio: sumar la cantidad de patrones publicados en tres de los libros más influyentes de la literatura de patrones, "*Design Patterns: Elements of Reusable Object Oriented Software*" [GoF95], "*Pattern Oriented Software Architecture, Volume 1*" [POSA96] y "*Patterns of Enterprise Application Architecture*" [Fowler02]. El número total es 100 (un número bastante grande de elementos para mantener en la memoria). Para complicar aún más las cosas, en cada uno de los libros mencionados se describe a los patrones utilizando diferentes estructuras (plantillas) y los ejemplos están escritos en distintos lenguajes de programación sobre distintas infraestructuras de implementación. Esto produce una diferencia de impedancia considerable entre todos ellos.

Existen varias "llamadas a la acción" en distintas obras de referencia de la literatura de patrones ([POSA96], [GoF95], [Berczuk94]) que hacen evidente la necesidad de una forma estandarizada de describir, clasificar y almacenar a los patrones junto con una forma sencilla de compartir y utilizar estas descripciones. En la actualidad existen varios enfoques que resuelven el problema en forma parcial, pero en todos los casos tienen limitaciones e inconvenientes recurrentes. Esto convierte a la descripción y catalogación de patrones en un problema de primera clase que todavía no ha sido resuelto en forma satisfactoria [Welicki04c] [Welicki05] [Welicki06].

En esta tesis se propone un modelo de meta-especificación y catalogación de patrones y conceptos como respuesta a las necesidades de gestión del conocimiento en éste ámbito de la ingeniería del software. La arquitectura general de la solución propuesta se compone de un lenguaje de meta-especificación para describir a los patrones a un alto nivel de abstracción, un catálogo de patrones creados con ese lenguaje, una infraestructura de catalogación y una herramienta de explotación del catálogo. Para verificar la factibilidad de la solución propuesta hemos creado un prototipo y lo hemos evaluado respecto a otras soluciones y enfoques existentes para demostrar que el modelo propuesto supera las dificultades recurrentes encontradas en otros enfoques.

Los patrones emergen de la experiencia [Welicki06]. Tienen un ciclo de vida [Welicki06b] que comienza con conocimiento tácito en la cabeza de una persona o grupo de individuos y termina con una descripción explícita y rígida de ese conocimiento que puede ser compartida. El modelo propuesto en esta tesis produce un cambio significativo en este ciclo haciéndolo más dinámico e interactivo, sentando las bases para la evolución y generación de conocimiento a partir de interacciones entre los miembros de una comunidad (que puede estar distribuida geográficamente). A partir de esta generación de conocimiento se promueve el refinamiento y mejora continua de los patrones, facilitando así su evolución constante para adaptarse a cambios no previstos en su contexto.

Palabras Claves: patrones, Meta, meta-especificación, clasificación, catalogación, catálogo de patrones, navegador de patrones, visualización de patrones, ingeniería de software orientada por patrones, lenguajes de dominio específico, modelos de objetos adaptativos, gestión del conocimiento.

Abstract

In the last years, the patterns community grew at a very fast pace. A lot of pattern languages and pattern systems emerged. Some of them are very popular, some of them are not. To show the tip of the iceberg, we propose a very simple, yet mind provoking, exercise: let's sum the number of patterns in three reference and mainstream books, "*Design Patterns*" [GoF95], "*Pattern Oriented Software Architecture, Volume 1*" [POSA96] and "*Patterns of Enterprise Application Architecture*" [Fowler02]. The final number is 100… quite big to remember all! To worsen things, each pattern language is described using different formal elements (templates) and the samples are written with different programming languages. This produces a considerable impedance mismatch among all of them.

We found some "calls to action" in several reference books in the patterns literature ([POSA96], [GoF95], [Berczuk94]) that made evident the need for a standardized way to describe, classify, share and store patterns and a simple way to use these descriptions. Some patterns description, cataloging and browsing approaches that partially solve the problem exists, but all of them fail in some aspects, presenting recurring problems. This makes patterns description and cataloging a first class problem that must be solved and hasn't been successfully addressed yet [Welicki04c] [Welicki05] [Welicki06].

In this thesis a patterns meta-specification and cataloging model is proposed as a means to provide a response to the knowledge management needs in this area of software engineering. The high-level architecture of our solution is composed of a meta-specification language for describing patterns at a very high level of abstraction, a catalog of patterns described using that language, a cataloging infrastructure and a web-based browser for exposing the contents of the catalog. In order to verify the feasibility of our proposal we built a research prototype and evaluated it regarding other existing approaches for demonstrating that our model overcomes the recurring problems that we have found in them.

Patterns emerge from experience [Welicki06]. They have a life-cycle [Welicki06b] that begins with tacit knowledge in the head of an individual or a group and ends with an explicit (and rigid) shareable description of that knowledge. The model proposed in this thesis produces important changes in this cycle, making it more dynamic and interactive establishing the foundations for knowledge evolution and generation from interactions among members of a community. This knowledge generation promotes continuous improvement and refinement of the patterns, making easier their constant evolution in order to adapt to changes in their context and environment.

Keywords: patterns, Meta, meta-specification, cataloging, classification, patterns catalog, patterns navigator, patterns browser, pattern oriented software engineering, domain specific languages, adaptive object model, knowledge management.

Agradecimientos

Una tesis es un largo camino personal, que se comienza con mucha ilusión e incertidumbre, se transita con mucho esfuerzo y contratiempos y finalmente se termina con agotamiento.

En mi caso, he tenido la suerte de contar con la mejor compañía en ese largo y arduo camino. Puede decirse que he viajado con ventaja, porque he contado con una compañera excepcional: mi esposa, amiga y compañera Mara, que ha estado siempre presente para alentarme, dándome las fuerzas y el apoyo necesarios para poder llegar en forma exitosa al final del camino, esforzándose tanto como yo. Esta tesis es tan suya como mía, dado que a lo largo de estos años hemos trabajado juntos como un equipo.

Durante el camino fuimos bendecidos con la llegada de Daniel (nuestro primer hijo), que es la luz de nuestros ojos y la razón de nuestro existir. Desde que llegó nos alegra (aún más) cada instante de nuestras vidas con sólo evocar su existencia. Esta tesis (y todo lo que hacemos y hagamos su madre y yo) es para él.

Toda tesis doctoral tiene un director. En mi caso, he contado con dos y no conforme con eso, ambos son figuras de excepción. Este trabajo no podría haberse realizado sin la inestimable ayuda de mis directores, el Profesor Dr. Juan Manuel Cueva Lovelle y el Profesor Dr. Luis Joyanes Aguilar, a quienes agradezco su esfuerzo y dedicación para que este trabajo llegue a buen puerto.

Como decía al inicio de este apartado la tesis es un camino. Este camino es parte del camino de la vida. Por eso quiero agradecer especialmente a mis padres por haberme dado los valores y herramientas necesarios para transitar ese camino con seguridad y por continuar enseñándome y guiándome cada día de mi vida. Pero por lo que más agradecido les estaré siempre es por haberme hecho el mejor regalo que se puede pedir: mi hermano Federico, quien además de mi hermano es mi amigo del alma.

A mi abuela Sabina, por darme la ilusión de que nada era imposible. A mi tío José, por regalarme el amor por el conocimiento. A mi abuela Bety, por regalarme el amor por la vida. A mi abuelo León, por su nombre y su guía.

A Mara y Daniel nuevamente, por estar conmigo y ser la parte más hermosa de mi vida.

A todos los que de alguna manera han colaborado en el desarrollo de este trabajo.

Muchas gracias.

Madrid, Octubre 2006.

Indice Resumido

Indice Extendido

Indice de Figuras

Indice de Tablas

Parte I
Planteamiento del Problema

"Si supiéramos lo que estamos haciendo,
no sería investigación"
Albert Einstein

Capítulo 1
Introducción

"Lo último que uno sabe
es por donde empezar"
Blaise Pascal

A lo largo de este capítulo se presentan la justificación y los principales objetivos buscados en el desarrollo de esta tesis doctoral, estableciendo un marco de requisitos generales a cumplir en la tesis enunciada en esta memoria y posteriormente demostrada mediante la creación y evaluación de una serie de prototipos.

Comenzamos el capítulo con una introducción al dominio de nuestra investigación. Continuamos con el planteamiento del problema, la hipótesis de partida y los objetivos establecidos para la investigación. Finalmente incluimos las principales aportaciones originales del modelo desarrollado durante la investigación y presentado a lo largo de esta memoria.

❖ ❖ ❖

1.1 Introducción

El software es el gran protagonista de los últimos (y de los próximos) lustros. Nuestra civilización "corre" sobre software[1]. Está presente en gran parte de los aspectos de la vida cotidiana: en los electrodomésticos, en los televisores, en las centrales de las empresas que proveen los principales servicios a los ciudadanos, en la gestión de las compañías privadas, etc. Por esto, no es casual que en el prestigioso *Wall Street Journal* [WSJ] se incluyera la frase "las ideas y los descubrimientos tecnológicos son los conductores del crecimiento económico" [Pressman02].

La construcción de software es una disciplina moderna que está en constante evolución, en búsqueda de un estado de madurez que sea comprehensivo respecto a su amplio espectro de aplicación. La reutilización promete ser una de las vías hacia ese anhelado estado, a través de una correcta gestión de las experiencias y conocimiento de ingenieros expertos. Los patrones [AIS77] [Alexander79] permiten reutilizar conocimiento y experiencia, convirtiéndose en el medio ideal (y ampliamente aceptado por la comunidad) para realizar esta gestión. Cada patrón es una regla de tres partes, que expresa una relación entre un contexto, un problema y una solución [Alexander79]. A esto se suma el espíritu pragmático de la comunidad de patrones [Hillside], que está en constante crecimiento a partir de reunir y destilar experiencias documentadas de ingenieros de todo el mundo[2].

En los últimos años, la comunidad de patrones creció a un ritmo vertiginoso, produciendo una gran cantidad de lenguajes y sistemas de patrones aplicables a diversos dominios. A modo de ejemplo, proponemos un simple pero provocador ejercicio: sumar la cantidad de patrones publicados en tres de los libros más influyentes de la literatura de patrones, "*Design Patterns: Elements of Reusable Object Oriented Software*" [GoF95], "*Pattern Oriented Software Architecture, Volume 1*" [POSA96] y "*Patterns of Enterprise Application Architecture*" [Fowler02]. El número total de patrones es 100 (un número bastante grande de elementos para mantener en la memoria). Para complicar aún más las cosas, los patrones en cada uno de los libros mencionados se describen utilizando diferentes estructuras [C2PatternForm] y los ejemplos están escritos en distintos lenguajes de programación[3] sobre distintas infraestructuras de implementación. Esto produce una diferencia de impedancia considerable entre todos ellos.

[1] "Our civilization runs on software" (texto original). Esta frase atribuida a Bjarne Stroupstrup [Stroupstrup] (creador del C++) es frecuentemente citada por Grady Booch [Booch].

[2] Dicha experiencia se recoge y destila en los "writers workshop" que se realizan en las diversas variantes de las conferencias Pattern Language of Programs (PLoP) [PLoP]. Existen varios tipos de PLoP emplazados en distintos sitios: PLoP (Illinois, USA), EuroPLoP (Irsee, Alemania), VinkingPLoP (Escandinavia), SugarLoafPLoP (Sudamérica) y ChilliPLoP (Arizona, USA). Las más representativas son PLoP y EuroPLoP. A partir de las actas de estas conferencias se han publicados varios libros de la serie PLOPD (Pattern Languages of Program Design).

[3] Por ejemplo, en [GoF95] el código de ejemplo está escrito en C++ y Smalltalk, en [POSA96] en C++ y en [Fowler02] en C# y Java.

En la literatura oficial sobre patrones ([GoF95], [Berczuk94], [POSA96]), es evidente la necesidad de una forma estándar de representación y almacenamiento de los patrones [Welicki06]. Esto convierte a la descripción y catalogación de patrones en una necesidad de primer nivel. También hacen evidente la importancia de este asunto los reiterados esfuerzos de las principales casas proveedores de software y de prestigiosas universidades y centros de investigación para resolver este problema[4]. [Welicki05] [Welicki04c]

Un interesante caso de uso y "llamada a la acción" para el desarrollo de un lenguaje de meta-especificación, un catálogo de patrones y una herramienta de visualización se incluye en el último capítulo de la influyente obra "*Pattern Oriented Software Architecture, Volume 1*" [POSA96], donde se discute sobre el "el futuro de los patrones":

> *"…En conclusión, el uso exitoso de patrones requiere de las capacidades intelectuales del desarrollador de software. Creemos que un navegador de patrones bien diseñado o una herramienta para la Web pueden ser mucho más eficientes en ayudar a los desarrolladores a encontrar y utilizar patrones de lo que puede ser un entorno totalmente integrado con soporte de patrones… "*

En la actualidad, existe un gran número de herramientas que permiten definir y navegar un catálogo de patrones, pero todas tienen falencias y carencias. En la mayoría de los casos el lenguaje de descripción utilizado no permite especificar el nivel de conocimiento (parte literaria) del patrón en forma flexible, la descripción del comportamiento está acoplada a un lenguaje de programación o entorno de ejecución y no permite describir conceptos de soporte al patrón (por ejemplo, categorías, refactorizaciones, autores, conceptos de orientación a objetos, etc.), entre otros problemas. Respecto a la visualización, en la mayoría de los casos se impone al usuario la forma de ver y navegar los patrones en función de las necesidades e intereses del publicador de la información. En el **capítulo 10** de esta tesis se analizan en detalle varias herramientas comerciales e iniciativas de la comunidad investigadora y se exponen las carencias de cada una de ellas.

1.1.1 Sobre esta Tesis

En la presente tesis se traza y explora el dominio de la meta-especificación y catalogación de patrones [Welicki04c] [Welicki05] [Welicki06] [Welicki06b], como respuesta al problema de describir, catalogar y compartir patrones de software. Se crea una infraestructura conceptual y tecnológica para describir patrones a un alto nivel de abstracción, independiente de la plantilla [C2PatternForm] utilizada para su descripción y de la plataforma y lenguaje de programación utilizada para su implementación, para su posterior utilización en forma organizada, facilitando la compartición y transmisión de conocimientos y experiencias entre ingenieros de software.

[4] Aún así, este problema no ha sido resuelto todavía en forma exitosa en ningún caso, como demostraremos más adelante en esta tesis.

La investigación realizada se inscribe en un marco interdisciplinario que incluye a la ingeniería del software, gestión del conocimiento e ingeniería Web. En el caso de la primera, nos centraremos en los patrones [AIS77] [Alexander79] de software [GoF95] [POSA96], lenguajes de dominio específico [Czarnecki00], modelos de objetos adaptativos [AOM] y sistemas emergentes [Johnson01]. Respecto a la gestión del conocimiento [Joyanes03], nos centramos en la compartición de conocimientos de ingeniería del software en cualquier lugar, en cualquier momento y con cualquier dispositivo [Joyanes03] mediante la creación de un catálogo de patrones y conceptos y de una infraestructura de catalogación. En este contexto es importante mencionar que los patrones son una vía para la gestión del conocimiento [Welicki05d] [Welicki06b]. Finalmente respecto a la ingeniería Web [Cueva03] [Murugesan99] [Joyanes03b], el foco está en la creación de una herramienta de visualización del catálogo de patrones y en la compartición de los contenidos del catálogo a través de interfaces desacopladas basadas en servicios [W3C04b].

A lo largo de esta tesis se estudia en profundidad el estado del arte en los temas relacionados con los patrones, niveles meta, lenguajes de dominio específico, modelos de objetos adaptativos y sistemas emergentes. Se prosigue con un análisis del estado del arte de los enfoques y soluciones existentes para el problema planteado. Luego, se propone una solución al problema de la descripción, catalogación y compartición de patrones de software mediante la definición de un lenguaje de meta-especificación de patrones (EML), la creación de un catálogo de patrones y conceptos escritos utilizando este lenguaje y el diseño y construcción de una herramienta para explotar el catálogo a través de la Web utilizando un navegador o consumiendo servicios Web (en el contexto de una arquitectura orientada a servicios [Helland04] [SW04]).

1.2 Planteamiento del Problema

Los patrones son una disciplina de resolución de problemas reciente en la ingeniería del software que ha emergido de la comunidad de orientación a objetos. Los patrones tienen raíces en muchas áreas, incluyendo "*literate programming*" [Knuth92] y más notablemente en el trabajo de Christopher Alexander [Alexander79] [AIS77] en planeamiento urbanístico y arquitectura civil. Se puede decir que los patrones son una forma empírica de gestión de conocimiento [Welicki05d] [Welicki06b]. En la actualidad existen cientos de lenguajes y sistemas de patrones (a modo de ejemplo podemos citar [GoF95], [POSA96], [Fowler96], [Fowler02], [BMMM98] y [Larman99]). Cada uno es descrito utilizando una estructura particular y sus ejemplos de implementación (si existen) están orientados a una tecnología concreta, aún cuando la idea que intentan expresar es muy general. La gran cantidad de patrones existentes y la heterogeneidad en su descripción producen una "sobrecarga de información" en este contexto que dificulta la transmisión y gestión del conocimiento expresado por los patrones.

Esto hace que la mayoría de los patrones no sean conocidos por la comunidad de desarrollo (lo cual provoca la incursión en el antipatrón "*Reinventar la Rueda*"[5] [BMMM98]), que la gente no pueda entender realmente a los patrones cuando los utiliza, que no pueda encontrarlos, que al encontrarlos no sepa cómo utilizarlos o que no pueda compartir su experiencia en forma adecuada.

Compartir patrones es compartir conocimiento [Welicki06]. La falta de un modelo para describir, clasificar y compartir patrones dificulta la transmisión del conocimiento que el patrón intenta expresar. No existe un único modelo de referencia que sirva como guía para la descripción y definición de los patrones. Un modelo de este tipo no tiene por qué ser restrictivo: la existencia de un conjunto de reglas no implica menores capacidades de representación. De hecho, podemos tomar como ejemplo el lenguaje de programación C [KR88]: un mismo programa puede ser escrito en infinidad de formas y en todos los casos será un conjunto de instrucciones que pueden ser entendidas por alguien o algo (en este caso, un ordenador) que resuelven un problema concreto a través de la implementación de un algoritmo [Welicki05] [Welicki06].

Un problema común en todos los catálogos públicos de patrones es que la interfaz de usuario fuerza al usuario a concentrarse en una única vista del patrón, que ha sido definida en función de los intereses u objetivos del publicador de la información. Hemos encontrado un caso concreto que sirve como ejemplo: en el mundo de desarrollo en entornos .NET [MSNET] existe un sitio Web muy popular[6] [DoFactory] que contiene la intención, estructura e implementación de los 23 patrones del GoF [GoF95]. En muchos foros formales e informales de desarrollo cuando se hace referencia a los patrones se está refiriendo a la implementación de éstos, pero no al patrón en sí (citando como ejemplo al sitio Web mencionado anteriormente en este párrafo). En este caso, algunas personas confunden una implementación concreta del patrón con el concepto real que el patrón intenta comunicar, minimizando así la utilidad y posibilidades de los patrones como herramienta de transferencia y comunicación de conocimiento (entendiendo correctamente al patrón podemos construir un millón de soluciones distintas basándonos en él [Alexander79], [GoF95]). En el caso que hemos puesto como ejemplo, el foco es la implementación. En otros casos el foco puede ser la descripción ([PatternShare], [Booch]) o asistir al modelado ([Together], [Rational]). Por tanto, cliente de visualización impone al usuario la forma en que visualizará, utilizará y por consiguiente comprenderá a los patrones que contiene el catálogo. Consideramos que esto no es correcto y que el usuario debería poder decidir en qué aspecto de los patrones quiere enfocarse según su necesidad (implementación, descripción, estructura, diagramas, etc.)

La utilidad y capacidad de transmisión de conocimiento de los catálogos existentes se ve disminuida por su falta de expresividad: en la mayoría de los casos sólo se centran fuertemente en el concepto de patrón, dejando de lado la posibilidad de

[5] "Reinvent the Wheel" en el texto original [BMMM98]

[6] Este sitio pertenece a una empresa que ofrece formación sobre desarrollo utilizando patrones, con lo cual la información que expone es un complemento a sus cursos y tiene fines promocionales.

describir otros conceptos que puedan permitir al usuario ***realmente*** comprender el patrón (como conceptos de orientación a objetos en que se funda el patrón, lenguaje o sistema de patrones donde el patrón está incluido, fuente de información de donde ha sido extraído el patrón, etc.). Como ejemplo práctico podemos mencionar el siguiente caso: para comprender mejor el patrón ABSTRACT FACTORY [GoF95] es deseable conocer y entender el "*Dependency Inversion Principle*" [Martin02], [Welicki05] que es el que nos permite comprender realmente y poder derivar[7] [ST01] este patrón.

Actualmente existen varios enfoques para descripción, catalogación y compartición de patrones ([PLML], [Booch], [PatternShare], [Together], [Rational], [WOP], [WS06]), pero todos ellos tienen carencias en algún aspecto. Estas carencias se analizan posteriormente en el **capítulo 10** de esta tesis.

Dichas carencias pueden centrarse en diversos ámbitos. Respecto a la descripción, podemos mencionar los siguientes problemas generales:

- En casi todos los casos, los patrones no son descriptos en su totalidad y no incluyen información comprehensiva del "nivel de conocimiento".

- En los casos centrados en la implementación, esta está fuertemente orientada a una plataforma de ejecución o lenguaje concreto.

- En general no es posible describir conceptos de soporte como lenguajes de patrones, categorías, refactorizaciones, principios de orientación a objetos, fuentes de información, etc., con las mismas construcciones sintácticas y semánticas utilizadas para describir a los patrones.

Respecto a la catalogación nos encontramos con problemas como reunir información de fuentes dispersas (por ejemplo, la información necesaria para comprender un patrón puede estar distribuida en varios libros, artículos y páginas Web). Esto se debe en gran medida a que el concepto central a describir es el patrón y a la falta del soporte adecuado para describir a estas entidades. Las capacidades relacionales de los catálogos también son un aspecto a tener en cuenta: debe ser posible crear relaciones entre todo tipo de elementos, aun cuando no sean de la misma clase o no estén en el mismo contexto. Otro problema es la necesidad de tener toda la información de una entidad para poder introducirla. Se debe facilitar la introducción de información incompleta y brindar la infraestructura de soporte necesaria para poder completarla a posteriori.

Finalmente, respecto a la visualización en la mayoría de los casos se produce el fenómeno negativo que hemos mencionado anteriormente: en todos los casos se impone al usuario una vista concreta sobre el catálogo.

[7] Derivar el patrón se refiere a poder entender como se llega a la solución que propone el patrón. En "*Design Patterns Explained*" [ST01] se derivan varios de los patrones del GoF.

1.3 Hipótesis

La construcción de software es una disciplina compleja [Booch], con un extraño balance entre arte y ciencia [HT00] donde la experiencia tiene un rol determinante. Por lo tanto, es necesario un medio para documentar y compartir esa experiencia a efectos de facilitar el trabajo de construcción.

Los patrones son el medio idóneo para compartir experiencia [Welicki06]. Cada patrón es una relación entre un contexto, un sistema de fuerzas que ocurren repetidamente en ese contexto y una configuración espacial que permite que esas fuerzas se resuelvan entre sí [Alexander79]. Los patrones permiten codificar el conocimiento en forma uniforme.

Compartir patrones es compartir conocimiento [Welicki06b]. Las tecnologías y especificaciones existentes para describir y compartir patrones se centran sólo en uno o un conjunto discreto de aspectos, dejando de lado otros aspectos relevantes. Actualmente no existe un modelo que permita compartir a los patrones en forma efectiva, dificultando la correcta transmisión de conocimiento en la ingeniería del software [Welicki06]

A partir de lo expuesto en los párrafos y secciones anteriores, se plantea la siguiente hipótesis de partida:

> Es posible codificar en forma abstracta a los patrones y a sus conceptos de soporte a un alto nivel de abstracción en forma flexible y extensible abarcando todos los modelos de descripción posibles, catalogarlos y compartirlos con la comunidad para gestionar y transmitir adecuadamente el conocimiento que intentan expresar.

1.4 Objetivos

De esta hipótesis se derivan los ***objetivos generales*** de esta tesis, que se resumen en el siguiente apartado:

Crear un lenguaje de meta-especificación de patrones que permita describir a los patrones en forma independiente de la plantilla con la que fueron creados y el lenguaje y plataforma de ejecución donde se implementarán.

Utilizar este lenguaje para construir un catálogo de patrones que incluya también todos los conceptos necesarios para poder realmente entender al patrón (lenguaje de patrones, categoría, conceptos en que se funda, etc.). El catálogo debe permitir a los usuarios encontrar y comprender fácilmente a los patrones y exponer su información mediante interfaces débilmente acopladas.

Exponer toda esta información mediante un visualizador, que haga que este conocimiento sea accesible para el público general, independientemente del fin para el que quiera utilizarlo. El catálogo debe ser un elemento vivo, lo cual le permitirá evolucionar a partir del conocimiento de la comunidad.

Para alcanzar los ***objetivos generales*** establecidos arriba se plantean los siguientes ***objetivos parciales***:

- ***Crear un lenguaje de representación de patrones.*** Representar a los patrones en un formato abstracto y extensible independiente de las plantillas, los lenguajes de programación y plataformas de ejecución. Esta representación debe ser auto-contenida, englobando los niveles de conocimiento e implementación.

- ***Dotar al lenguaje de expresividad para describir conceptos.*** Dotar al lenguaje de construcciones que permitan describir los conceptos adicionales que dan soporte a la clasificación y comprensión del patrón. Estos conceptos incluyen lenguajes de patrones, categorías, tipos de patrones, autores, conceptos de orientación a objetos, etc. Se deben poder crear nuevos tipos de conceptos en función de las necesidades de descripción de un patrón o de otro concepto.

- ***Dotar al lenguaje de capacidades relacionales.*** Poder enlazar a los patrones con otros patrones y con conceptos (principios de diseño, temas de ingeniería, autores, obra en que se incluye, etc.), dado que los patrones no son entidades que existen en forma "aislada".

- ***Dotar al lenguaje de capacidades de anotación.*** Permitir anotar a los patrones con etiquetas para su posterior búsqueda, clasificación y utilización. Estas etiquetas pueden ser establecidas en forma colaborativa.

- ***Construir un catálogo de patrones.*** Construir un catálogo de patrones existentes representados con el lenguaje desarrollado en esta tesis.

- ***Crear la infraestructura de catalogación***. Establecer una forma para poder almacenar los patrones en un catálogo y una forma eficiente de poder registrarlos, anotarlos, buscarlos, referirlos, enlazarlos y utilizarlos.

- ***Crear una herramienta de visualización del catálogo y de los patrones y conceptos.*** Crear una herramienta para navegar y utilizar el catálogo que sea fácilmente accesible para el público masivo.

- ***Establecer un mecanismo de visualización de los patrones y conceptos***. El usuario debe visualizar los aspectos que considere relevantes de los patrones.

- ***Habilitar el trabajo colaborativo para evolucionar a los patrones***. Potenciar la discusión y participación de la comunidad a través de foros, discusiones, edición colaborativa, etc., a efectos de establecer el marco propicio para la evolución del conocimiento del catálogo y la creación de nuevo conocimiento a partir del existente.

- ***Convertir al catálogo en un proveedor de servicios de información.*** Exponer la información alojada en el catálogo mediante una interfaz de servicios, para que pueda ser utilizado por clientes heterogéneos débilmente acoplados sin dependencia alguna sobre la arquitectura propuesta.

1.5 Aportaciones y Beneficios

La implementación del esquema de meta-especificación da como resultado un repositorio contextualizado, que contiene a los patrones de software conocidos y documentados al momento de su creación y la capacidad de introducir nuevos, permitiendo generar diversos artefactos de software cuando sea apropiado (como por ejemplo, código fuente o diagramas UML [UML], entre otros). De esta manera, los patrones actúan en forma generativa [AIS77] [Alexander79], siendo la materia prima de entrada para crear software (y para generar más patrones).

Estas técnicas y tecnologías brindan soporte a los equipos[8] de desarrollo e investigación (tanto en contextos empresariales como investigadores), ofreciendo una plataforma para la gestión de las experiencias de diseño, arquitectura e implementación exitosas (patrones e idiomas) y generación automática de artefactos, aplicables tanto al mundo académico como empresarial.

La creación de una herramienta de explotación del catálogo basada en tecnologías Web permitirá que esta base de conocimiento sea accesible a un gran público. Mediante su utilización se podrán buscar, utilizar y relacionar patrones, siguiendo con la filosofía práctica y pragmática de la comunidad de patrones.

[8] El conjunto de tecnologías propuestas en esta tesis brinda soporte a equipos que pueden estar distribuidos geográficamente.

La combinación de lenguajes de dominio específico (DSL) y modelos de objetos adaptativos (AOM) permiten la creación de un lenguaje de gran expresividad soportado sobre una plataforma de ejecución flexible [Welicki06].

A continuación, se enumeran los principales beneficios y aportaciones originales que surgen de esta tesis:

- ✓ Un lenguaje de meta-especificación para describir patrones y conceptos asociados a un alto nivel de abstracción comprendiendo los niveles de conocimiento e implementación, abarcando todas las plantillas de patrones existentes. Permite describir también relaciones y metadatos para búsquedas y clasificación. (**capítulo 12**).

- ✓ Un catálogo de patrones y conceptos, para establecer una base de conocimiento práctico. El catálogo no es un mero contenedor de pasivo de información: contiene componentes activos que tienen como responsabilidad la gestión y compartición de sus contenidos. Contiene también mecanismos de comunidad que permite la evolución y generación de conocimientos a partir de sus contenidos. (**capítulo 13**).

- ✓ Una herramienta Web de visualización que expone a los usuarios los contenidos del catálogo en forma usable, ofreciéndoles varios mecanismos de navegación y visualización de entidades y la posibilidad de discutir sobre los contenidos (**capítulo 14**).

- ✓ Combinación de lenguajes de dominio específico (DSL) y modelos de objetos adaptativos (AOM) para resolver el problema de la meta-especificación catalogación de patrones de software en forma ágil y flexible (§ 13.4.2).

- ✓ Ciclo de vida de los patrones más dinámico e interactivo, estableciendo las bases para la evolución continua y generación de conocimiento a partir de interacciones entre los miembros de la comunidad (§ 11.6).

Capítulo 2
Metodología y Desarrollo de la Investigación

"Si buscas resultados distintos,
no hagas siempre lo mismo"
Albert Einstein

En este capítulo presentaremos la metodología utilizada para el desarrollo de esta tesis doctoral. A continuación se expone el desarrollo temporal de la investigación, exponiendo los principales hitos en orden cronológico. Finalmente se presenta la organización de la memoria, estructurada tanto en secciones como en capítulos para ofrecer al lector una visión general y de alto nivel del contenido de esta tesis.

❖ ❖ ❖

2.1 Metodología

Este trabajo de investigación ha sido desarrollado en varias etapas utilizando un enfoque iterativo incremental. En la figura 2.1 a continuación se muestra en forma gráfica el marco metodológico de la investigación con referencias temporales.

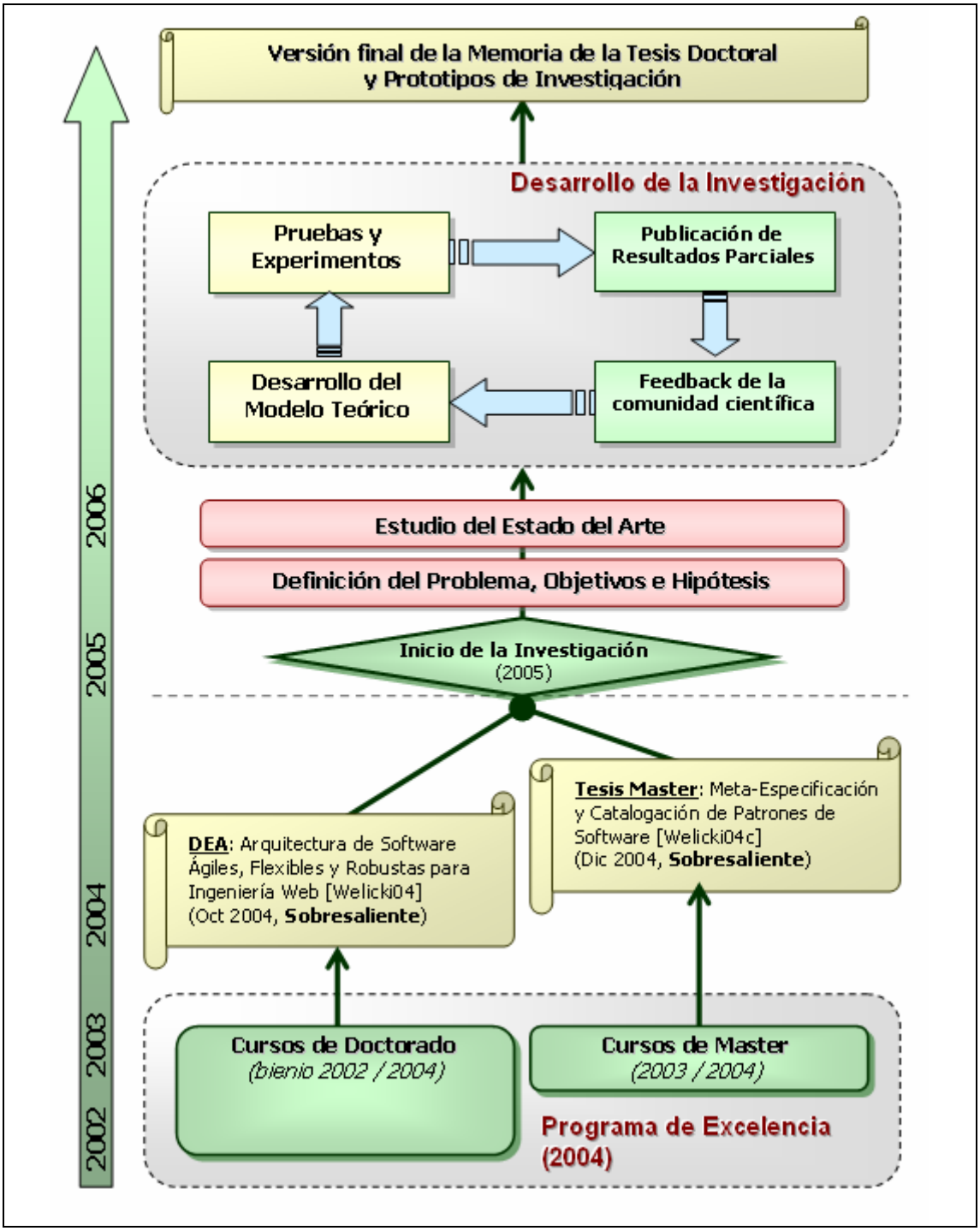


Figura 2.1 – Metodología iterativa-incremental (con referencias temporales) utilizada durante el desarrollo de la investigación. En esta figura puede verse a gran escala el desarrollo temporal así como los antecedentes y la filosofía de constante experimentación y contraste con la comunidad científica.

El punto de partida de nuestra investigación es el trabajo realizado en el marco del "*Programa de Excelencia*"[9] [UPEX] de la Universidad Pontificia de Salamanca, campus Madrid. A partir de los conocimientos y experiencias obtenidas en los cursos de los programas de Master en Ingeniería del Software y Doctorado en Ingeniería Informática (programa Ingeniería del Software) se realizan dos trabajos de investigación que son cruciales para el desarrollo de esta investigación: la memoria de investigación desarrollada en el marco de la investigación tutelada del programa doctoral para obtener la suficiencia investigadora ("*Arquitecturas de Software Ágiles, Flexibles y Robustas para Ingeniería Web*" [Welicki04]) y la tesis de maestría ("*Meta-Especificación y Catalogación de Patrones de Software*" [Welicki04c]). Estos trabajos sientan bases claras y sólidas para la investigación realizada en el desarrollo de esta tesis doctoral.

A partir del estudio del estado del arte, prototipos construidos y conclusiones obtenidas en esos dos trabajos comenzamos con el desarrollo de esta tesis. El paso siguiente fue establecer claramente el problema a resolver y partir de éste los objetivos e hipótesis de partida. A partir de éstos ampliamos el estudio del estado del arte, que había sido cubierto en gran medida en los trabajos anteriores. Este nuevo estado del arte "ampliado" mejoraba el existente en los trabajos previos y se ampliaba con la inclusión de los lenguajes de dominio específico, modelos de objetos adaptativos y un estudio más exhaustivo de las tecnologías y enfoques existentes. Es importante destacar que el estado del arte no ha sido una sección estanca y se ha visto actualizada y mejorada en reiteradas ocasiones conforme avanzábamos con nuestra investigación y surgían nuevos retos e inquietudes.

Para el desarrollo de la investigación hemos optado por la aplicación de un enfoque ***iterativo e incremental***. Cada iteración comenzaba estableciendo un modelo teórico de un conjunto de aspectos concretos y continuaba con la creación de una serie de prototipos para realizar experimentos sobre el modelo planteado. Los resultados parciales de estos experimentos e ideas se publicaban en foros nacionales e internacionales de reconocido prestigio (como por ejemplo OOPSLA 2006, PLoP 2005, EuroPLoP 2006, JSWEB 2005 y SISOFTW 2006, entre otros) donde eran contrastados y discutidos con la comunidad científica e investigadora internacional[10]. El feedback obtenido servía como realimentación del proceso, convirtiéndose en la entrada para extender o modificar el modelo teórico, modificar o crear nuevas funcionalidades en el prototipo, realizar nuevos experimentos y volver a publicar para discutir estos nuevos resultados con la comunidad. Es importante destacar la importancia del análisis retrospectivo de la información obtenida. A modo de ejemplo citaremos un caso concreto: a partir de las opiniones recabadas en el PLoP 2005 (donde se realizó una presentación de 45 minutos sobre una versión preliminar del modelo propuesto en esta tesis)

[9] Programa de la escuela de postgrado que permite simultanear estudios de Master y Doctorado

[10] A lo largo del desarrollo de esta tesis se han realizado 21 publicaciones en congresos en Estados Unidos, Europa, España e Iberoamérica, las cuales se enumeran en el capítulo 18

pudimos entender mejor las necesidades de la comunidad de patrones y establecer mejor el foco de nuestra investigación[11].

2.2 Desarrollo Temporal de la Investigación

A continuación, presentaremos algunos de los hitos más significativos de la investigación:

- **Octubre de 2002**: Inicio de los cursos de doctorado en el campus de Madrid de la Universidad Pontificia de Salamanca.

- **Mayo de 2003**: publicación del trabajo "*La Gestión del Conocimiento en las Empresas de Consultoría Informática*" [Welicki03] en el CISIC 2003, donde se establece la necesidad de una forma de gestión de conocimiento de ingeniería en las empresas de software. En ese mismo mes se publica también el trabajo "*Un Modelo de Compartición de Conocimiento en Red para Integrar Comunidades del Tercer Mundo*" [Welicki03b] donde se plantea un modelo de red basado en las reglas de la emergencia para compartir y gestionar conocimientos heterogéneos distribuidos cultural y geográficamente.

- **Agosto 2003**: publicación del trabajo "*Desarrollo Multilenguaje con Patrones de Diseño en .NET*" [Welicki03c] en el SISOFT 2003 (junto con Fernando Cano), donde se propone un esquema de utilización de patrones para implementar en forma eficiente la programación políglota [Meyer02b] (o multilenguaje) usando las capacidades del CLR de .NET [Box02].

- **Marzo / Mayo 2004**: publicación de los trabajos "*El precio de las palabras en la sociedad de la información*" en JIS 2004 [Welicki04b] y "*XText: un modelo para publicación de textos en múltiples dispositivos*" en JPIII 2005 [Welicki04b] que sientan las bases del modelo de interacción que se utilizará posteriormente para crear el visor del catálogo de meta-especificaciones.

- **Enero / Octubre 2004**: realización del trabajo de investigación para el DEA, titulado "*Arquitecturas de Software Ágiles Flexibles y Robustas para Ingeniería Web*" [Welicki04]. En este trabajo se estudia en forma exhaustiva el estado del arte que establece las bases fundamentales de esta investigación: una de las líneas de investigación propuestas era "*meta-modelado de patrones*", la cual ha evolucionado posteriormente para convertirse en esta tesis.

- **Octubre de 2004**: Presentación del trabajo de investigación "*Arquitectura de Software Ágiles Flexibles y Robustas para Ingeniería Web*" [Welicki04] donde se

[11] Esto podemos afirmarlo en forma rotunda, dado que varios meses después publicamos nuevos resultados de nuestro trabajo en el EuroPLoP 2006 y recibimos muy buena acogida por parte de la comunidad

propone públicamente ante el tribunal el "meta-modelado de patrones" como línea de investigación. La calificación obtenida en la defensa es 10 (sobresaliente).

- **Diciembre 2004**: Presentación de la tesis de Master en Ingeniería del Software titulada "*Meta-especificación y Catalogación de Patrones de Software*" [Welicki04c]. En este trabajo se profundiza el estado del arte relevado en el trabajo anterior, se estudian herramientas existentes y se buscan sus carencias y se establece un modelo enfocado en la meta-especificación de patrones y generación de código. Se crea la primera versión de EML (llamada entonces M4PS) y se crea un prototipo de investigación para verificar que lo que se propone es factible.

- **Mayo 2005**: la tesis de master "*Meta-especificación y Catalogación de Patrones de Software*" [Welicki04c] es defendida ante el tribunal de la facultad de Informática de la Universidad Pontificia de Salamanca. La calificación obtenida en la defensa es "*sobresaliente*".

- **Enero 2005 - Junio 2005:** una vez verificada la factibilidad de la propuesta se mejora el prototipo y se comienza a trabajar en la catalogación y en la mejora del analizador EML (todavía llamado M4PS, versión 0.9). En Junio se envía el trabajo titulado "*A Model for Meta-Specification and Cataloging of Software Patterns*" [Welicki05] a la conferencia *12th Pattern Language of Programs* (PLoP 2005) [PLoP]. Este evento es el máximo referente sobre patrones a nivel mundial y a éste asisten los principales referentes de la comunidad de patrones. Previamente se realiza un trabajo de mejora del paper (conocido como *shepherding* [Shepherding]) durante un mes y medio con Sargon Hasso, del Illinois Institute of Technology (USA) [IIT].

- **Julio 2005 / Septiembre 2005:** Se publica en la revista MJT.NET de MSDN (Microsoft Developers Network) un artículo de dos partes titulado "*Patrones y Antipatrones: una introducción*" [Welicki05b] [Welicki05c], donde se presenta una visión general sobre patrones, que es la que utilizamos en esta tesis (**capítulo 3**).

- **Septiembre de 2005:** "*A Model for Meta-Specification and Cataloging of Software Patterns*" [Welicki05] es presentado en el PLoP 2005 en Illinois, USA, en un track especial de 45 minutos. Nuevas versiones del modelo teórico y del prototipo son presentadas ante la comunidad. Se obtiene un importante feedback de la comunidad internacional de patrones que es crucial para el desarrollo futuro de la investigación.

- **Diciembre 2005:** Se crea la versión 0.9.1 de EML (en esta versión se descarta el nombre MP4S), que incluye nuevas construcciones para catalogación. Se diseña e implementa un *Adaptive Object Model* [AOM] [YBJ01] [YJ02] que permite componer elementos atómicos de EML para

poder describir cualquier entidad o concepto (fuente de información, autor, refactorización, principio de diseño, etc.). Se mejora el analizador de EML para poder soportar estas nuevas funcionalidades.

- **Enero 2006:** Se implementa el "*gradiente semántico*". Se crean vistas genéricas con capacidad de layout basado en metadatos, separando aún más la visualización de la representación. El prototipo implementa una fuerte separación entre datos y presentación y sus funcionalidades principales son dirigidas por metadatos. Se potencia la flexibilidad a partir del acoplamiento abstracto y la inyección de dependencias.

- **Febrero 2006:** Se envía el trabajo "*Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models*" al congreso 11th European Pattern Lanaguages of Programs Conference (EuroPLoP 2006), máxima referencia en el ámbito de patrones a nivel Europeo. El trabajo presentado es un resumen de los contenidos y enfoques principales de esta tesis.

- **Febrero / Abril 2006:** Durante Febrero, Marzo y Abril, se realiza un trabajo de mejora del paper enviado a EuroPLoP (conocido como shepherding [Shepherding]) con Till Schuemmer, de la Universidad Fernuni en Haagen [FUH] (Alemania). Paralelamente se prosigue con el desarrollo del prototipo y con mejoras en el modelo general.

- **Mayo 2006:** "*Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models*" es aceptado para su presentación en EuroPlop 2006.

- **Julio 2006:** "*Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models*" es presentado en EuroPLoP 2006 en Kloster Irsee, Alemania. En esa misma conferencia el autor organiza un sesión BOF[12] (junto con Wolfram Schobert, autor de COPE [SS06]) para discutir con la comunidad sobre descripción y catalogación de patrones. Se obtiene importante feedback sobre el trabajo realizado.

- **Junio 2006:** "*Meta-Specification and Cataloging of Software Patterns: Towards Knowledge Management in Software Engineering*" es enviado para la sesión de posters de la 21st ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA 2006), junto con un extended abstract que expresa en forma clara y concisa la visión de gestión del conocimiento y patrones establecida en esta tesis.

- **Julio / Agosto 2006:** "*Meta-Specification and Cataloging of Software Patterns: Towards Knowledge Management in Software Engineering*" es aceptado para su presentación en la sesión de posters de OOPSLA 2006. En Agosto se

[12] Birf of a Feather (en ingles)

envía la versión definitiva del extended abstract para su publicación en las actas del congreso.

- **Octubre 2006:** "*Meta-Specification and Cataloging of Software Patterns: Towards Knowledge Management in Software Engineering*" es presentado en la sesión de posters de OOPSLA 2006. Se obtiene un feedback positivo sobre el trabajo realizado.

2.3 Organización de esta Tesis

La presente memoria está estructurada tanto en secciones como en capítulos. En este apartado presentaremos en forma resumida la organización de esta tesis para facilitar su lectura posterior.

La ***parte I*** titulada "***Planteamiento del Problema***" incluye los capítulos introductorios donde se presentan el problema, la hipótesis y su contexto. En el **capítulo 1**, "***Introducción***", se presentan la justificación, motivación y planteamiento del problema a resolver en esta tesis. Luego se presenta la hipótesis y se establecen los objetivos de la investigación. Finalmente se enumeran brevemente las principales aportaciones y beneficios. En el **capítulo 2** titulado "***Metodología y Desarrollo de la Investigación***" se presenta información metodológica sobre el desarrollo de esta tesis incluyendo la metodología, el desarrollo temporal y la estructura de esta memoria.

En la ***parte II*** titulada "***Conceptos, Soluciones y Tecnologías Existentes***" se presentan las tecnologías, conceptos sobre los que se funda la investigación realizada en esta tesis doctoral. Los primeros capítulos de esta sección se centran en el estudio de tecnologías y conceptos existentes incluyendo "***Patrones***" (**capítulo 3**), "***Plantillas de Patrones***" (**capítulo 4**), "***Gestión del Conocimiento***" (**capítulo 5**), el "***Meta-Niveles en la Ingeniería Informática***" (**capítulo 6**), "***Lenguajes de Dominio Específico***" (**capítulo 7**), "***Modelos de Objetos Adaptativos***" (**capítulo 8**) y "***Sistemas Emergentes***" (**capítulo 9**). Posteriormente en el **capítulo 10** titulado "***Enfoques Existentes***" se estudian en detalle tecnologías y soluciones existentes en la actualidad al problema planteado al inicio de esta memoria y se plantean una serie de problemas recurrentes encontrados en todas ellas.

En la ***parte III*** titulada "***Solución Propuesta***" se presenta la solución propuesta en esta tesis para el problema planteado al inicio en el capítulo 1. En el **capítulo 11** "***Arquitectura General de la Solución***" se presenta la solución a nivel conceptual y de arquitectura. Este capítulo ofrece una visión concreta de la solución completa desarrollada en esta tesis. En el **capítulo 12**, "***EML - El Lenguaje de Meta-Especificación de Patrones***", se presenta a EML (*Entity Meta-specification Language*) el lenguaje creado para describir a los patrones y las entidades a un elevado nivel de abstracción. En el **capítulo 13**, "***El Catálogo***", se

presenta el catálogo de entidades que es el contenedor de las meta-especificaciones de las entidades junto con la infraestructura para su gestión, almacenamiento y compartición. Finalmente, en el **capítulo 14**, "***El Visor del Catálogo***", se exponen los principios y diseño del catálogo de patrones, que es la herramienta que permite visualizar el contenido del catálogo a través de la Web.

En la ***parte IV*** titulada "***Desarrollo de un Prototipo***" se presenta el prototipo construido durante el desarrollo de esta tesis doctoral. En el **capítulo 15**, "***Descripción del Prototipo***", se explica el prototipo desarrollado a un nivel general, incluyendo sus funcionalidades, arquitectura general y tecnologías utilizadas para su desarrollo. En el **capítulo 16**, "***Desarrollo del Prototipo***", se explican en forma detallada algunos módulos específicos del prototipo desarrollado. En el **capítulo 17**, "***Evaluación***" se evalúa el prototipo, contrastándolo con otros enfoques existentes (estudiados previamente en la ***parte II*** de esta tesis).

En la ***parte V*** titulada "***Conclusiones***" se presentan las conclusiones generales que pueden extraerse de esta tesis doctoral (**capítulo 18**). Estas conclusiones incluyen un análisis general de los resultados obtenidos, una síntesis del trabajo propuesto, la verificación del cumplimiento de objetivos, el escrutinio público del trabajo realizado y las aportaciones realizadas. Posteriormente se presentan las futuras líneas de investigación derivadas del trabajo realizado.

En la ***parte VI*** titulada "***Apéndices***" se presentan apéndices varios que enriquecen los contenidos presentados a lo largo de la disertación. Se incluye un breve ensayo sobre el mal uso de los patrones (**apéndice A**), la especificación de la versión 1.0 de EML (**apéndice B**) y la visión de gestión de conocimiento que se presenta en esta tesis (**apéndice C**).

Parte II
Conceptos, Soluciones y Tecnologías Existentes

"No basta con adquirir sabiduría,
es preciso además saber usarla"
Cicerón

Capítulo 3
Patrones

"El aprendizaje más importante proviene
de la experiencia directa"
Nonaka & Takehuchi

Los patrones son una disciplina de resolución de problemas reciente en la ingeniería del software que ha emergido de la comunidad de orientación a objetos. Los patrones tienen raíces en muchas áreas, incluyendo "*literate programming*" [Knuth92] y más notablemente en el trabajo de Christopher Alexander [AIS77] [Alexander79] en planeamiento urbanístico y arquitectura civil.

En este capítulo estudiaremos el concepto de patrón, para luego analizar su aplicación a diversos ámbitos de la ingeniería del software (especialmente al diseño y a la arquitectura). Proseguiremos nuestro estudio analizando a los antipatrones y enumerando distintos tipos de patrones. Finalmente haremos una breve revisión de las iniciativas en este campo de tres de las principales compañías de software (Microsoft, IBM y Sun).

❖ ❖ ❖

3.1 Patrones

Los patrones son una disciplina de resolución de problemas reciente en la ingeniería del software que ha emergido de la comunidad de orientación a objetos. Los patrones tienen raíces en muchas áreas, incluyendo "*literate programming*" [Knuth92] y más notablemente en el trabajo de Christopher Alexander [AIS77] [Alexander79] en planeamiento urbanístico y arquitectura civil.

3.1.1 Definición

En el libro "*The Timeless Way of Building*" [Alexander79], el arquitecto Christopher Alexander define al término patrón en la siguiente forma:

> *"Cada patrón es una regla de 3 partes, que expresa una relación entre un contexto, un problema y una solución. Como un elemento en el mundo, cada patrón es una relación entre un contexto, un sistema de fuerzas que ocurren repetidamente en ese contexto y una configuración espacial que permite que esas fuerzas se resuelvan entre sí."*

Como elemento de un lenguaje, un patrón es una instrucción que muestra como puede ser usada esta configuración espacial una y otra vez para resolver el sistema de fuerzas, siempre que el contexto lo haga relevante.

Continuando con la definición anterior podemos añadir otro párrafo de Alexander en [Alexander79]:

> *"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma."*

Si nos fijamos en las construcciones de una determinada zona rural observaremos que todas ellas poseen apariencias parejas (tejados de pizarra con gran pendiente, etc.), pese a que los requisitos personales por fuerza han debido ser distintos. De alguna manera la esencia del diseño se ha copiado de una construcción a otra, y a esta esencia se pliegan de forma natural los diversos requisitos. Diríase aquí que existe un patrón que soluciona de forma simple y efectiva los problemas de construcción en tal zona.

A modo de complemento, citaré dos principios postulados por Martin Fowler en "*Analysis Patterns*" [Fowler96]:

- *"Los patrones son un punto de partida, no un destino."*
- *"Los modelos no están bien o mal, sino que son más o menos útiles."*

Como reflexión final, un patrón no es una solución en si mismo, sino la documentación de la forma en que construyeron soluciones a problemas similares en el pasado, lo cual permite una mejor gestión de la experiencia y transferencia de conocimientos.

3.1.2 Patrones de Software

Una definición con amplia aceptación en el mundo del software es la dada por Richard Gabriel [Dreamsongs] en "*Timeles Way of Hacking*" [Hillside]

> *"Cada patrón es una regla de tres partes que expresa una relación entre un contexto, un sistema de fuerzas que ocurren repetidamente en ese contexto y una configuración de software que permite que esas fuerzas se resuelvan."*

Otra definición complementaria, más enfocada en el ámbito técnico, es la que se da en [Microsoft04]:

> *"Una descripción de un problema recurrente que ocurre en un contexto y basado en un conjunto de fuerzas recomienda una solución. La solución es usualmente un mecanismo simple: una colaboración entre dos o más clases, objetos, servicios, procesos, hilos, componentes o nodos que trabajan juntos para resolver el problema identificado por el patrón."*

Finalmente, citaremos la definición que se presenta en [Welicki05], que es la que utilizamos como punto de partida y base para nuestra investigación:

> *"Una patrón es una pieza de conocimiento que incluye información sobre un problema y su solución en un contexto, con sus ventajas e inconvenientes y toda la información necesaria para poder tener un buen entendimiento de los aspectos relacionados con él. Eventualmente puede contener información específica sobre su comportamiento, la cual permite generar artefactos de software"*

Los patrones de software facilitan el reutilización del diseño y de la arquitectura, capturando las estructuras estáticas y dinámicas de colaboración de soluciones exitosas a problemas que surgen al construir aplicaciones.

3.1.3 Características de un Buen Patrón

Documentar buenos patrones es una tarea muy difícil [GoF95]. Citando a James Coplien en [Hillside], un buen patrón:

- **Resuelve un problema**: los patrones capturan soluciones, no principios o estrategias abstractas.

- **Es un concepto probado**: capturan soluciones, no teorías o especulaciones. En el caso de [GoF95], un criterio utilizado para decidir si

algo era un patrón o no es que debía tener al menos tres implementaciones reales.

- **La solución no es obvia:** muchas técnicas de resolución de problemas (como los paradigmas o métodos de diseño de software) intentan derivar soluciones desde principios básicos. Los mejores patrones generan una solución a un problema indirectamente (un enfoque necesario para los problemas de diseño más difíciles).

- **Describe una relación**: los patrones no describen módulos sino estructuras y mecanismos.

- **Tiene un componente humano significante**: el software sirve a las personas. Los mejores patrones aplican a la estética y a las utilidades (de hecho, no es casual que varios de los primeros lenguajes de patrones tengan que ver con temas estéticos y utilidades como HotDraw [Hotdraw] o ET++ [ET]).

3.1.4 Lenguajes de Patrones

Un lenguaje de patrones define una colección de patrones y las reglas para combinarlos en un estilo arquitectural. Los lenguajes de patrones describen marcos de trabajo[13] de software o familias de sistemas relacionados [Hillside].

3.1.5 Taxonomía de Patrones

Los patrones existen en distintos niveles de abstracción. En [POSA96] se define una taxonomía que se muestra en la figura a continuación.

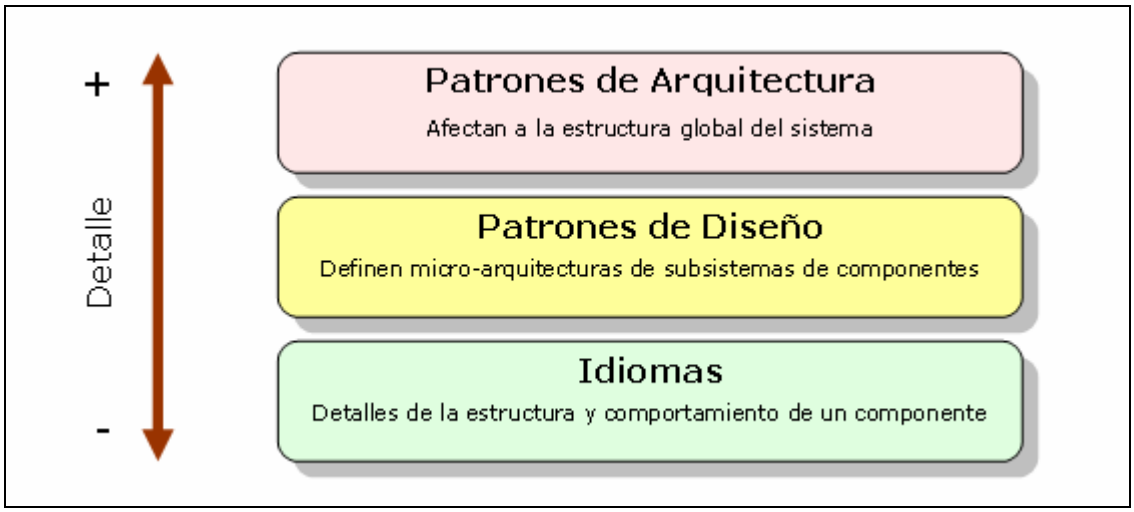


Figura 3.1 – Patrones según el nivel de detalle, adaptado de [POSA96].

[13] Frameworks (en inglés)

Es importante destacar que esta división no es absoluta ni totalmente comprehensiva, dado que no incluye a otros patrones de amplia aceptación y utilización como los de análisis, integración de aplicaciones, pruebas, etc.

3.1.6 Pattern Happy

Los patrones no son siempre la solución adecuada o mejor para un problema. Si pueden añadir flexibilidad, también añaden complejidad. Es por esto que se debe ser cuidadoso al momento de seleccionar patrones. Siempre hay que recordar que los patrones son un punto de partida y que no son dogmas incuestionables.

El uso indiscriminado de patrones, aun en situaciones donde no aportan ningún beneficio, puede ser un antipatrón o un claro indicador de ser un *Pattern Happy* [Kerievsky04] (esto es "alguien que abusa de los patrones sin razón", concepto que se explica en detalle en el **apéndice A**).

3.1.7 Los Patrones y la Gestión del Conocimiento

Los patrones permiten establecer un vocabulario común de diseño [GoF95], cambiando el nivel de abstracción a colaboraciones entre entidades y permitiendo comunicar conocimiento sobre problemas y soluciones en un contexto.

Son también un gran mecanismo de comunicación para transmitir la experiencia de los ingenieros y diseñadores experimentados a los más noveles, convirtiéndose en unas de las vías para la gestión del conocimiento.

Lo que se pretende con un catálogo de patrones no es favorecer al diseñador experto (que quizás no necesite en absoluto los patrones), sino mas bien ayudar al diseñador inexperto a adquirir con cierta rapidez las habilidades de aquél [GoF95], como también comunicar al posible cliente, si es el caso, las decisiones de diseño en forma clara y autosuficiente [Cueva04].

3.2 Patrones de Diseño

Un patrón de diseño describe una estructura recurrente de componentes que se comunican para resolver un problema general de diseño en un contexto particular [GoF95].

Nomina, abstrae e identifica los aspectos claves de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable. Identifica las clases e instancias participantes, sus roles y colaboraciones y la distribución de responsabilidades [GoF95].

Tiene, en general, cuatro elementos esenciales[14]: [GoF95]

[14] Estos elementos se explican con mayor detalle en el libro *Design Patterns* [GoF95]

- **Nombre**: permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Al dar nombres a los patrones estamos incrementando nuestro vocabulario de diseño, lo cual nos permite diseñar y comunicarnos con un mayor nivel de abstracción (en lugar de hablar de clases individuales nos referimos a colaboraciones entre clases).

- **Problema**: describe cuándo aplicar el patrón. Explica el problema y su contexto. A veces incluye condiciones que deben darse para que tenga sentido la aplicación del patrón.

- **Solución**: describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o implementación en concreto, sino que es más bien una plantilla que puede aplicarse en muchas situaciones diferentes.

- **Consecuencias**: son los resultados, así como ventajas e inconvenientes de aplicar el patrón.

Los patrones de diseño no son dogmas que deben ser aceptados. Qué es y qué no es un patrón de diseño es una cuestión que depende del punto de vista de cada uno[15] [GoF95] y del nivel de abstracción en que se trabaje.

3.2.1 Principios de Diseño

En [GoF95], se proponen unos principios fundamentales de diseño subyacentes a los patrones de diseño que permiten crear aplicaciones más flexibles y robustas:

- Programar para interfaces y no para una implementación.
- Favorecer la composición de objetos frente a la herencia de clases.
- Determinar qué es común y qué es variable (commonality analisys).
    - Común = Estable
- Permitir el reemplazo de lo variable mediante una interfaz común.

En este aspecto podemos trazar un paralelismo con los sistemas emergentes (que se analizan posteriormente en el **capítulo 9** de esta tesis), dado que estos principios individuales muy sencillos a nivel local producen sistemas robustos y flexibles en el nivel global.

[15] Esta afirmación concuerda con el comentario de Ralph Johnson, miembro del GoF, al respecto de qué es una arquitectura en el artículo de Martin Fowler "Who needs an architect" [Fowler03b]. Allí, Johnson destaca que la definición de arquitectura de IEEE no es totalmente correcta dado que las partes más importantes de un sistema no son las mismas desde el punto de vista de un ingeniero que desde el negocio.

3.2.2 Clasificación de Patrones de Diseño

En [GoF95] se definen 23 patrones de diseño y se dividen en tres categorías:

- **De Creación**: abstraen el proceso de creación de instancias de objetos. [Cueva04]

- **Estructurales**: tratan con la composición de clases u objetos [GoF95]. Se ocupan de cómo clases y objetos son utilizados para componer estructuras de mayor tamaño [Cueva04].

- **De Comportamiento**: caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad [GoF95]. Atañen a los algoritmos y a la asignación de responsabilidades entre objetos [Cueva04].

En la tabla a continuación se muestra la distribución de los 23 patrones definidos en [GoF95], libro de referencia de patrones de diseño:

		De Creación	**Estructurales**	**De Comportamiento**
Ambito	Clase	Factory Method	Adapter (de clase)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (de objetos) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Tabla 3.1 – Clasificación de patrones de diseño del GoF [GoF95]

3.2.3 Relajación de Patrones

Muchas veces los patrones son seguidos estrictamente, olvidando que son más bien un punto de partida en lugar de una meta. Es interesante observar como inclusive en el libro del GoF [GoF95] se relajan ciertas condiciones al momento de implementar un patrón. Para ver un ejemplo explicito de esto se recomienda revisar la implementación del patrón ABSTRACT FACTORY en C++ (hay otra en Smalltalk) en dicho libro [GoF95].

3.2.4 Sistemas Flexibles y Robustos con Patrones de Diseño

En el libro del GoF [GoF95], se presenta como caso de estudio la construcción de un editor de documentos, el *Flexi*. Luego de una primera lectura, puede observarse que sus requisitos no funcionales incluyen la flexibilidad y la robustez[16].

Mediante el uso de patrones de diseño (en este caso concreto se utilizan 8 de los 23 patrones presentados en el libro) se logra una solución que cumple con los objetivos propuestos. Para una descripción completa de Flexi, ver el capítulo 2 de [GoF95].

3.2.5 Los Patrones de Diseño no Son Perfectos

Muchas veces se trata a los patrones de diseño como verdades universales. Esto no puede ser más incorrecto. De hecho, los miembros del mentado GoF destacan que los patrones no son un destino sino un punto de partida.

En *Patterns Hatching* [Vlissides98], John Vlissides, uno de los miembros del GoF, analiza los patrones de diseño tradicionales desde un punto de vista diferente, intentando quitar el aura de misticismo que los rodea, detallando inclusive el proceso de descubrimiento de patrones utilizado por el GoF[17].

En esa misma obra trata temas críticos y muchas veces ignorados tales como el ciclo de vida de un SINGLETON (¿cuándo y cómo muere?), los problemas del OBSERVER y como reemplazarlo en algunos casos por un VISITOR y pérdidas de memoria en el VISITOR entre otros.

3.3 Patrones de Arquitectura

Los patrones de arquitectura expresan el esquema fundamental de organización para sistemas de software. Proveen un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y guías para organizar las relaciones entre ellos. [POSA96]

Los patrones de arquitectura representan el nivel más alto en el sistema patrones propuesto en [POSA96], reflejado en la figura 3.1. Ayudan a especificar la estructura fundamental de una aplicación .Cada actividad de desarrollo es gobernada por esta estructura, por ejemplo, el diseño detallado de los subsistemas, la comunicación y colaboración entre diferentes partes del sistemas, etc. [POSA96]

Cada patrón de arquitectura ayuda a conseguir una propiedad específica en el sistema global, como por ejemplo la adaptabilidad de la interfaz de usuario [POSA96]. Los patrones que dan soporta a características similares se agrupan en una misma categoría

[16] Para más detalle sobre los objetivos de diseño de Lexi, ver el capítulo 2 de [GoF95]
[17] En esa misma obra vuelca en contenido textual de los correos, donde se discute si Multicast debía ser un patrón individual o un caso particular del Observer. Finalmente, no se publicó como patrón en el libro del GoF.

3.3.1 Categorías de Patrones de Arquitectura

En esta sección enumeraremos los patrones de dos de las obras más relevantes en el ámbito de los patrones de arquitectura: "*Pattern Oriented Software Architecture, Volume 1*" [POSA96] (en adelante POSA) y "*Patterns of Enterprise Application Architecture*" [Fowler02] (en adelante PEAA).

3.3.1.1 Categorías de POSA

En [POSA96], libro de referencia de patrones de arquitectura, se divide a los patrones en las siguientes categorías:

- **From Mud to Structure**: los patrones en esta categoría ayudan a evitar un "mar" de componentes u objetos. En particular, soportan una descomposición controlada de una tarea del sistema en subtareas que cooperan.
- **Distributed Systems**
- **Interactive Systems**
- **Adaptable Systems**

En la tabla a continuación se muestra la distribución de los patrones definidos en *Patterns of Software Architecture* [POSA96]:

From Mud to Structure	Distributed Systems	Interactive Systems	Adaptable Systems
Layers Pipes and Filtres Blackboard	Broker	Model-View-Controller Presentation-Abstraction-Control	Microkernel Reflection

Tabla 3.2 – Clasificación de patrones de arquitectura de POSA [POSA96]

3.3.1.2 Categorías de PEAA

En [Fowler02][18] Martin Fowler describe una gran cantidad de patrones orientados a la arquitectura de aplicaciones empresariales. La visión de Fowler es más pragmática y esta en sintonía con la visión de arquitectura que se establece en [Welicki04]. En esa obra se da la siguiente definición de arquitectura:

> *"…1) nivel más alto de descomposición de un sistema en sus partes; 2) cosas que son difíciles de cambiar…"*

[18] Patterns of Enterprise Application Architecture (conocido también como PEAA). Un detalle interesante para destacar de este libro [Fowler02] es que se provee código fuente en Java y/o C# de todos los patrones

En PEAA [Fowler02] se definen las siguientes categorías de patrones[19]:

- **Layering:** patrones para dividir un sistema en capas.

- **Organización de la lógica del dominio**: formas de organizar los objetos del dominio.

- **Mapping to Relational Databases**: se relaciona con la comunicación entre la lógica del dominio y los repositorios de datos. Incluye el mapeo entre modelos de objetos y bases de datos relacionales. En la actualidad, se consume mucho tiempo de desarrollo en la realización de estas tareas debido a las diferencias de impedancia entre SQL y los lenguajes orientados a objetos como Java, C#, C++, etc.

- **Presentación Web**: la presentación Web es uno de los desafíos que han tenido que sortear en los últimos años las aplicaciones empresariales. Los clientes delgados Web proveen muchas ventajas, siendo una de las principales la facilidad de distribución (no es necesario instalar software en los equipos cliente). Esta categoría incluye una serie de patrones para gestionar la presentación Web.

- **Concurrencia**: manejo de la concurrencia. Las aplicaciones actuales basadas en tecnologías Web tienen grandes necesidades de gestión de la concurrencia.

- **Estado de Sesión**: patrones para el manejo de la sesión de servidores Web.

- **Estrategias de Distribución**: distribución de objetos en múltiples emplazamientos, basado en conocimientos empíricos en clientes.

En la tabla a continuación se muestra la distribución de los patrones definidos en *Patterns of Enterprise Application Architecture* [Fowler02] en las categorías mencionadas arriba:

[19] Es interesante destacar que algunos de estos patrones no coinciden con la descripción de arquitectura del WWISA [WWISA] o de Bredemeyer [Bredemeyer01] [Bredemeyer02] [Bredemeyer02b], dado que tienen un alcance local, aunque si coinciden con la definición que da Fowler en el capitulo 1 de su libro y que hemos citado en el párrafo anterior.

Domain Logic Pattern	**Mapping to Relational Databases**	**Web Presentation Patterns**	**Distribution Patterns**	**Offline Concurrency Patterns**	**Session State Pattern**	**Base Patterns**
Transaction Script Domain Model Table Module Service Layer	*Data Source Architectural Patterns* Table Data Gateway Row Data Gateway Active Record Data Mapper *Object Relational Behavioral Patterns* Unit of Work Identity Map Lazy Load *Object Relational Structural Patterns* Identity Field Foreign Key Mapping Association Table Mapping Dependent Mapping Embedded Value Serialized LOB Single Table Inheritance Class Table Inheritance Table Inheritance Concrete Inheritance Inheritance Mappers *Object-Relational Metadata Mapping Patterns* Metadata Mapping Query Object Repository	Model View Controller Page Controller Front Controller Template View Transform View Two Step View Application Controller	Remote Façade Data Transfer Object	Optimistic Offline Lock Pesimistic Offline Lock Coarse-Grained Lock Implicit Lock	Client Session State Server Session State Database Session State	Gateway Mapper Layer Supertype Separated Interface Registry Value Object Money Special Case Plugin Service Stub Record Set

Tabla 3.3 – Clasificación de patrones de arquitectura de aplicaciones empresariales de [Fowler02].

3.3.2 Ejemplo: El Patrón Modelo-Vista-Controlador

El patrón MODEL-VIEW-CONTROLLER[20] (en adelante MVC) fue introducido inicialmente en la comunidad de desarrolladores de Smalltalk-80 [Smalltalk]. MVC divide una aplicación interactiva en tres áreas: procesamiento, salida y entrada [POSA96]. Para esto, utiliza las siguientes abstracciones:

- **Modelo (Model)**: encapsula los datos y las funcionalidades. El modelo es independiente de cualquier representación de salida y/o comportamiento de entrada. [POSA96]

- **Vista (View)**: muestra la información al usuario. Obtiene los datos del modelo. Pueden existir múltiples vistas del modelo. Cada vista tiene asociado un componente controlador. [POSA96]

- **Controlador (Controller)**: reciben las entradas, usualmente como eventos que codifican los movimientos o pulsación de botones del ratón, pulsaciones de teclas, etc. Los eventos son traducidos a solicitudes de servicio[21] para el modelo o la vista. El usuario interactúa con el sistema a través de los controladores. [POSA96]

La Vistas y los Controladores conforman la interfaz de usuario. Un mecanismo de propagación de cambios asegura la consistencia entre la interfaz y el modelo.

La separación del modelo de los componentes vista y del controlador permite tener múltiples vistas del mismo modelo. Si el usuario cambie el modelo a través del controlador de una vista todas las otras vistas dependientes deben reflejar los cambios. Por lo tanto, el modelo notifica a todas las vistas siempre que sus datos cambios. Las vistas, en cambio, recuperan los nuevos datos del modelo y actualizan la información que muestran al usuario. [POSA96]

En la figura 3.2 a continuación se muestra la estructura del patrón MVC.

[20] Modelo-Vista-Controlador
[21] En el texto original [POSA96] "service request"

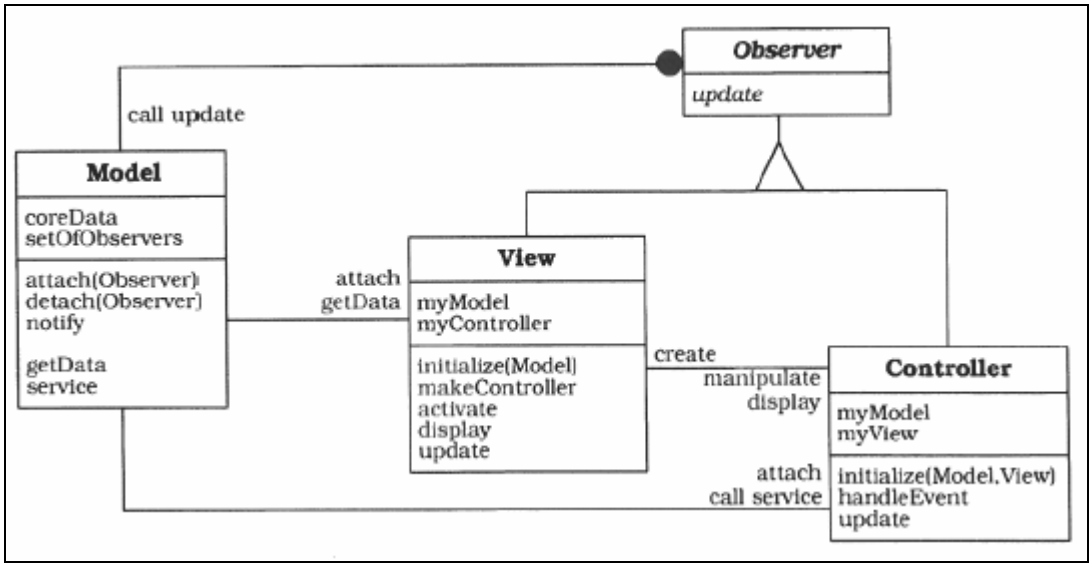


Figura 3.2 – Diagrama de clases de MVC [POSA96]

Este patrón es muy popular y ha sido portado a una gran cantidad de entornos y marcos de trabajo como Struts [Struts], Swing [SJFC], WinForms [MSWF], ASP.NET [MSAN], etc. Las herramientas de programación visual como Visual Basic [MSVB], Delphi [BID], JBuilder [BJB], etc. siguen también este esquema.

El MVC es un patrón ampliamente utilizado en múltiples plataformas y lenguajes. Algunos de sus principales beneficios son:

- Menor acoplamiento
  - Desacopla las vistas de los modelos
  - Desacopla los modelos de la forma en que se muestran e ingresan los datos

- Mayor cohesión
  - Cada elemento del patrón esta altamente especializado en su tarea (la vista en mostrar datos al usuario, el controlador en las entradas y el modelo en su objetivo de negocio)

- Las vistas proveen mayor flexibilidad y agilidad.
  - Se pueden crear múltiples vistas de un modelo.
  - Se pueden crear, añadir, modificar y eliminar nuevas vistas dinámicamente.
  - Las vistas pueden anidarse.
  - Se puede cambiar el modo en que una vista responde al usuario sin cambiar su representación visual.
  - Se pueden sincronizar las vistas.
  - Las vistas pueden concentrarse en diferentes aspectos del modelo.

- Mayor facilidad para el desarrollo de clientes ricos en múltiples dispositivos y canales

  - Una vista para cada dispositivo, que puede varias según sus capacidades
  - Una vista para la Web y otra para aplicaciones de escritorio.

- Más claridad de diseño.
  - Facilita el mantenimiento.

- Mayor escalabilidad.

3.3.3 Patrones de Diseño en el MVC

Un patrón de arquitectura puede contener varios patrones de diseño [Welicki05e]. A modo de ejemplo, citaremos el caso del patrón de arquitectura MVC (analizado en el apartado anterior) que contiene (o puede contener) los siguientes patrones de diseño:

- OBSERVER para el mecanismo de publicación y suscripción que permite la notificación de los cambios en el modelo a las vistas.

- COMPOSITE para la creación de vistas compuestas. Utilizando este patrón podemos crear una jerarquía de vistas y tratar a cada vista compuesta igual que una a una vista normal.

- STRATEGY en la relación entre las vistas y los controladores. Utilizando este patrón podemos cambiar dinámicamente (o en tiempo de compilación) los algoritmos del controlador mediante los cuales responde a su entorno.

- FACTORY METHOD parar especificar la clase controlador predeterminada de una vista.

- DECORATOR para añadir capacidades adicionales a una vista (como por ejemplo scroll).

- PROXY para distribuir la arquitectura (Modelo y Vista-Controlador) en diferentes emplazamientos.

Para más detalles sobre este caso particular de composición de patrones, ver el capítulo 1 de [GoF95] o [POSA96].

3.4 Antipatrones

Los antipatrones son soluciones negativas que presentan más problemas que los que solucionan. Son una extensión natural a los patrones de diseño. Comprender

los antipatrones provee el conocimiento para intentar evitarlos o recuperarse de ellos [BMMM98].

El estudio de los antipatrones permite conocer los errores más comunes relacionados con la industria del software. La obra más popular en este campo es "*Antipatterns: Refactoring Software, Architectures and Projects in Crisis*", publicada en 1998 [BMMM98].

Los antipatrones se documentan con cierto cinismo, lo cual los hace bastante graciosos y fáciles de recordar. Los nombres siempre aluden al problema que tratan con humor.

Se documentan mediante una plantilla[22] (como los patrones de diseño) que incluye secciones para documentar el la solución origen (que es la causa del problema), el contexto, las fuerzas en conflicto y las soluciones correctas propuestas (para más detalles, ver el capítulo 3 de [BMMM98]).

Un buen antipatrón explica por qué la solución original parece atractiva, por qué se vuelve negativa y como recuperarse de los problemas que ésta genera. Según Jim Coplien, un antipatrón es "algo que se ve como una buena idea, pero que falla en malamente cuando se implementa" [Hillside].

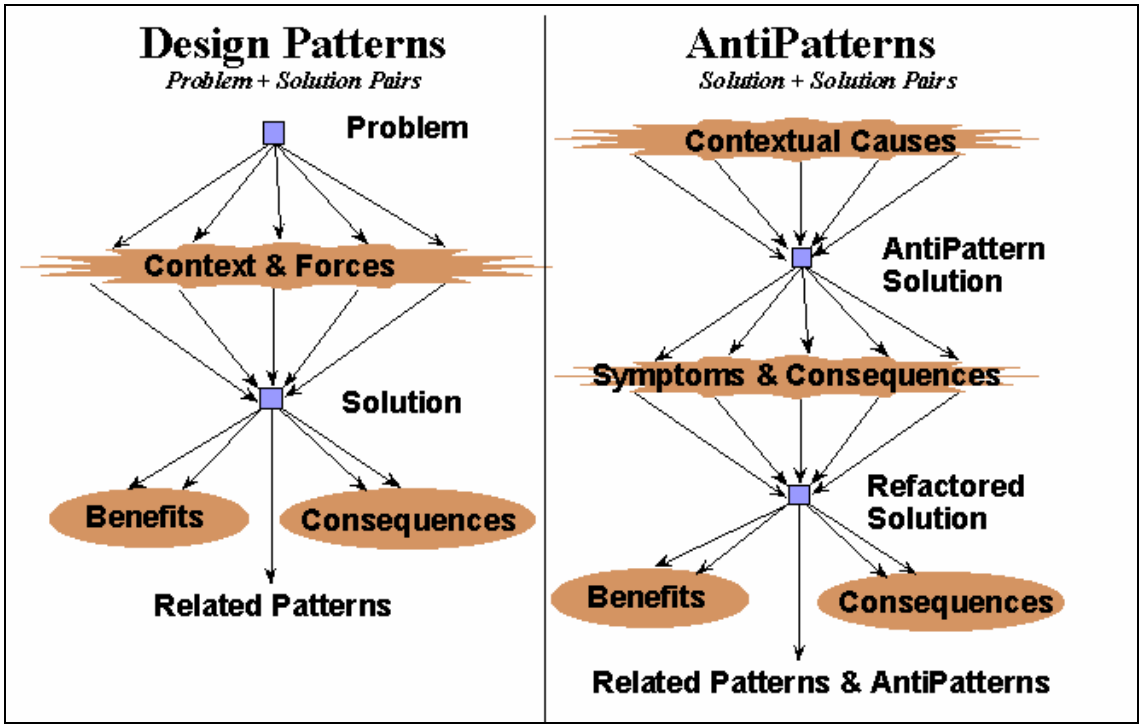


Figura 3.3 – Patrones y antipatrones. Notar como en los antipatrones se parte desde una solución (que es la que genera el problema), mientras que en los patrones se parte de un problema a resolver (figura tomada de [McCormick98])

[22] A modo de comentario personal, la plantilla usada en el libro es muy verborrágica y más extensa de lo necesario, lo cual resulta irritante a raíz de las críticas de los autores a otras plantillas

3.4.1 ¿Por qué Estudiar Antipatrones?

Un AntiPattern (o Antipatrón en castellano) es una forma literaria que describe una solución común a un problema que genera decididamente consecuencias negativas. Según [BMMM98], los antipatrones:

- Son un método eficiente para vincular una situación general a una clase de solución específica.

- Proveen experiencia del mundo real para reconocer problemas recurrentes en la industria del software, ofreciendo también una solución para sus implicaciones más comunes.

- Establecen un vocabulario comuna para identificar problemas y discutir soluciones.

- Soportan la resolución holística de conflictos, utilizando recursos organizacionales a diferentes niveles.

3.4.2 Categorías de Antipatrones

En [BMMM98], se dividen a los antipatrones en tres grandes categorías[23] a las cuales denomina puntos de vista, a saber:

- **Desarrollo de Software:** se centran en problemas asociados al desarrollo de software a nivel de aplicación.

- **Arquitectura de Software:** se centran en la estructura de las aplicaciones y componentes a nivel de sistema y empresa.

- **Gestión de Proyectos de Software**[24]: en la ingeniería del software, más de la mitad del trabajo consiste en comunicación entre personas y resolver problemas relacionados con éstas. Los antipatrones de gestión de proyectos de software identifican algunos de los escenarios clave donde estos temas son destructivos para el proceso de software. [BMMM98]

En la tabla a continuación se muestra la distribución de los patrones definidos en [BMMM98] en las categorías mencionadas arriba:

[23] En ese mismo libro se plantea un modelo de referencia teórico que va mas allá de estas divisiones, para más detalles, ver los capítulos 2, 3 y 4 de esa obra.
[24] *Management* en el texto original

Desarrollo de Software	Arquitectura	Gestión
The Blob	Stovepipe Enterprise	Analysis Paralysis
Lava Flow	Vendor Lock-in	Death by Planning
Functional Decomposition	Architecture by Implication	Corncob
Poltergeists	Design by Committee	Irrational Management
Golden Hammer	Reinvent the Wheel	Project Missmanagement
Spaghetti Code		
Cut and Paste Programming	*Mini antipatterns*	*Mini antipatterns*
	Autogenerated Stovepipe	Blowhard Jamboree
Mini antipatterns	Jumble	Viewgraph Engineering
Continuous Obsolescence	Cover your Assets	Fear of Success
Ambiguous Viewpoint	Wolf Ticket	Intellectual Violence
Boat Anchor	Warm Bodies	Smoke and Mirrors
Dead End	Swiss Army Knife	Throw it over the wall
Input Kludge	The Grand Old Duke of York	Fire Drill
Walking through a Minefield		The Feud
Mushroom Management		E-Mail is Dangerous

Tabla 3.4 – Clasificación de antipatrones [BMMM98]

3.5 Otros Tipos de Patrones…

Los patrones pueden encontrarse en todas las áreas de la ingeniería informática. En esta sección enumeraremos una serie de áreas donde existen patrones aceptados y conocidos en la industria. En cada caso indicaremos una referencia a alguna publicación al respecto:

- **Idiomas**: son específicos del lenguaje de programación. Describen como implementar ciertos aspectos de un problema utilizando las características de un lenguaje de programación [POSA96].

- **Patrones de Análisis**: los patrones enumerados en el libro *Analysis Patterns* [Fowler96] provienen de diversos dominios, incluyendo el área de la salud, servicios financieros y contabilidad. Cada uno de los patrones se describen en forma textual y con una simple notación pre-UML.

- **Patrones de Integración de Aplicaciones**: patrones para integración de aplicaciones. La obra más popular al respecto es *Enterprise Integration Patterns* [HW03] de Gregor Hophe y Bobby Wolf.

- **Patrones de UI**: patrones referentes interfaces de usuarios. Existen distintas categorías bien diferenciadas: algunas se encargan de detalles relacionados con la cognición, memoria a corto plazo y mejoras en la experiencia del usuario, mientras que otros describen técnicas de ingeniería para crear interfaces de usuario.

- **Patrones de Pruebas**: patrones para diseñar y realizar pruebas.

3.5.1 ¡Patrones en Todas Partes!

Al momento de escribir este trabajo, una búsqueda de la palabra "*pattern*" en Google retorna 341.000.000 de resultados[25]. Evidentemente no todos los resultados se refieren a patrones de software, pero en la primera página de resultados aparecen sitios emblemáticos de patrones de software como Hillside, Portland Pattern Repository y PatternLanguage.com.

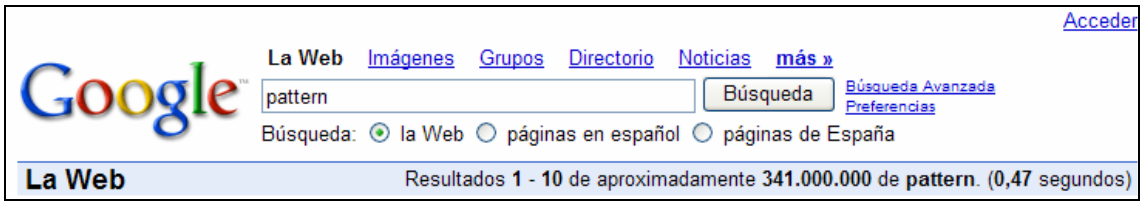


Figura 3.4 – Búsqueda del término "pattern" en el buscador Google [Google].

Actualmente, hay disponible una amplia literatura sobre patrones. De hecho, las principales casas de software como Microsoft y Sun (entre otras) tienen sus publicaciones sobre patrones que se implantan directamente sobre sus tecnologías [Microsoft03], [Microsoft04], [SBPC].

3.6 Iniciativas Empresariales

En esta sección, se analizan brevemente algunas iniciativas en el ámbito de los patrones de tres de las principales corporaciones de software existentes a nivel global: IBM, Microsoft y Sun Microsystems.

3.6.1 Microsoft Patterns and Practices

En los últimos años, Microsoft ha realizado grandes esfuerzos en esta área. Uno de ellos ha sido la contratación de mentados miembros de la comunidad de patrones, entre los que destaca Ward Cunnigham. Adicionalmente, entre los revisores de varias de sus guías y documentos figuran nombres como los de Martin Fowler y Robert C. Martin, entre otros.

El grupo de arquitectura prescriptiva (PAG) [MSPP], ha tenido gran actividad en los últimos años. Entre todas sus actividades se encuentran el diseño y la construcción de "*building blocks*" de código abierto disponibles para los desarrolladores. Estos bloques proveen soluciones a problemas que no son soportados[26] directamente por el framework .NET.

[25] "Software patterns" arroja 124.000.000 resultados, "design patterns" 168.000.000 y la palabra "patterns" (plural de pattern) 414.000.000.
[26] También pueden ser mejoras sobre funcionalidades existentes

3.6.2 IBM: Design Patterns Project

IBM ha sido quizás históricamente la organización más activa en esta comunidad. Un aspecto digno de mención es que dos de los cuatro miembros del GoF eran empleados de IBM (Richard Helm y John Vlissides) al momento de redacción del libro "*Design Patterns*" [GoF95]. Adicionalmente, uno de los iniciadores de la comunidad de patrones, Bruce Anderson, era también empleado de esta corporación. Actualmente, en la Web de su área investigación (IBM Research), IBM tiene un sitio dedicado a su proyecto de patrones[27].

Si bien en su página no se hace referencia directa a ningún producto específico, esta compañía utiliza ampliamente a los patrones en sus distintas líneas de productos. Es por esto que es difícil mencionar productos concretos derivados de esta área de investigación. Quizás sea pertinente mencionar productos como XDE, WebSphere y los diferentes marcos de trabajo[28] que ofrece IBM. Otra interesante aportación es el prototipo de generación de código a partir patrones de diseño desarrollado por el equipo de John Vlissides, cuya difusión ha sido realizada mediante el artículo "*Automatic Code Generation from Design Patterns*" [BFVY96].

3.6.3 SUN: Blueprints Patterns Catalog

Esta iniciativa de SUN [SBPC] está enfocada a describir y compartir patrones para facilitar la construcción de aplicaciones en entornos Java y J2EE. El sitio Web que expone el catálogo contiene una amplia variedad de información referente a patrones que permiten construir soluciones eficientes y robustas con estas tecnologías.

[27] http://www.research.ibm.com/designpatterns/
[28] Frameworks

Capítulo 4
Plantillas para la Descripción de Patrones

"Encontrar patrones es más fácil que describirlos"
GoF

Los patrones se describen utilizando un formato consistente [GoF95], [POSA96], [Fowler96] [Fowler02]. Las plantillas [C2PatternForm] dan una estructura uniforme a la información, haciendo que los patrones sean más fáciles de describir y por consiguiente de comprender y utilizar. En el Portland Pattern Repository [PPR] se hace la siguiente reflexión acerca de las plantillas[29] de los patrones:

> *"Una plantilla (PatternForm) es la estructura literaria de la descripción escrita de un patrón. La plantilla más sencilla es 'ThereforeBut'. El libro 'Design Patterns' utiliza una plantilla más detallada y estructurada. Es interesante estudiar a las plantillas (opuestamente a sólo estudiar a los patrones) debido a que son herramientas pedagógicas aplicables más allá de la arquitectura de software."*

En este capítulo, analizaremos las plantillas de patrones[30] utilizadas en las obras más populares de la literatura de patrones. En cada caso, mencionaremos la obra o el contexto de referencia y luego enumeraremos y explicaremos sus componentes.

❖ ❖ ❖

[29] También se las conoce como "forms", que puede traducirse como "formas" o "formularios". En este capítulo utilizaremos en forma indistinta la palabra "form" y "template"
[30] "pattern templates" en el texto original.

4.1 Lista de Plantillas

En este capítulo analizaremos a las siguientes plantillas:

- **GoF**: plantilla del libro "*Design Patterns: Elements of Reusable Object Oriented Software*", de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides [GoF95]. Dada la gran popularidad de esta obra, es una de las plantillas más extendidas[31].

- **POSA**: plantilla del libro "*Pattern Oriented Software Architecture Volume 1: A System of Patterns*", de Frank Buschmann et al. [POSA96]. Esta es otra obra de referencia, por lo cual estas plantillas gozan de amplia aceptación.

- **PEAA**: plantilla del libro "*Patterns of Enterprise Application Architecture*", de Martin Fowler [Fowler02]. Esta es quizás la más sucinta y pragmática de todas, destacando por su brevedad y claridad, pero también por su falta de formalidad.

- **DPE**: plantilla usada en el libro "*Design Patterns Explained*", de Shalloway y Trot [ST01]. Es un intermedio entre la del GoF y la de POSA, pasando también por la Forma Canónica.

- **AP**: plantilla usada en el libro "*Analysis Patterns: Reusable Object Models*", de Martin Fowler [Fowler96]. Esta plantilla destaca por su falta de formalismo y su absoluto pragmatismo.

- **AUPL**: plantilla usada en el libro "*A UML Pattern Language*", de Paul Evitts[32] [Evitts00]. Esta plantilla es una forma reducida de la forma Canónica.

- **Forma Canónica**: plantilla canónica de patrones, popularizada por James Coplien. Esta plantilla goza de amplia aceptación y podría decirse que puede ser el supertipo de todas las demás plantillas.

- **SBPP:** plantilla usada en el libro "*Smalltalk Best Practices Patterns*" [Beck96], de Kent Beck.

- **Alexandrian**: plantilla utilizada por Christopher Alexander en "*A Pattern Language*" [AIS77]. Esta plantilla es la más antigua (Alexander es el iniciador

[31] Si bien esta es una de las plantillas más extendidas (por ser la que se utiliza en el libro más vendido) no es una de las más populares: en la comunidad de patrones se realizan críticas frecuentes a esta plantilla. Otra muestra clara de su baja aceptación es el bajo grado de utilización de esta plantilla en los patrones que se presentan cada año en las variantes de las conferencias PLoP [Hillside]

[32] Esta obra no es muy popular pero la hemos incluido porque consideramos que la plantilla que utiliza es muy interesante para ser analizada en este capítulo.

del movimiento de patrones, aunque en el campo de la arquitectura civil) y es el punto de partida de todas las demás. Destaca por su foco y claridad.

4.1.1 Design Patterns (GoF)

- **Nombre del patrón y clasificación**. El nombre del patrón transmite sucintamente su esencia. Un buen nombre es vital, porque pasará a formar parte de nuestro vocabulario de diseño. La clasificación del patrón refleja el esquema de clasificación.

- **Propósito**. Una frase breve que responde a las siguientes preguntas: ¿Qué hace este patrón de diseño? ¿En qué se basa? ¿Cuál es el problema concreto de diseño que resuelve?

- **También conocido como**. Otros nombres por los que se conoce a este patrón (si existen).

- **Motivación**. Un escenario que ilustra un problema de diseño y cómo las estructuras de clases y objetos del patrón resuelven el problema. El escenario ayuda a entender la descripción que sigue.

- **Aplicabilidad**. ¿En que situaciones se puede aplicar el patrón de diseño? ¿Qué ejemplos hay de malos diseños que el patrón puede resolver? ¿Cómo se puede reconocer dichas situaciones?

- **Estructura**. Una representación gráfica de las clases del patrón usando una notación basada en OMT[33] [RBP91]. También se hace uso de diagramas de interacción [JCJO92] para mostrar secuencias de peticiones y colaboraciones entre objetos.

- **Participantes**. Las clases y objetos participantes en el patrón de diseño, junto con sus responsabilidades.

- **Colaboraciones**. Como colaboran los participantes para llevar a cabo sus responsabilidades.

- **Consecuencias**. ¿Cómo consigue el patrón sus objetivos? ¿Cuáles son las ventajas e inconvenientes y los resultados de usar el patrón? ¿Qué aspectos de la estructura del sistema se pueden modificar en forma independiente?

- **Implementación**. ¿Cuáles son las dificultades, trucos o técnicas que deberíamos tener en cuenta a la hora de aplicar el patrón? ¿Hay cuestiones específicas del lenguaje?

[33] Object Modeling Techinque (OMT)

- **Código de ejemplo**. Fragmentos de código que muestran como se puede implementar el patrón en C++ o Smalltalk.

- **Usos conocidos**. Ejemplos del patrón en sistemas reales. Incluye al menos dos ejemplos de diferentes dominios.

- **Patrones relacionados**. ¿Qué patrones de diseño están estrechamente relacionados con éste? ¿Cuáles son las principales diferencias? ¿Con qué otros patrones debería usarse?

4.1.2 Pattern Oriented Software Architecture (POSA)

- **Nombre**. Nombre y breve resumen del patrón.

- **También conocido como**. Otros nombre para el patrón, si hay alguno conocido.

- **Ejemplo**. Ejemplo del mundo real, demostrando la existencia del problema y la necesidad del patrón.

- **Contexto**. Situaciones en las cuales puede aplicarse el patrón.

- **Problema**. El problema que resuelve el patrón, incluyendo una discusión de sus fuerzas asociadas.

- **Solución.** El principio fundamental de solución subyacente del patrón.

- **Estructura.** Especificación detallada de los aspectos estructurales del patrón, incluyendo tarjetas CRC para cada participante y diagramas de clase OMT [RBP91].

- **Dinámica.** Escenarios que describen el comportamiento en tiempo de ejecución del patrón, ilustrados utilizando diagramas de secuencia.

- **Implementación.** Guías para implementar el patrón. Son sólo sugerencias y no reglas inmutables. Deben ser adaptadas para ajustarse a las necesidades del usuario, modificando, quitando, reordenando o añadiendo nuevos pasos. En las series POSA [POSA96], [POSA00], [POSA03] se incluye código en C++, Java y Smalltalk, entre otros.

- **Ejemplo resuelto**. Discusión de cualquier aspecto importante para resolver el ejemplo que no han sido cubiertas todavía en las secciones de *Solución*, *Estructura*, *Dinámica* e *Implementación*.

- **Variantes**. Breve descripción de las variantes o especializaciones del patrón.

- **Usos conocidos**. Ejemplos de utilización del patrón, tomados de sistemas existentes.

- **Consecuencias.** Beneficios de la utilización del patrón, como así también posibles inconvenientes.

- **Ver también.** Referencias a patrones que resuelven problemas similares y a patrones que ayudan a refinar el patrón que se esta describiendo.

4.1.3 Patterns of Enterprise Application Architecture (PEAA)

- **Nombre**. Nombre del patrón. El nombre es crucial, parque parte del objetivo de los patrones es crear un vocabulario que permita a los diseñadores comunicarse en forma más efectiva.

- **Propósito**. Resumen del patrón en una o dos oraciones.

- **Sketch**. Representación visual del patrón, generalmente utilizando UML[34] [UML].

- **Motivación**. Problema que motiva al patrón. No es necesariamente el problema que resuelve el patrón, pero si motiva su existencia.

- **Cómo funciona**. Describe la solución. Se discuten temas de implementación y variaciones. Se intenta que la discusión sea independiente de la plataforma de implementación.

- **Cuándo utilizarlo**. Describe cuándo debe ser utilizado el patrón. Se incluyen los trade-offs que hacen que se elija esta solución en lugar de otra.

- **Lecturas adicionales**. Apuntador a discusiones adicionales sobre este patrón. No pretende ser una bibliografía comprehensiva, sino que son referencias a recursos que pueden ser significativos para comprender e implementar el patrón.

- **Ejemplos**. Bloques de código de ejemplo del patrón, en Java o C#.

4.1.4 Design Patterns Explained (DPE)

- **Nombre**. Todos los patrones tienen un nombre único que los identifica.

[34] Unified Modeling Language

- **Propósito**. Propósito del patrón.
- **Problema**. El problema que el patrón está intentando resolver.
- **Solución**. Cómo el patrón brinda una solución al problema en el contexto en que se produce.
- **Participantes y colaboradores.** Entidades involucradas en el patrón.
- **Consecuencias**. Las consecuencias de utilizar el patrón. Investiga las fuerzas que participan en el patrón.
- **Implementación**. Cómo puede ser implementado el patrón (las implementaciones son sólo manifestaciones concretas del patrón y no deben ser tomadas como el patrón en sí).
- **Referencia en GoF.** Dónde encontrar el patrón en el libro del GoF [GoF95].

4.1.5 A UML Pattern Language (AUPL)

- **Nombre**. Nombre del patrón.
- **Problema**. Problema que intenta resolver el patrón. Permite saber fácilmente la utilidad del patrón.
- **Contexto.** Provee más ayuda para determinar la aplicabilidad del patrón.
- **Fuerzas**. Fuerzas que debe balancear el patrón.
- **Solución**. Solución al problema, en el contexto establecido y con las fuerzas especificadas.
- **Contexto resultante.** Contexto resultante luego de aplicar el patrón.
- **Discusión**. Discusión sobre temas relacionados con el patrón y su aplicación.

4.1.6 Analysis Patterns[35] (AP)

- **Título.** Nombre del patrón.

[35] También conocida como "Forma de Fowler" (Fowler Form), en honor a su autor, Martin Fowler.

- **Resumen**. Resumen del patrón.

- **Discusión**. Discusión sobre los inconvenientes que se evitan al aplicar el patrón y como el patrón ayuda a evitarlos.

- **Ejemplos**. Ejemplos de aplicación del patrón.

4.1.7 Forma Canónica[36]

- **Nombre**. El nombre del patrón. Según James Coplien, debe ser elegido "*con el mismo cuidado que el de un hijo*". Los nombres de los patrones deben ser sucintos, fáciles de recordar y relevantes.

- **Alias (opcional).** Otros nombres con los que se conoce a este patrón.

- **Problema**. Breve descripción del problema que se resuelve con el patrón.

- **Contexto**. La situación que da a lugar al problema.

- **Fuerzas**. Descripción de las fuerzas relevantes y restricciones.

- **Solución**. Solución al problema (previamente probada).

- **Ejemplo (opcional).** Aplicaciones de ejemplo del patrón.

- **Contexto resultante (opcional).** Estado del sistema luego de que se aplica el patrón.

- **Fundamento[37] (opcional).** Explicación de pasos o reglas en el patrón.

- **Usos conocidos.** Ocurrencias y aplicaciones del patrón en sistemas del mundo real.

- **Patrones relacionados.** Patrones con relaciones estáticas y dinámicas con este patrón.

4.1.8 Smalltalk Best Practice Patterns[38] (SBPP)

- **Título**. Nombre del patrón.

- **Contexto (opcional).** La situación que da a lugar al problema.

[36] También conocida como "*Forma de Coplien*", en honor a su autor James Coplien.
[37] "Rationale" en el texto original
[38] También conocida como "*Forma de Beck*", en honor a su autor, Kent Beck.

- **Problema**. Siempre expresada en la forma de pregunta que el lector debe contestar por sí mismo, como por ejemplo, "*¿Cómo puedo ordenar una colección?*"

- **Fuerzas**. Descripción de las fuerzas relevantes y restricciones.

- **Solución**. Solución al problema en el contexto. La solución debe incluir el nombre del patrón en alguna forma.

- **Contexto resultante**. Contexto resultante luego de aplicar la solución del patrón.

4.1.9 Alexandrian Form

- **Nombre**. Un nombre o frase corta descriptiva[39], usualmente más indicativa de la solución que del ***problema*** o del ***contexto***.

- **Ejemplo**. Una o más fotos, diagramas y/o descripciones que ilustran la aplicación prototípica.

- **Contexto.** Delineación de situaciones en que el patrón es aplicable. Generalmente incluye el trasfondo, discusiones de por qué este patrón existe y evidencias, para generalidad.

- **Problema**. Descripción de las fuerzas y restricciones relevantes y cómo interactúan. En muchos casos, se centran casi enteramente en restricciones del problema sobre las que el lector probablemente jamás ha pensado. Los asuntos de diseño y construcción forman parte a veces de las restricciones.

- **Solución**. Relaciones estáticas y reglas dinámicas que describen cómo construir artefactos de acuerdo al patrón, generalmente enumerando varias variantes y/o formas de ajustar las circunstancias. Las soluciones referencias y relaciones otros patrones de mayor y menor nivel.

[39] Algunos ejemplos incluyen Alcoves, Main entrance, Public outdoor room, Parallel roads, Density rings, Office connections, Sequence of sitting spaces, e Interior windows.

Capítulo 5
Gestión del Conocimiento

"Si HP supiera lo que HP sabe,
seríamos tres veces más redituables"
Lewis Platt, Gerente General de Hewlett-Packard

El conocimiento es uno de los activos principales de las organizaciones actuales [DP99]. Esto lo convierte en un claro generador de valor que debe ser correctamente gestionado. La gestión del conocimiento se refiere a entregar a las personas los datos e información necesarios para ser eficientes en sus trabajos en cualquier lugar y momento, con cualquier dispositivo [Joyanes03].

En este capítulo comenzaremos por realizar un análisis sobre el conocimiento y su gestión, a efectos de establecer las bases para nuestro estudio posterior, incluyendo la definición del concepto de conocimiento, las dimensiones y ciclo de creación del conocimiento [NT94], y la gestión del conocimiento [Joyanes03]. Proseguiremos con una aplicación de estos conceptos a la ingeniería del software, estableciendo un caso práctico aplicado a las empresas de servicios de consultoría [Welicki03b].

❖ ❖ ❖

5.1 El Conocimiento y su Gestión

Comenzaremos con uno de los temas, quizás, más complejos de tratar: el conocimiento y su gestión. En esta sección trataremos temas tales como las definiciones de los términos "conocimiento" y "gestión del conocimiento". El objetivo no es dar una definición exacta de estos conceptos sino mostrar la variedad de definiciones que existen para definirlos y sentar las bases conceptuales para nuestro análisis posterior.

5.1.1 Conocimiento

A continuación, presentaremos distintas definiciones de diversos autores sobre el concepto *conocimiento*. No es el objetivo principal definir este concepto sino mostrar la variedad de definiciones existentes para luego establecer una definición que servirá como marco conceptual para este trabajo:

- "*Creencia verdadera o justificada*" (Platón, en [NT94])
- *"Información más contexto"* (Luis Joyanes Aguilar) [Joyanes03]
- "*Es la capacidad para transformar datos e información en acciones efectivas.*" (J.D. Edwards, citado en [Joyanes03])
- "…*El conocimiento es una mezcla fluida de experiencia estructurada, valores, información contextual e internalización experta que proporciona un marco para la evaluación e incorporación de nuevas experiencias e información. Se origina y se aplica en la mente de los conocedores.*" (Davenport y Prusak, en [DP99])
- "*Es un proceso humano dinámico de justificación de la creencia personal en busca de la verdad.*" (Nonaka y Takeuchi, en [NT94])
- "*El conocimiento es la existencia de un modelo en un sistema cibernético, que le permite hacer predicciones, que es, anticipar procesos en su entorno. Por lo tanto, el sistema toma control sobre su entorno. Tal modelo es una construcción personal, no un reflejo objetivo de la realidad externa.*" (Principia Cybernetica Web, en [PCW])

Como es posible apreciar fácilmente, existen diversas definiciones sobre este término. Tomemos algunas de las palabras más relevantes: *contexto*, *experiencia*, *dinámico*, *verdad*, *objetividad*, *predicción*, *personal*, *construcción*.

A modo de conclusión de esta sección, y a efectos de sentar las bases de estudio posterior, intentaré dar una definición integradora [Welicki03]:

> *"El conocimiento es un elemento intrínseco de las personas. Esta a un grado de abstracción y agregación semántica mayor que la información. Para que una información pueda convertirse en conocimiento, la persona destinataria de la misma debe poder darle significado (comprensión) dentro de un contexto. Es un término con alto grado de subjetividad, ya que esta determinado por el contexto y la experiencia del agente observador que realiza su abstracción (en este aspecto, podríamos establecer un paralelismo con el concepto de relatividad de la Teoría General de los Sistemas [Bertalanffy68])." [Welicki03]*

En la imagen a continuación (figura 5.1), se ofrece una representación gráfica de la tríada dato-información-conocimiento y su relación con los conceptos de abstracción, componente semántica y objetividad. Como se puede apreciar, el nivel de abstracción y la componente semántica tiene sentido inverso con el nivel objetividad.

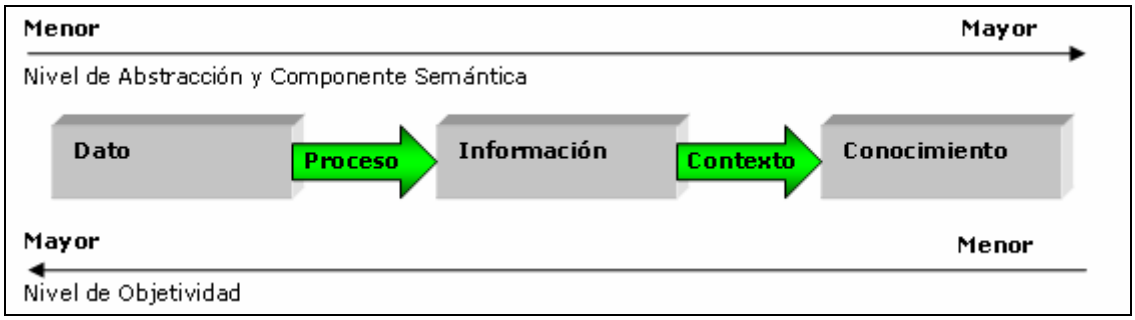


Figura 5.1 – Progresión entre dato-información-conocimiento. La objetividad seria una flecha en sentido inverso a de la abstracción y la semántica.

5.1.1.1 Tipos de Conocimiento

En "*The Knowledge Creating Company*" [NT94], Nonaka y Takeuchi identifican dos tipos de conocimiento, el tácito y el explícito. A continuación, definiremos a cada uno de ellos [Joyanes03]:

- **Conocimiento tácito** [Joyanes03], Es el empleado por el sujeto cognoscente, pero sin que éste sea capaz de identificar claramente qué conocimiento está utilizando ni como lo usa. Es aquel que el trabajador del conocimiento conoce y que se deriva de la experiencia, de sus valores habilidades, etc. No es posible su representación y, por consiguiente no se puede recoger en un soporte físico.

- **Conocimiento explícito** [Joyanes03], Es aquel que quien lo utiliza es plenamente consciente del mismo y de cómo lo usa. Se puede codificar y representar mediante expresiones de carácter verbal, numérico, gráfico, imágenes estáticas o dinámicas. Se recoge en distintos tipos de soporte: impreso, audio o vídeo, código informático, documentos, vídeos; se crea con el objeto de comunicarse entre personas.

5.1.1.2 Dimensiones de Creación de Conocimiento

Para graficar las dimensiones de creación de conocimiento dentro del modelo, tomaremos como base el modelo de creación de conocimiento de Nonaka y Takeuchi [NT94], donde la creación del conocimiento se a través de en dos ejes: el epistemológico y el ontológico (figura 5.2).

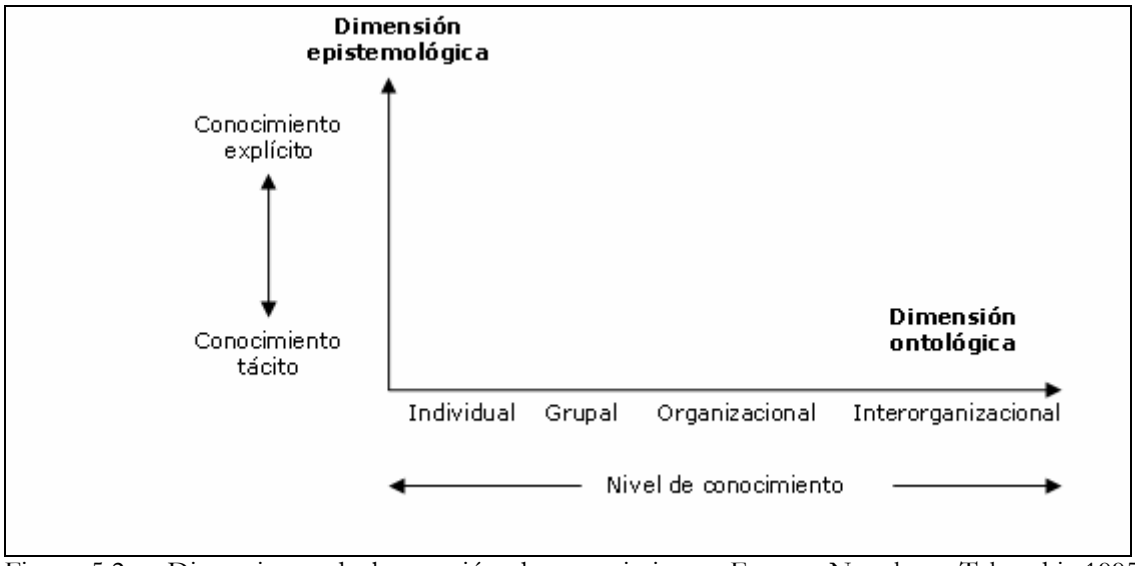


Figura 5.2 – Dimensiones de la creación de conocimiento. Fuente: Nonaka y Takeuchi, 1995 (adaptado de [NT94])

Las cuatro formas de conversión de conocimiento, que conforman la espiral de creación de conocimiento, se dan en estas dos dimensiones. Para más detalles, ver el capítulo 3 de *The Knowledge Creating Company* [NT94]

5.1.1.3 Ciclo de Conversión del Conocimiento

De acuerdo con el modelo dinámico de creación de conocimiento (fundamentado en la espiral de creación de conocimiento), defendido por Nonaka y Takeuchi [NT94], para la creación de conocimiento organizacional es necesario, en primer lugar, el conocimiento tácito de los miembros de la organización, pues constituye la base de ésta. En segundo lugar, la organización precisa movilizar y ampliar el conocimiento tácito acumulado por cada individuo, creando el conocimiento organizacional. [BS03]

Para que sucedan los procesos de movilización y ampliación de conocimiento, los autores defienden que debe existir una interacción social entre el conocimiento tácito y el explícito, similar al que acontece con el conocimiento humano. A esta integración es lo que denominan "*conversión de conocimiento*". [BS03]

Dicha conversión se realiza mediante cuatro procesos, que se enumeran a continuación:

- **Socialización (de tácito a tácito)**: es un proceso que consiste en compartir experiencias y por tanto crear conocimiento tácito tal como los modelos mentales compartidos y las habilidades técnicas. Un individuo puede adquirir conocimiento tácito de otro sin usar el lenguaje. La clave para obtener el conocimiento tácito es la experiencia. Sin alguna forma de experiencia compartida, a una persona le resulta extremadamente difícil proyectarse a sí misma al interior de otra persona. [NT94]

    - **Tecnologías**: Reuniones-e, Colaboración síncrona (chat), Mensajería Instantánea [Joyanes03]

- **Exteriorización (de tácito a explícito)**: es un proceso a través del cual se enuncia el conocimiento tácito en forma de conceptos explícitos. Es un proceso esencial en la creación de conocimiento en el que el conocimiento tácito se vuelve explícito y adopta forma de metáforas, analogías, conceptos, hipótesis o modelos. Cuando intentamos conceptuar una imagen, expresamos casi siempre su esencia usando el idioma[40]. Un método muy utilizado para crear conceptos es combinar la deducción y la inducción. [NT94]

    - **Tecnologías**: Visualización, presentaciones por navegación de audio/video [Joyanes03]

- **Combinación (de explícito a explícito)**: es un proceso de sistematización de conceptos con el que se genera un sistema de conocimiento. Esta forma de conversión de conocimiento implica la combinación de distintos cuerpos de conocimiento explícito. Los individuos intercambian y combinan conocimientos a través de distintos medios[41]. La reconfiguración de la información existente que se lleva a cabo clasificando, añadiendo, combinando y categorizando el conocimiento explícito puede conducir a nuevo conocimiento. La creación de conocimiento que se da en las escuelas y al entrenamiento formal, por lo general, adopta esta forma. [NT94]

    - **Tecnologías**: Responder a preguntas, anotaciones [Joyanes03]

- **Interiorización (de explícito a tácito)**: es un proceso de conversión de conocimiento tácito a explícito y esta muy relacionada con el "aprender haciendo". Cuando las experiencias son internalizadas en la base de conocimiento tácito de los individuos a través de la socialización, la exteriorización y la combinación, en la forma de modelos metales compartidos y know-how técnico, se vuelve activos muy valiosos. [NT94]

    - **Tecnologías**: Buscar texto, "categorización de documentos", Portales, Software de Colaboración y de "e-learning" [Joyanes03]

[40] Las capacidades de expresión varían según el medio. Un análisis más profundo de esto puede encontrarse en "El precio de las palabras en la sociedad de la información" [Welicki04]

[41] Nuevamente, remitimos al trabajo "El precio de las palabras en la sociedad de la información" [Welicki04]

En la figura 5.3 a continuación se muestran las cuatro formas de conversión de conocimiento contextualizadas:

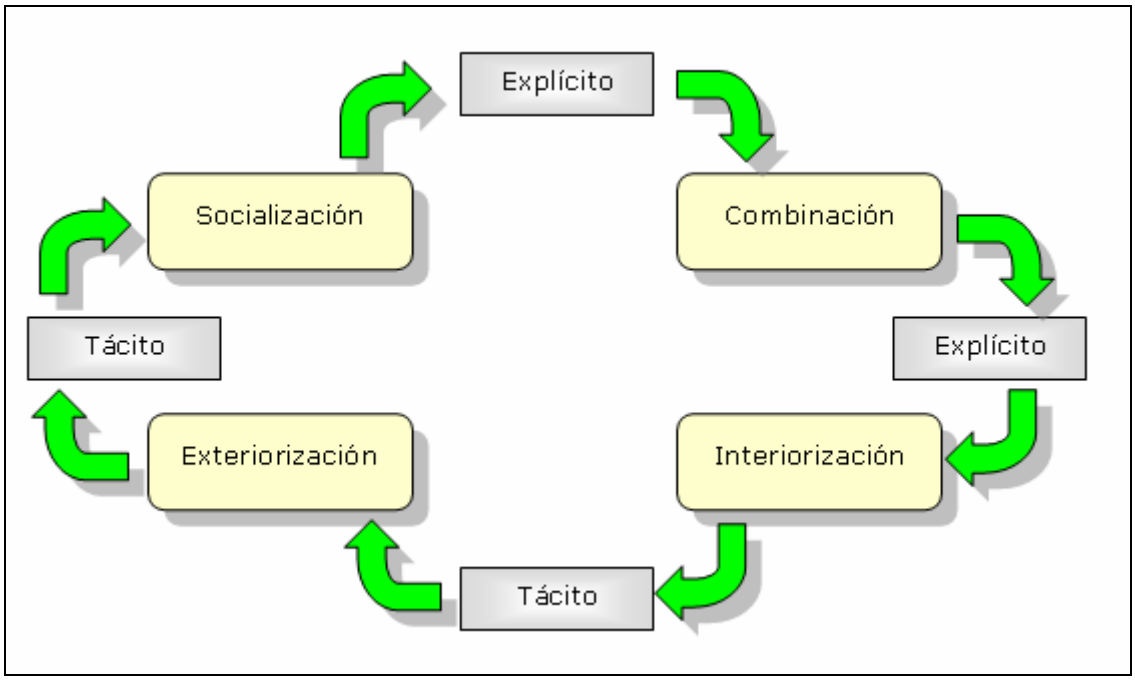


Figura 5.3 – Conversión del conocimiento en la organización, según Nonaka y Takeuchi [NT94]

5.1.3.4 La Espiral de Creación del Conocimiento

Para comenzar una nueva espiral de creación de conocimiento es necesario que el conocimiento tácito acumulado sea socializado con otros individuos de la organización haciendo viable entonces la creación de conocimiento organizacional. Los contenidos de conocimiento generados en las cuatro formas de conversión interactúan entre sí en una espiral de creación de conocimiento organizacional, generando una nueva espiral y así sucesivamente [BS03].

En la figura a continuación se muestra la espiral completa, con las etapas del proceso de conversión ubicadas en los dos ejes mencionados anteriormente (epistemológicos y ontológicos).

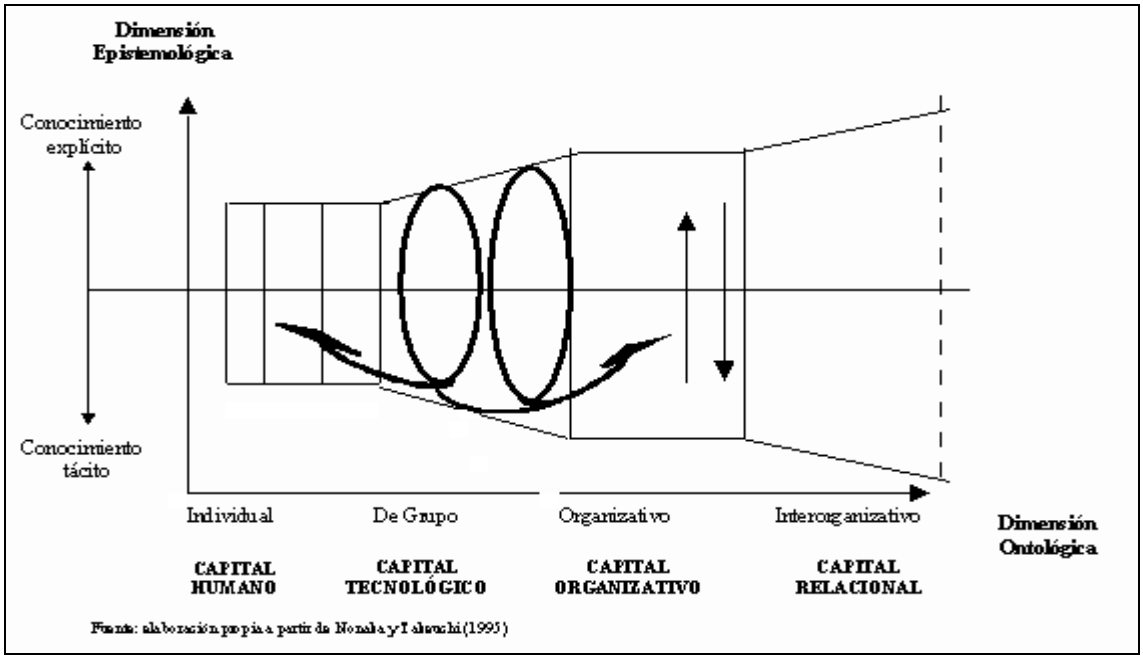


Figura 5.4 – Espiral de creación de conocimiento, tomada de [Garcia01]

Para más detalles sobre el modelo completo, ver "*The Knowledge Creating Company*", de Nonaka y Takeuchi [NT94].

5.1.2 Gestión del Conocimiento

Ya hemos definido, en la sección anterior, que es el *conocimiento*, a través de definiciones de diversos autores para luego concluir con una definición propia. En esta sección, haré lo propio con el concepto *gestión del conocimiento*. En forma análoga al concepto tratado en el párrafo anterior, existe una gran variedad de definiciones para definirlo, donde cada una hace hincapié en distintos aspectos. A los efectos de nuestro trabajo, definiré este concepto a través de dos definiciones, a saber:

- *"Entregar a las personas los datos e información necesarios para ser eficientes en sus trabajos en cualquier lugar y momento, con cualquier dispositivo"* (Luis Joyanes Aguilar, en [Joyanes03])

- *"La gestión del conocimiento provee a las características críticas de adaptación organizacional, supervivencia y competencia de cara a los cambios discontinuos en el entorno… Esencialmente, agrupa procesos organizacionales que buscan combinaciones sinérgicas de datos y capacidades de procesamiento de información de las tecnologías de la información, y la capacidad creativa e innovadora de los seres humanos"* (Yogesh Maholtra, [Maholtra98])

5.1.2.1 ¿Por qué Gestionar el Conocimiento?

El conocimiento es uno de los activos principales de las organizaciones actuales [DP99]. De hecho, varios de los más ricos del mundo (entre ellos Bill Gates de

Microsoft y Larry Ellison de Oracle) provienen de la industria del software, un claro campo de conocimiento.

En [Joyanes03], se enumeran los siguientes beneficios asociados a la gestión del conocimiento:

- Facilidad de relaciones con los socios, clientes,...
- Gestión de la experiencia del volumen de negocios / ventas...
- Descentralización en la toma de decisiones.
- Simplicidad y conocimiento.
    - Pirámide de conocimiento: datos, información, conocimiento.
    - La simplicidad implica proporcionar directrices, conceptos y principios para asimilación de datos nuevos e información como conocimiento.
    - La importancia del pensamiento sobre el conocimiento en términos de simplicidad es que se pueden:
        - Aplicar conceptos similares, recurriendo a situaciones que produzcan efectos predecibles.
        - Aplicar conceptos a nuevas situaciones para producir innovaciones.
        - Refinar directrices a través de aplicaciones repetidas y verificación.

Otros beneficios de gestionar el conocimiento:

- El encuentro de una diferenciación estratégica.
- Ser capaces de producir nuevo conocimiento mediante la experiencia, las aptitudes y el cambio de actitud en la cultura organizacional.
- Mejorar la comunicación.
- Identificar y calificar las fuentes de conocimiento y ser capaces de transferirlo eficazmente.
- Estar en condiciones de poder medir los resultados a partir de los datos, información y conocimiento dentro y fuera de la organización.
- Acortar los tiempos en los proyectos de planeamiento.
- Optimizar los procesos, incrementando la productividad.
- Utilizar en mayor grado los recursos existentes dentro de la organización.
- Posibilitar la creación de un círculo virtuoso entre el aprendizaje individual y el de la organización en pleno.

5.1.2.2 La tecnología en la GC

La tecnología es sólo un facilitador[42] y no la solución en si [Joyanes03]. Alfons Cornella, de Infonomía dice en su último libro "*sin espacio social, no existe el espacio digital*" [Cornella03]. Siguiendo con esta tesis, se recomienda revisar el artículo "*La*

[42] Enabler (en ingés)

Infoestructura: Un concepto esencial en la sociedad de la información" [Cornella98], del mismo autor donde profundiza en esta aserción.

5.2 Gestión del conocimiento en proyectos de software

5.2.1. ¿Qué Conocimiento Queremos Gestionar?

No se busca "abrir la cabeza de las personas y extraer todo su conocimiento a una base de datos", ya que consideramos que esto no aporta ninguna solución al problema planteado inicialmente. Lo que necesitamos es una herramienta de bajo costo de producción, mantenimiento y alimentación que brinde el soporte necesario para generar un mercado de conocimiento en red. Intentaremos gestionar conocimiento corporativo referente a los siguientes temas:

- Quiénes son las personas trabajan en nuestra organización de desarrollo.
- Qué competencias tienen las personas que trabajan en nuestra organización.
- Las lecciones aprendidas a lo largo de diversos proyectos y relaciones con clientes (observación empírica).
- Base de conocimientos técnicos y mejores practica recomendadas.

Como resultado de dicha gestión, se pretende obtener las siguientes mejoras corporativas:

- Conocer a nuestros empleados y sus competencias.
- Brindar un marco técnico corporativo (política) que sirva de guía a nuestros empleados.
- Brindar alto valor agregado al cliente a través de una red de empleados que intercambian conocimientos para solucionar problemas.
- Conocer cuál es nuestro capital y potencial intelectual.
- Poder gestionar mejor los proyectos, armando equipos de trabajo en forma más eficiente.

5.2.2 El Antipatrón de las Islas de Funcionalidad

En muchos casos las empresas de consultoría de software no gestionan correctamente los activos de conocimiento de sus empleados [Welicki03b]. En el campo de los antipatrones de software (§ 3.4), existe uno (varios en realidad) que se adapta perfectamente a esta situación: el de las islas de funcionalidad [BMMM98], donde cada equipo (que puede ser unipersonal) desarrolla y soluciona problemas con total desconocimiento de los demás, siguiendo la metáfora de islas inconexas. Esto provoca que en cada caso se deba pensar la solución desde cero, generando una amplia diversidad de soluciones para problemas similares.

Extrapolando esto último a la situación que nos ocupa, cada empleado en esta modalidad generalmente es una de las metafóricas islas de las mencionadas en el

párrafo anterior. Cada vez que alguien tiene que hacer algo, reinventa la rueda [BMMM98] debido al propio desconocimiento de lo que hay hecho en la casa (en este caso la consultora). Esto tiene las siguientes connotaciones negativas:

- Las buenas soluciones no se registran al nivel corporativo y no se reproducen.
- Los empleados no pueden reutilizar componentes existentes en la organización por desconocimiento de los mismos.
- Ante una situación desconocida, el empleado debe preguntar como resolver dicha situación al cliente, dado que es su punto de contacto y no conoce a nadie en la consultora que pueda ayudarlo. Esto atenta contra el valor agregado que la consultora ofrece como tal (en este caso, es nulo o podría decirse que tiende a menos infinito).
- El equipo no ofrece ningún tipo de valor añadido.

5.2.3 Proyectos Irrepetibles

En este contexto, los proyectos, se parecen muy frecuentemente a las obras de arte: son irreproducibles. Respecto a los proyectos, hemos identificado la siguiente taxonomía[43] [Welicki03b]:

- En muchos casos, lamentablemente, la presupuestación, análisis y diseño se realizan en forma totalmente ad-hoc, sin seguir un procedimiento o metodología específico y por supuesto, sin generar documentación reutilizable.

- En otros casos, los proyectos se realizan en forma más ordenada, pero no se genera documentación reutilizable.

- Y finalmente, en otros casos, todo se hace como se debe, pero la empresa no colabora ni ofrece el marco adecuado para que la documentación y las lecciones aprendidas se extiendan a todos sus integrantes.

Por supuesto que existe también el caso donde todo se hace como corresponde, pero ese caso esta fuera de nuestro dominio de análisis. Cada uno de los casos es altamente perjudicial para la empresa, dado que está tirando por la borda su principal activo: el conocimiento generado por sus integrantes.

5.2.4 La Paradoja del Conocimiento

En muchos equipos de desarrollo y empresas de consultoría y servicios informáticos, se suele dar frecuentemente una paradoja que oscila entre lo grotesco y lo cómico [Welicki03b]:

[43] Esta clasificación es muy parecida a los niveles de los modelos CMM y SPICE

> *"Se intenta obtener máximo beneficio de las personas en términos inmediatos, pero se desprecia todo beneficio generado a partir del conocimiento del empleado." [Welicki03b]*

Según Peter Drucker (citado en [Joyanes03]) el saber es hoy el único recurso significativo. Los tradicionales factores de producción (suelo, recursos naturales, mano de obra y capital) se han convertido en secundarios; pueden obtenerse y con facilidad siempre que haya saber.

5.2.5 Solución: Aplicar Gestión del Conocimiento

Los problemas planteados anteriormente pueden ser resueltos utilizando técnicas de gestión del conocimiento. En [Welicki03b] se presenta un modelo de gestión del conocimiento sencillo y de bajo coste de implementación para resolver este problema, cuya arquitectura se muestra en la figura 5.5 a continuación.

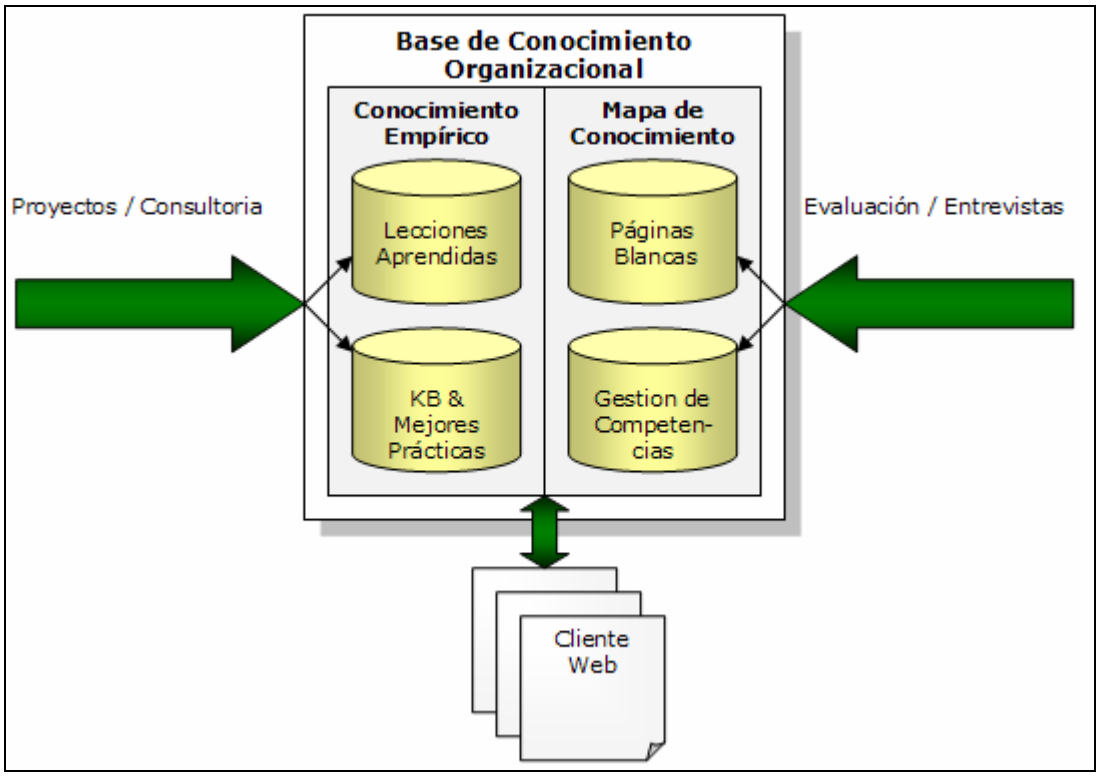


Figura 5.5 – Arquitectura de alto nivel de la solución de gestión de conocimiento para equipos de desarrollo de software, tomada de [Welicki03b]

El resultado de la aplicación de las herramientas es una base de conocimiento organizacional constituida por dos elementos fundamentales:

- **Mapa de Conocimiento**, que es el soporte fundamental del mercado de conocimiento. Esta conformado por:
  - **Paginas blancas**, directorio de empleados con su información de contacto y currículo resumido.

- **Gestión de competencias**, repositorio de las competencias de cada empleado, tanto técnicas como no-técnicas.

- **Conocimiento Empírico**, que es el soporte para la normalización de procesos y reutilización del conocimiento empírico generado por la empresa. Esta conformado por:

  - **Lecciones aprendidas**, repositorio de experiencias recabadas a lo largo de los diferentes proyectos y actividades en clientes.

  - **Mejores prácticas**, repositorio de prácticas recomendadas en diferentes ámbitos tecnológicos y de negocios.

Mediante la implantación del modelo propuesto se busca conectar a todos los empleados con la empresa y establecer los canales apropiados para que se comuniquen entre ellos para intercambiar conocimiento y experiencia (empleado-empleado y empleado-empresa), como se muestra en la figura a continuación.

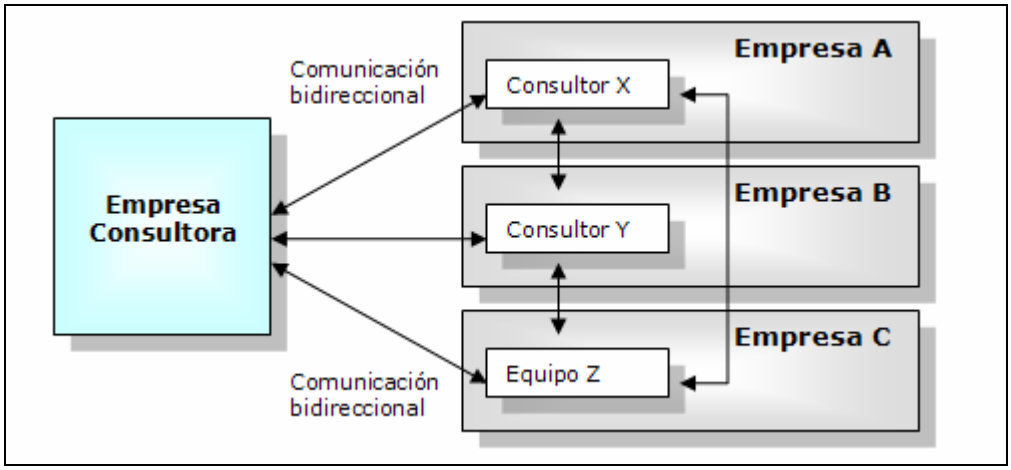


Figura 5.6 – Aplicación del modelo de gestión de conocimiento a una empresa de consultoría informática, tomada de [Welicki03b]

Las líneas que comunican a las distintas entidades en el gráfico son bidireccionales e implementan la comunicación necesaria para dar soporte al modelo propuesto. En la tabla a continuación, se especifica el significado de la comunicación en cada dirección.

De / A	Empleado	Empresa
Empleado	Páginas blancas y gestión de competencias: Intercambio de conocimiento y experiencias a través del mercado de conocimiento.	Actualización de datos (envió de mejores prácticas candidatas o datos empíricos recabados en un cliente). Actualización de sus datos personales o competencias.
Empresa	Conocimiento Empírico: Políticas, base de conocimiento y mejores prácticas	Necesidad de hacer revisiones para actualizar el mapa de conocimiento o la base empírica

Tabla 5.1 – Líneas de comunicación entre los empleados y la empresa.

Capítulo 6
Meta-Niveles en la Ingeniería Informática

"En el arte existen sencilleces
más dificiles que las
complejidades más enrevesadas"
Aldous Huxley

El prefijo Meta viene del campo de la filosofía e indica un nivel de descripción más elevado. A partir de las definiciones anteriores, podríamos decir que se refiere a "trascender o ir más allá de". Por ejemplo, si X es un concepto, entonces meta-X son datos acerca de, procesos u operaciones sobre X [TCDM]. Según el diccionario Merriam-Webster [MW], sus raíces etimológicas parten del latín o el griego.

En este capítulo analizaremos el nivel meta y dos meta-niveles actualmente importantes en la ingeniería informática y relevantes para esta tesis: los meta-datos y el meta-modelado. En este último caso nos centraremos en la arquitectura de cuatro niveles del OMG y en MDA [OMGMDA].

❖ ❖ ❖

6.1 Introducción

Para la mayoría de los científicos de la computación significa "acerca de" [McComb03] y se refiere a un nivel superior de abstracción. A modo de ejemplo podemos citar el caso de los *metadatos* que son "datos acerca de los datos".

Para clarificar este concepto, incluyo los siguientes ejemplos [TCDM]:

- Una meta-sintaxis es una sintaxis que especifica una sintaxis
- Un metalenguaje es un lenguaje utilizado par discutir lenguajes
- Metadatos son datos acerca de los datos
- Meta-razonamiento es razonamiento acerca del razonamiento

6.1.1 El Nivel Meta en la Informática

En muchos casos, cuando los informáticos resuelven un problema tienden a "*ir al nivel meta*". Generalmente, esto significa resolver el problema en forma más general, a un mayor nivel de abstracción, lo cual implica una solución más flexible y con mayores capacidades de adaptación a cambios en su entorno (dicho entorno no es universal, sino que esta inmerso en un contexto determinado por la semántica y abstracciones asociadas al nivel meta desarrollado).

Pero estos beneficios no son gratuitos: este tipo de soluciones tienen un mayor grado de complejidad y penalidades de rendimiento. Es por esto que el paso a un nivel meta debe ser correctamente analizado, sin dejar de tener en cuenta la relación coste/beneficio y la necesidad de negocio que se desea cubrir.

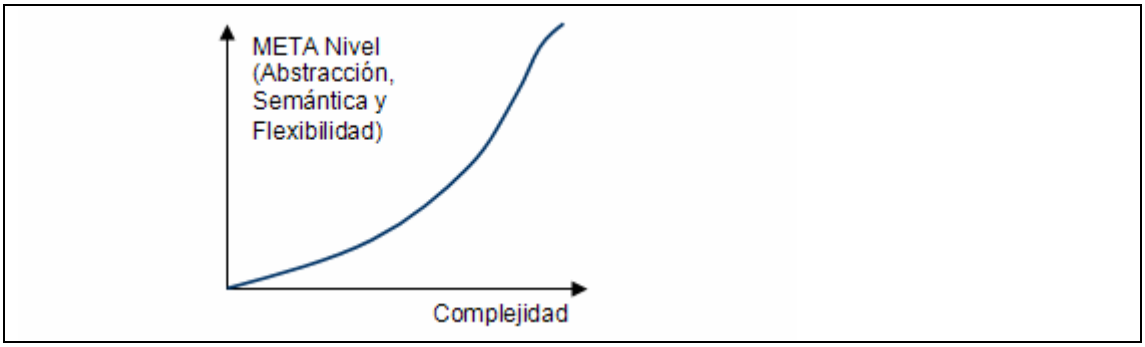


Figura 6.1 – Meta niveles y complejidad. Notar como la complejidad aumenta con el nivel meta, aunque no en forma directamente proporcional. Esta relación es aproximada, a los efectos de explicar la relación entre los dos ejes.

6.1.2 Subiendo Niveles

Al subir un nivel, cambiamos el nivel de abstracción y la carga semántica de sus elementos. Estos nos brindan mayor flexibilidad y posibilidad de separar los datos

del sistema de la implementación (de los algoritmos que los manipulan). Esto nos permite *poner las abstracciones en el código y los detalles en los metadatos* [HT00]

Cuando se trata con niveles "meta", es importante saber cuando parar. Es posible definir un gran número de niveles de abstracción, pero en muchos casos puede ser contraproducente. De hecho, se puede llegar hasta el nivel de átomo (el libro meta-patrones se dedica a analizar los patrones en la naturaleza, dentro del ámbito de la cibernética y la filosofía).

6.1.3 Cuidado con el Nivel Meta…

Según Ralph Johnson, uno de los miembros del GoF, "*ser seducido a ir al nivel meta es el equivalente en ciencias de la computación a ser seducido a ir al lado oscuro de la fuerza*" [C2MP]. Su argumento al respecto es que "*las abstracciones son muy buenas, pero necesitas experiencia concreta previa antes de que puedas entenderlas bien y mas aún, inventarlas. Es por esto que debemos enfocarnos en conocer patrones particulares antes de hacer una teoría de los patrones en general*". [C2MP]

El nivel meta suele añadir complejidad adicional a un modelo o implementación. Por ejemplo, cualquier base de datos podría diseñarse como una relación configurable con posibilidades de cambiar y adaptarse en tiempo de ejecución, lo cual la haría muy flexible pero muy difícil de utilizar, mantener y explotar (para más información sobre formalismos matemáticos de bases de datos relacionales ver [Johnsonbaugh97] y [KS93])

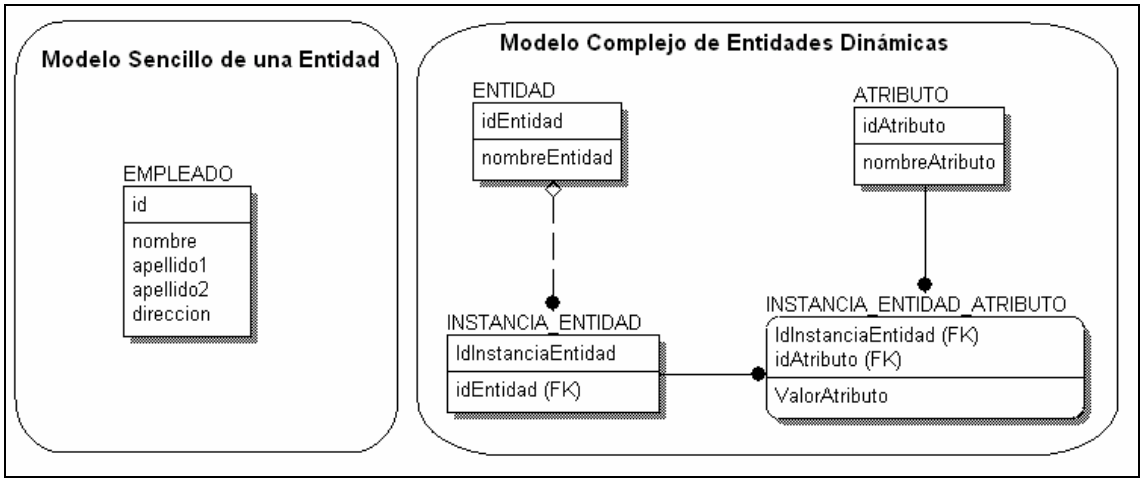


Figura 6.2 – Dos formas de modelar un mismo concepto. En el segundo caso, tenemos mayor flexibilidad a expensas de la transparencia, opacidad y facilidades de explotación. En el caso de nuestro ejemplo, generalmente el modelo completo con facilidades de meta-definición (que es el de la derecha) no suele ser recomendable, ya que añade un elevado nivel de complejidad innecesaria.

Es por esto que debemos ser cuidadosos al respecto y no caer en la trampa del exceso de niveles de abstracción e indirección. Para esto, es bueno siempre tener una buena visión de arquitectura del sistema a construir combinada con un buen

entendimiento del negocio y no olvidar que el *objetivo del sistema es dar soporte al negocio en tiempo y forma.*

Como ejemplo positivo, podemos citar el caso del Object Management Group (OMG) [OMG], que ha definido sólo cuatro niveles para expresar meta-modelos (sobre los que profundizaremos más adelante en este capítulo). Podrían haber continuado, pero consideraron que esa era la cantidad adecuada de niveles.

6.2 Meta-Datos

Los metadatos son "*Datos acerca de los datos*" [McComb03], [HT00], [DOS03]. Esta definición tiene una fuerte connotación semántica, lo cual implica que los metadatos agregan algún tipo de significado a los datos que describen. Los metadatos son un elemento fundamental para los sistemas inspirados en semántica [McComb03].

En la tabla 6.1, a continuación, se muestra a modo de ejemplo la diferencia entre datos y metadatos. Es fácilmente observable que los datos son los valores específicos en un dominio o contexto y que los meta datos expresan el significado o propósito de esos datos [DOS03].

Dato	**Meta Dato**
Leon Welicki	Nombre
Plaza Conde de Casal, 2	Dirección
Madrid	Ciudad
Madrid	Estado
28007	Código Postal

Tabla 6.1 – Datos contra metadatos, adaptada de [DOS03]

En los últimos años, los metadatos han pasado de ser un artefacto que se dejaba para el final (diccionario de datos) a un elemento central en las arquitecturas de software. En esta evolución se han pasado cronológicamente por los siguientes estadios [McComb03]:

- Diccionario de datos
- Base de datos de Diccionario de Datos
- Diccionario Activo
- Herramientas CASE
- Data Definition Language (DDL)
- Esquemas
- SGML
- XML
- DTD
- XML Schemas
- MOF

- RDF

Para una descripción detallada de cada uno de estos pasos evolutivos, ver "*Semantics in Business Systems*" [McComb03].

6.2.1 Metadatos y Flexibilidad

El comportamiento de los sistemas que contienen y procesan metadatos puede ser cambiado en tiempo de ejecución por los usuarios. Esto les da un mayor control del sistema que están utilizando, permitiéndoles "reprogramarlo" en tiempo de ejecución (dentro de unos límites establecidos) para que pueda cumplimentar requisitos cambiantes. El uso de metadatos y de las aplicaciones basadas en metadatos (§ 6.2.3) permite una mayor flexibilidad y aumenta la expresividad semántica del sistema.

6.2.1.1 Meta Meta Dato Dato

Es interesante la reflexión de Ralph Kimball en su articulo "*Meta Meta Data Data*" [Kimball98], refiriéndose a un datawarehouse*: los metadatos son todo excepto los datos*. Para llegar a esta conclusión, analiza la información de configuración, gestión, procesos y explotación del datawarehouse. Esto hace que súbitamente, los datos (que son el motivo de su existencia) se transformen en la parte más sencilla del problema.

6.2.2 Representación de Metadatos

Si bien no existe ninguna tecnología obligatoria para implementar metadatos (pueden implementarse usando ficheros de texto, bases de datos, ficheros ini[44], etc.), la opción más indicada y ampliamente utilizada es XML.

XML brinda una sintaxis estándar para metadatos [DOS03], ofreciendo una aproximación eficaz para describir la estructura y el propósito de los datos. Adicionalmente, normaliza las reglas sintácticas básicas de representación y los modelos de utilización (DOM, SAX, etc.). Varios de los formatos principales de representación de metadatos como RDF, OWL, XML Schemas, WSDL y XMI se basan en XML.

6.2.3 Aplicaciones de Metadatos

Aunque no lo sepamos (o no nos demos cuenta), estamos en constante contacto con los metadatos. A continuación, enumeraremos algunos campos muy populares donde tienen una fuerte participación.

[44] Este tipo de ficheros eran muy comunes en versiones antiguas de Windows

6.2.3.1 Reflectividad

La reflectividad o reflexión[45] es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo así como ajustar su comportamiento en función de ciertas condiciones [Ortin02]. Para esto, el sistema tiene una serie de metadatos asociados a cada elemento ejecutable. Es por esto que en [Lowy03] se define a la reflectividad como la acción programática de leer metadatos asociados a un tipo (reflectividad estructural) y a la posibilidad de definir nuevos tipos o modificar los existentes en tiempo de ejecución.

6.2.3.2 Bases de Datos Relacionales

Un esquema de base de datos se especifica utilizando un lenguaje llamada *Data Definition Language* (DDL) [KS93]. El resultado de la compilación de sentencias de DDL es un conjunto de tablas las cuales se almacenan en un diccionario de datos o directorio [KS93].

El directorio de datos es un archivo que contiene metadatos y se utiliza antes de leer o modificar los datos reales [KS93]. En bases de datos relacionales como Oracle [Oracle] o Microsoft SQL Server [MSSQL], estos datos se almacenan en tablas. Dichas tablas contienen información acerca de instancias de entidades (por ejemplo, que tablas existes, que columnas hay en cada tabla, el tipo de cada tabla, etc.). Estas tablas pueden ser consultadas también por los usuarios de la base de datos, siempre que tengan los permisos adecuados[46].

A continuación, enumeraremos algunas de estas tablas existentes en las bases de datos SQL Server (llamadas catálogos). La lista completa puede encontrarse en [MSDNSQL].

[45] Reflection (en inglés)

[46] Podríamos decir entonces que estos sistemas implementan reflectividad estructural, dado que esos metadatos aportan información acerca de la estructura de la base de datos. Las tablas del diccionario son también conocidas como metatablas.

Tabla	Descripción
SYSOBJECTS	Contiene una fila por cada objeto (*constraint*, *default*, *log*, *rule*, *stored procedure*, etc.) creado en la base de datos.
SYSINDEXES	Contiene una fila por cada índice y tabla en la base de datos. Esta tabla existe en todas las bases de datos.
SYSCOLUMNS	Contiene una fila por cada columna en todas las tablas, vistas y por cada parámetro en un procedimiento almacenado. Esta tabla existe en todas las bases de datos.
SYSREFERENCES	Contiene los mapeos de las definiciones de las *constraints* de *foreign key* con las columnas referenciadas. Esta tabla existe en todas las bases de datos.
SYSCONSTRAINTS	Contiene los mapeos de las *constraints* y los objetos a los que les pertenecen. Esta tabla existe en todas las bases de datos.
STSDATABASES	Contiene una fila por cada base de datos en un servidor Mictrosoft SQL Server™. Cuando se instala el SQL Server esta tabla contiene entradas para las bases de datos *master*, *model*, *msdb*, *mssqlweb* y *tempdb*. Esta tabla sólo existe en la base de datos *master*.

Tabla 6.2 – Principales catálogos de metadatos de SQL Server, adaptado de [MSDNSQL]

En las bases de datos Oracle también existen tablas con información del sistema. A continuación, enumeraremos algunas de ellas. Para una lista completa, ver [TON]

Tabla	Descripción
ALL_OBJECTS	Todos los objetos accesibles por el usuario
ALL_TABLES	Descripción de todas las tablas relacionales accesibles por el usuario
ALL_TAB_COLUMNS	Columnas de las tablas de usuario, vistas y clusters
ALL_TRIGGERS	Triggers (disparadores) accesibles por el usuario actual
ALL_VIEWS	Todas las vistas accesibles por el usuario
DBA_OBJECTS	Todos los objetos de la base de datos

Tabla 6.3 – Principales catálogos de metadatos de Oracle, adaptada de [TON].

6.2.3.3 Herramientas de ORM

ORM[47] es la sigla de Object-Relational Mapping y se refiere al mapeo de los objetos existentes en una aplicación desarrollada en un lenguaje OO (C++, Java, C#, etc.) a las tablas de una base de datos relacional (Oracle, DB2, SQL Server, Sybase, etc.).

En "*Patterns of Enterprise Application Architecture*" [Fowler02], se dedican dos capítulos a este tema y titula a uno de ellos "Object-Relational Metadata Mapping Patterns". Allí describe una serie de patrones que hacen uso intensivo de metadatos para realizar el mapeo. Uno de ellos, llamado METADATA MAPPING [Fowler02], sugiere el uso de metadatos y reflectividad[48] para la realización de esta tarea. Utilizando este patrón se reduce el esfuerzo necesario para realizar el mapeo entre los objetos de negocio y las tablas de la base de datos relacional.

Actualmente hay una gran cantidad de productos de software para ORM[49], muchos de ellos Open Source (código abierto). En su gran mayoría, utilizan los metadatos existentes en las bases de datos relacionales (descritos en la sección anterior) para crear automáticamente el código de persistencia o inclusive el código completo de la capa de acceso a datos.

6.2.3.4 HTML Meta Tags

HTML permite a los autores de páginas especificar meta datos mediante los tags META [W3C99]. Estos metadatos pueden ser utilizados por los buscadores. Por ejemplo, para especificar el autor de un documento HTML, se puede utilizar un elemento META como se muestra en el fragmento de código a continuación:

```
<META name="Author" content="León Welicki">
```

Código 6.1 – El elemento META especifica una propiedad (en este caso "Autor") y especifica un valor (en este caso "León Welicki")

6.2.3.5 Información de Configuración

La información de configuración de una aplicación puede ser vista como metadatos. Un sistema como SAP [SAP] tiene más de 10,000 opciones de configuración, que permiten modificar aspectos del sistema en forma dinámica. El resultado de esto es que se puede modificar la semántica del sistema modificando los valores de estos parámetros [McComb03].

[47] También se lo conoce como O/RM
[48] También propone el uso de generación de código (ver Metaprogramación en este capítulo)
[49] Los más destacados son Hibernate (para Java) [Hibernate] y NHibernate (para .NET) [NHibernate]

6.2.4 Meta Data-Driven Applications

Si bien los metadatos pueden ser muy útiles para definir opciones de configuración, esquemas y preferencias, se puede ir un paso más adelante e intentar que la aplicación pueda ser manejada a través de metadatos en tiempo de ejecución en el mayor grado posible.

Para esto, hay que cambiar el paradigma e intentar pensar en *forma declarativa* (*qué* es lo que hay que hacer y no *cómo*) a efectos de crear programas altamente dinámicos y adaptables [HT00]. La regla en este caso podría ser "*programar para el caso general y poner los detalles específicos en otro sitio*" [HT00]. Esto nos permitirá tener aplicaciones que puedan ser modificadas sin necesidad de tocar su código fuente o de realizar nuevos despliegues.

De esta forma, podemos obtener aplicaciones más flexibles y adaptables, con todos los beneficios que aporta el uso de metadatos (y también con sus inconvenientes, por supuesto). Es importante tener en cuenta lo expuesto anteriormente en "Cuidado con el nivel Meta" (§ 6.1.3).

6.2.4.1 Beneficios

- Fuerza a desacoplar el diseño, lo cual da como resultado un programa más flexible y adaptable
- Fuerza a crear un diseño más robusto y abstracto al diferir detalles (diferirlos fuera del programa)
- Permite personalizar la aplicación sin recompilarla o reinstalarla
- Los metadatos pueden ser expresados en un lenguaje mas cercanos al dominio del problema que los lenguajes de programación
- Facilita la reutilización de motores de ejecución (componentes, middleware, etc.) basados en esos metadatos.

6.3 Meta Modelado

Un *metamodelo* es un modelo de un modelo. Es también a su vez un modelo. Un modelo es una abstracción de algo que existe en la realidad. El meta modelado es el proceso de diseñar lenguajes mediante meta y meta-meta notaciones, las cuales ayudan a asegurar especificaciones sintácticamente correctas y a la construcción de diseños parametrizables.

6.3.1 La Arquitectura de Cuatro Capas del OMG

El Object Management Group (OMG) [OMG] ha definido una arquitectura de metamodelado de cuatro niveles para sus estándares. Cada capa se define con elementos de su capa superior y esta a un menor nivel de abstracción que ésta. En

la terminología del OMG se llaman M0, M1, M2 y M3 y se detallan en la tabla a continuación:

Capa	Descripción	Ejemplos
Meta-metamodelo (M3)	Es la infraestructura para la arquitectura de metamodelado. Define el lenguaje para especificar metamodelos.	MetaClass, MetaAttribute, MetaOperation
Metamodelo (M2)	Instancia de un Meta-metamodelo. Define un lenguaje para especificar un modelo.	Class, Attribute, Operation, Component
Modelo (M1)	Instancia de un metamodelo. Define un lenguaje para describir un dominio de información.	StockShare, askPrice, sellLimitOrder, StockQuoteServer
Objetos de Usuario (M0)	Una instancia de un modelo. Define un dominio de información específico.	<Acme_Software_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>

Tabla 6.4 – Niveles de modelado definidos por el Object Management Group

En principio, se podrían haber agregado mas niveles, pero encontraron que no esto no era muy útil [KWB03]. En lugar de definir un nivel M4, el OMG decidió que todos los elementos del nivel M3 deben ser definidos en términos de nivel M3.

Asumiendo que el meta-modelo es lo suficientemente expresivo, la arquitectura de cuatro niveles puede soportar la mayoría, sin todas, las formas imaginables de meta-información. Permite relacionar diferentes formas de metadatos y permite el intercambio de meta datos (modelos) y meta-meta datos (meta-modelos) [Bathia03].

6.3.1.1 M0: Instancias

En esta capa esta el sistema en ejecución y es donde existen las instancias reales. Las instancias, por ejemplo, pueden ser "*Juan Carlos*" que vive en "*Juan XXIII*" en "*Madrid*". Comúnmente, hay varias instancias distintas conviviendo en forma concurrente, cada una con sus propios datos. Cuando se modela un negocio y no software, las instancias en este nivel son los elementos en negocio, como por ejemplo la gente [KWB03].

6.3.1.2 M1: El Modelo del Sistema

Esta capa contiene modelos, como por ejemplo los diagramas UML del sistema de software. En este nivel el concepto "A*lumno*" se define con los atributos "*nombre*", "*dirección*" y "*ciudad*". Los elementos de M1 especifican directamente la forma de los de M0. Cada elemento de M0 debe ser una "*instancia de*" otro de M1 [KWB03].

6.3.1.3 M2: El Modelo del Modelo

Los elementos que existen en M1 (clases, atributos, etc.) son instancias de elementos de M2, el siguiente nivel superior. Los elementos en M2 especifican los de M1. La misma relación que existe entre los elementos de M0 y M1 existe entre los de M1 y M2. El modelo que reside en M2 se conoce como metamodelo. Cada modelo de UML definido en M1 es una instancia de un metamodelo, definido en [UML]. Cuando se construye un metamodelo en M2, se esta creando un lenguaje de modelado que puede utilizarse para escribir modelos [KWB03].

6.3.1.4 M3: El Modelo de M2

Describe los metamodelos que se describen en M2. Es la capa de mayor nivel de abstracción. Todos los elementos de los lenguajes de modelado de la capa M2 (y mismo los de M3) deberían poder definirse en términos de elementos de esta capa. Dentro del OMG, el Meta Object Facility (MOF), [OMGMOF] es el lenguaje de la capa M3. El MOF es un estándar que describe como deben describirse los metamodelos. UML, por ejemplo, es uno de los metamodelos descritos utilizando el MOF.

6.3.1.5 El Modelo Completo

En la figura a continuación se muestra un diagrama del modelo completo. Nótese que los elementos de nivel inferior son instancias de los conceptos de su nivel superior.

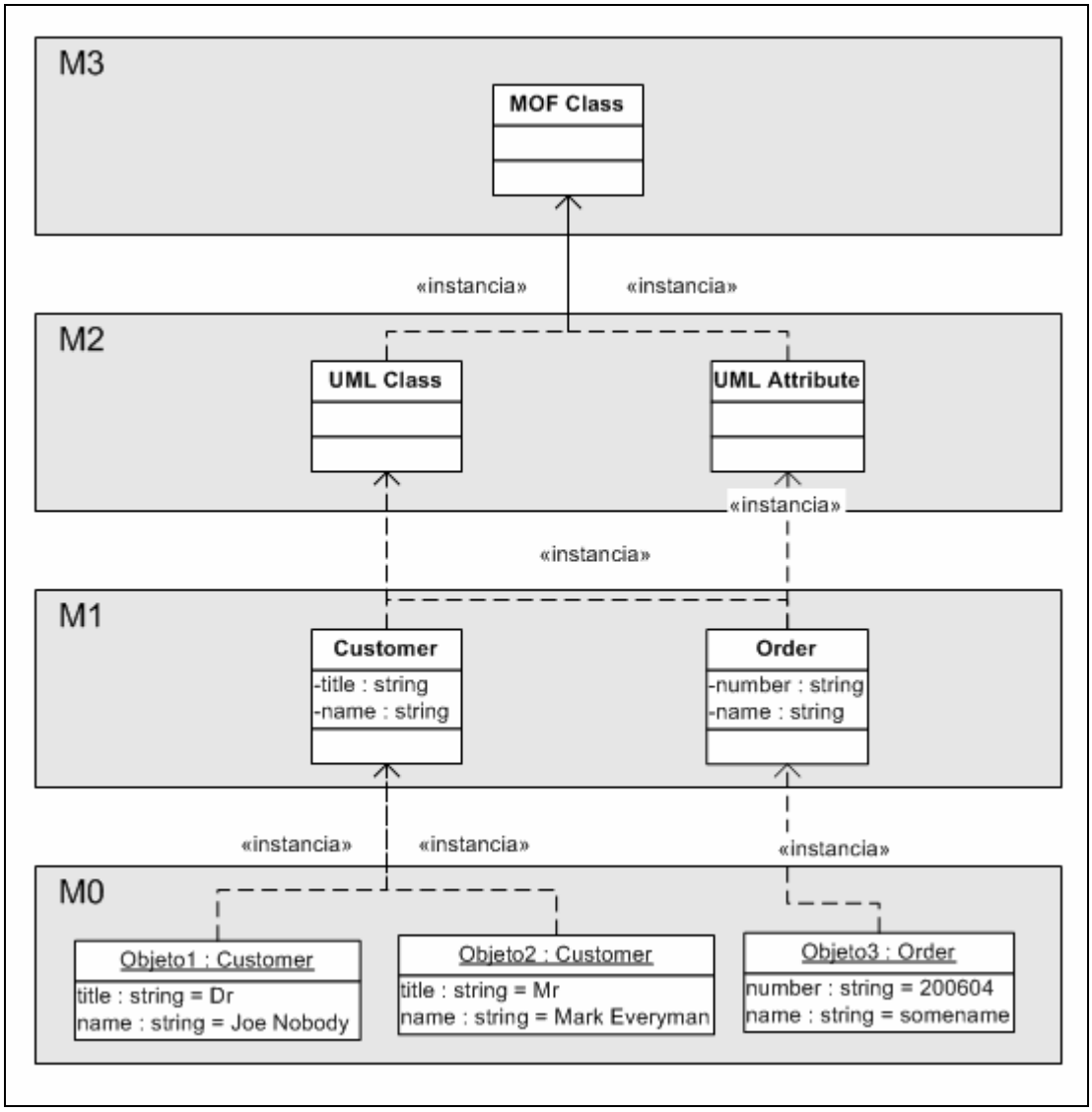


Figura 6.3 – Ejemplo de los 4 niveles de modelado, adaptado de [KWB03]

6.3.2 Ejemplo: Uso de Metamodelado en MDA

MDA [OMGMDA] significa *Model Driven Architecture*. Es un framework para el desarrollo de software creado por el OMG [OMG]. En MDA el proceso de software es guiado por el modelado.

En MDA se parte de un modelo independiente de la plataforma (Platform Independent Model, PIM), que se encarga de modelar requisitos y funciones de negocios. Mediante una transformación bien definida (especificada en un metanivel, como por ejemplo el MOF), este modelo puede transformarse en un modelo de plataforma específica (Plaftorm Specific Model o PSM), que se ajusta a los términos de construcciones de implementación (como por ejemplo un diagrama de entidad relación para base de datos, un diagrama de clases o un diagrama de navegación Web). Finalmente, el PSM se transforma en alguna forma de código ejecutable (utilizando transformaciones basadas en un lenguaje bien

definido) que puede ser ejecutado. Todas las transformaciones mencionadas se realizan en forma automática mediante herramientas.

El objetivo de MDA es generar un sistema de software completo en forma automática con el menor trabajo humano posible en el proceso [KWB03]. En la figura 6.4, a continuación, se muestra un ejemplo del proceso de MDA.

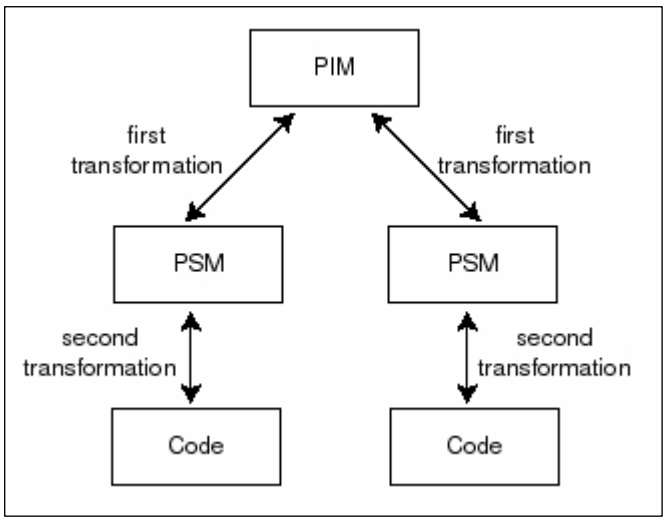


Figura 6.4 – Transformaciones en MDA, tomada de [PSWK]

MDA utiliza los conceptos de metamodelado en forma intensiva. Para más información sobre MDA, ver [OMG], [OMGMDA] y [KWB03].

Capítulo 7
Lenguajes de Dominio Específico

"Una palabra bien elegida puede economizar
no sólo cien palabras sino cien pensamientos"
Henri Poincaré

Un lenguaje de dominio específico[50] (DSL) es un lenguaje especializado para un tipo de problema concreto [Czarnecki00]. La idea básica de un DSL es un lenguaje de ordenador orientado a resolver un tipo particular de problemas [Ford06] a diferencia de un lenguaje de propósito general[51] que es creado para resolver cualquier clase de problema [Fowler04b]. Los DSLs pueden ser textuales o gráficos y pueden tener diferentes niveles de especialización. En general, son necesarios varios DSLs para especificar una aplicación completa. [Czarnecki00].

En este capítulo estudiaremos los lenguajes de dominio específico, comenzando por una definición, un conjunto de características y una comparación con los lenguajes de propósito general (GPLs). A continuación presentaremos un lenguaje de patrones para diseñar y construir DSLs. Proseguiremos nuestro estudio analizando los tipos de DSLs existentes y dando ejemplos de DSLs reales con amplia aceptación. Finalmente revisaremos sus campos principales de aplicación.

❖ ❖ ❖

[50] Domain Specific Language (DSL) en el texto original
[51] General Purpose Language (GPL) en el texto original

7.1 Introducción y Definición

Un lenguaje de dominio específico[52] (en adelante, DSL) es un lenguaje especializado para un problema particular [Czarnecki00]. La idea básica de un DSL es un lenguaje de ordenador orientado a resolver un tipo particular de problemas a diferencia de un lenguaje de propósito general[53] (en adelante, GPL) que es creado para resolver cualquier clase de problema [Fowler04b]. Los DSLs pueden ser textuales o gráficos y pueden tener diferentes niveles de especialización. En general, son necesarios varios DSLs para especificar una aplicación completa. [Czarnecki00].

7.1.1 Características de los DSL

Un lenguaje de dominio específico es un lenguaje de programación adaptado a un dominio de aplicación concreto: en lugar de ser de propósito general captura precisamente la semántica del dominio [Spinellis01]. Los DSLs son lenguajes limitados, diseñados para resolver una clase específica de problemas [Ford06]. Ofrecen una ganancia sustancial en la expresividad y facilidad de uso en su dominio de aplicación en comparación con los lenguajes de propósito general [MHS05]. Los DSLs cambian expresividad por generalidad en un dominio concreto [MHS05].

En general se necesitan varios DSLs para especificar una aplicación completa. Se pueden proveer varios DSLs diferentes diseñados para diferentes categorías de usuario objetivo para especificar cada aspecto de una aplicación[54]. [Czarnecki00]

Los DSLs pueden ser textuales (SQL, HTML, LaTeX, etc.) o gráficos (GUI Builder tipo Swing [SJFC] o WinForms [MSWF], Microsoft DSL Tools [VSTSM], etc.). Pueden tener también diferentes niveles de especialización

Los DSLs se focalizan en realizar bien una tarea concreta. Por ejemplo GraphViz ha sido diseñado para crear imágenes de grafos en una gran variedad de formas, pero no tiene capacidades para acceder a sockets o manejar cadenas.

Los DSLs también han sido llamados a lo largo del tiempo de las siguientes formas:

- Lenguajes Pequeños (*Litte Languages*)
- Lenguajes de Aplicación (*Application Languages*)
- Lenguajes de Muy Alto Nivel (*Very High Level Languages*)
- Macros

[52] Domain Specific Language (DSL) en el texto original
[53] General Purpose Language (GPL) en el texto original
[54] De hecho, ese es el enfoque que hemos utilizado para construir nuestra solución. La arquitectura general de la solución se presenta en el capítulo 11 de esta tesis y el detalle del lenguaje en el capítulo 12.

El uso de DSLs no es algo nuevo: APT (un DSL para programar máquinas de control numérico) fue desarrollado entre 1957 y 1959. BNF se remonta a 1959. [MHS05]

7.1.2 DSL versus GPL

Un DSL permite especificar software desde un punto de vista específico y acotado. Define abstracciones que codifican el vocabulario del dominio en que está focalizado. En lugar de resolver problemas generales capturan en forma precisa la semántica de un dominio [Spinellis01]. En contrapartida, un lenguaje de propósito general (GPL) no está especializado en un dominio o punto de vista.

Los DSLs están enfocados a realizar un conjunto acotado de tareas. Por ejemplo, HTML (que es un DSL) puede representar la estructura de un documento de hipertexto pero no tiene funcionalidades para escritura y lectura de ficheros. Por otro lado, C# o Java (que son GPLs) tienen primitivas muy generales que permiten resolver todo tipo de problema, pero no están especializadas en ningún dominio concreto.

Por lo tanto, hay una tensión de fuerzas entre ambos tipos de lenguajes: los DSLs tienen un campo de aplicación limitada, pero permiten resolver en forma eficiente problemas en su dominio. Por otro lado, los GPLs tienen un campo de aplicación mucho más amplio[55] (pueden resolver todo tipo de problemas), pero requieren un mayor esfuerzo para resolver problemas en un dominio que los DSLs. Con un GPL podemos realizar todas las tareas que pueden hacerse con un DSL, pero con un esfuerzo considerablemente mayor[56].

Según [Frechot03] los DSLs tienen las siguientes ventajas respectos a los GPLs[57]: son más fáciles de programar, fomentan el re-uso sistemático y la verificación es más sencilla.

7.2 Patrones para la Construcción de DSLs

En el **capítulo 3** hemos estudiado a los patrones de software y hemos visto que éstos pueden definirse en cualquier dominio. La construcción de DSLs no es ajena a esta tendencia: en [Spinellis01][58] se presenta un lenguaje de patrones (§ 3.1.4) para la creación de Lenguajes de Dominio Específico. En ese trabajo se describen ocho patrones para el diseño y construcción de DSLs, los cuales se enumeran a continuación [Spinellis01]:

[55] Utilizamos la expresión "mucho más amplio" por no decir "ilimitado", dado que gran parte de los GPLs tienen limitaciones intrínsecas.
[56] Esta afirmación no es verdadera en sentido inverso
[57] La argumentación de estas ventajas se encuentra en [Frechot03]
[58] "Notable Design Patterns for Domain-Specific Languages" de Domidis Spinellis

- PIGGYBACK: utiliza las capacidades de un lenguaje existente como anfitrión base para un nuevo DSL (patrón estructural).

- PIPELINE: resuelve un problema de composición de DSLs (patrón de comportamiento).

- LEXICAL PROCESSING: ofrece un método eficiente para diseñar e implementar DSLs (patrón de creación).

- LANGUAGE EXTENSION: se utiliza para añadir nuevas características a un lenguaje existente (patrón de creación).

- LANGUAGE SPECIALISATION: elimina características de un lenguaje base para conformar un DSL (patrón de creación).

- SOURCE TO SOURCE TRANSFORMATION: permite la implementación eficiente de traductores de DSLs (patrón de creación).

- DATA STRUCTURE REPRESENTATION: permite la especificación declarativa y específica de dominio de datos complejos (patrón de creación).

- SYSTEM FRONT-END: la configuración y adaptación de un sistema puede ser relegada frecuentemente a un frontal DSL[59] (patrón de estructural)

En la siguiente figura se muestra el mapa del lenguaje de patrones (este mapa contiene a todos los patrones del lenguaje junto con sus relaciones):

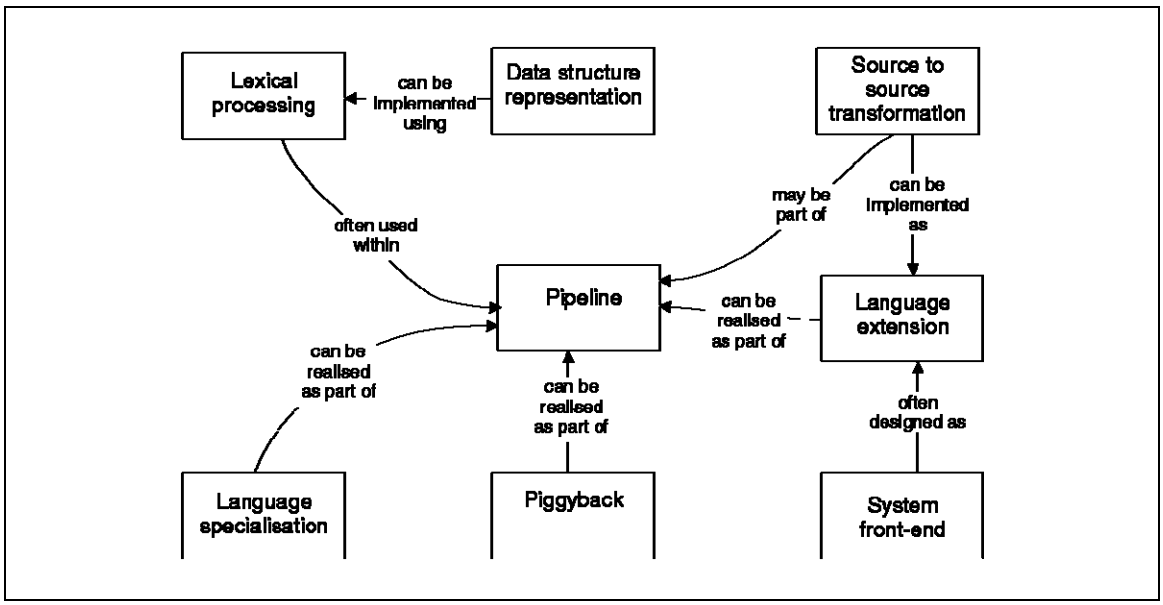


Figura 7.1 – Mapa del lenguaje de patrones para diseño y construcción de DSLs, tomada de [Spinellis01].

[59] DSL Front-End (en el texto original)

7.3 Tipos de DSLs

En [Czarnecki00] se presenta la siguiente clasificación de tipos[60] de DSLs:

- **Fixed, Separate DSLs**: estos lenguajes usualmente se implementan utilizando un procesador de lenguajes separado. Este tipo de lenguaje es el primero que viene a la mente de la gente cuando oye el acrónimo "DSL" [Czarnecki00]. Desafortunadamente, estos son también los tipos de DSLs más problemáticos. Desarrollar el traductor y el procesador para un lenguaje completo y mantenerlos es una tarea costosa. Ejemplos: LATEX y SQL.

- **DSLs embebidos**: estos DSLs están usualmente embebidos en un lenguaje de programación general (GPL). La forma más sencilla de utilizar este tipo de DSLs es mediante clases y procedimientos. Desafortunadamente este enfoque tiene varias limitaciones, como por ejemplo la inhabilidad de representar optimizaciones, reportes de error y sintaxis específicas del dominio. Ejemplos: meta-programación en C++ [Czarnecki00] y APIs [Fowler04b] [Ford06].

- **Modularly composable DSLs**: la idea de este tipo de lenguaje es ver a cada DSL como un componente. Es posible contar con componentes (DSLs) más grandes o más pequeños que pueden ser combinados en distintas configuraciones [Czarnecki00]. Hay dos sub-tipos DSLs modulares [Czarnecki00]:

    - **Encapsulated DSLs:** no influencian la semántica de los otros DSLs en la composición. Por lo tanto, son más fáciles de combinar. Ejemplo: SQL Embebido.

    - **Aspectual DSLs (sincronizacion):** su especificación influencia la semántica de las otras especificaciones (DSLs) que participan en la composición. Su implementación debe poder interactuar en forma compleja con el resto de los DSLs con los que se combina. Ejemplos: lenguajes para sincronización o gestión de errores, AspectJ, etc.

- **Pre-procesadores:** los pre-procesadores son populares mecanismos de extensión de lenguajes existentes. Usualmente expanden macros embebidas en el lenguaje de programación de destino, al cual se conoce como "lenguaje anfitrión"[61]. La ventaja de los pre-procesadores es que no deben comprender completamente el lenguaje anfitrión y por lo tanto el coste de su desarrollo puede ser menor que el de desarrollar un compilador para un lenguaje completo. Desafortunadamente, esto también es una desventaja: los errores en el destino se reportan por el compilador en términos de la salida del pre-

[60] En este caso hemos mantenido los nombres originales en inglés en algunos casos, dado que facilitan la lectura y comprensión de los conceptos.
[61] Host Language (en inglés)

procesador y no en términos de la entrada que se ha dado al pre-procesador. Otro problema es la depuración. Por lo tanto, usualmente este tipo de DSL no brinda un soporte adecuado a los programadores. [Czarnecki00]

- **Lenguajes con soporte para meta-programación:** algunos lenguajes incluyen mecanismos de extensión del lenguaje. Los templates de C++ permiten implementar optimizaciones de dominio específico manteniendo la sintaxis y semántica de C++. Los lenguajes reflectivos como Smalltalk y CLOS[62] permiten implementar cualquier clase de extensión debido a que su definición es accesible al programador en forma de librería extensible y modificable. El problema con este tipo de DSLs es que pueden hacer que el código de la aplicación sea difícil de comprender y por lo tanto mantener. Una forma de evitar este problema es separando la programación de la meta-programación. Otro problema es la complejidad de la depuración. [Czarnecki00]. Ejemplos: Templates de C++, atributos del CLR de .NET.

En [Fowler04b] [Fowler05] y [Ford06] se realiza una clasificación más sencilla de los DSLs, donde se los divide en las siguientes categorías:

- **Internos:** son lenguajes escritos sobre otro existente, utilizando su sintaxis y semántica. Por ejemplo, si Java es nuestro lenguaje, un DSL interno escrito en Java utiliza la sintaxis de Java para definir el lenguaje [Ford06] [Fowler04b]. Gran parte de los DSLs que existen en la actualidad son internos, como por ejemplo, Ruby on Rails y Rake[63].

- **Externos:** están escritos en un lenguaje separado del principal de la aplicación. Los "*lenguajes pequeños*" de Unix y los ficheros de configuración XML son buenos ejemplos de este estilo [Fowler05]. Para crear un nuevo lenguaje son necesarios tres elementos: una gramática, un analizador léxico y un analizador sintáctico [Ford06]. La principal desventaja de este tipo de lenguajes es la necesidad de construir y mantener la infraestructura necesaria para dar soporte al lenguaje. En contrapartida su ventaja principal es la posibilidad de definir un lenguaje muy alineado con el dominio.

Los dos esquemas de clasificación son complementarios y pueden utilizarse conjuntamente.

7.3.1 Ventajas de los Modularly Composable DSLs

Los *modularly composable* DSLs tienen varias ventajas sobre los DSLs monolíticos, incluyendo: [Czarnecki00]

- **Reusabilidad**: es posible dividir un lenguaje monolítico en varias extensiones modulares, permitiendo utilizarlas en distintas configuraciones.

[62] Common Lisp Object System
[63] Herramienta de Make para Ruby

- **Escalabilidad**: es posible desarrollar un sistema pequeño utilizando una pequeña configuración de los componentes de lenguaje necesarios. Cuando el sistema crece y necesita añadir nuevas funcionalidades, podemos añadir a la configuración inicial nuevas extensiones modulares para atender a los nuevos requisitos. Las extensiones modulares de lenguaje[64] evitan los problemas bien conocidos de los DLSs grandes y monolíticos que principalmente son difíciles de evolucionar.

- **Movimiento Rápido de Características[65]**: las extensiones modulares de lenguaje son una vía más rápida para distribuir nuevas características de lenguajes que los compiladores tradicionales. Las características deben "sobrevivir a partir de sus méritos" porque pueden ser cargadas y descargadas en cualquier momento. Este es no es el caso de las características de los lenguajes cerrados, donde no es posible deshacerse de una opción cuestionable una vez que ésta es parte del lenguaje.

7.4 Ejemplos de DSLs

Aun sin darnos cuenta, hemos estado utilizando DSLs durante años [Ford06]. El uso de DSLs no es algo nuevo: APT (un DSL para programar máquinas de control numérico) fue desarrollado entre 1957 y 1959. BNF[66] se remonta a 1959. [MHS05]

En la tabla 7.1, a continuación, se incluyen algunos DSLs exitosos. En cada caso se incluye su nombre y dominio de aplicación.

Lenguaje	Dominio
BNF	Especificación de sintaxis
Excel	Hojas de cálculo
HTML	Páginas Web de hipertexto
LaTeX	Edición
Make	Procesos de "build" de software
MATLAB	Computación técnica
SQL	Consultas a bases de datos
VHDL	Diseño de Hardware

Tabla 7.1 – Ejemplos de DSLs, tomada de [MHS05]

Algunos consideran a COBOL un DSL para aplicaciones de negocio. Sin embargo, otros opinan que considerar a COBOL un DSL es llevar el concepto de dominio demasiado lejos. [MHS05]

[64] Modular Language Extensions (en inglés)
[65] Fast Feature Turnover (en inglés)
[66] BNF es la sigla de Backus-Naur Form

7.5 Campos de Aplicación de los DSLs

Los DSLs pueden ser aplicados a una amplia variedad de campos en la ingeniería del software. A continuación presentaremos algunos de los más relevantes:

- **Generación de Código**: desde inicio de los noventa en adelante, la generación de código a partir de DSLs ha sido considerada por muchos una panacea para resolver la famosa *crisis del software*. La generación a partir de DSLs tiene el potencial necesario para reducir el tiempo de desarrollo de las aplicaciones, hacer el código menos complejo y reducir la curva de aprendizaje para desarrollar software. [Wijngaarden03]

- **Software Factories**: la iniciativa de *Software Factories* (cuyo principal impulsor es Microsoft) se basa en construir familias de aplicaciones de software a partir de DSLs, patrones, marcos de trabajo[67] y guías[68] [GS04]. Las *Software Factories* utilizan colecciones de DSLs hechos a medida para proveer un conjunto de abstracciones personalizadas que satisfacen las necesidades de familias específicas de aplicaciones, como por ejemplo e-commerce, banca, finanzas, etc. En este enfoque los modelos no sólo se utilizan para análisis y diseño, sino que para dar también soporte a varios tipos de computación a lo largo de todo el ciclo de desarrollo [VSTSM]. Adicionalmente, Microsoft ha construido un conjunto de herramientas de desarrollo para dar soporte a este proceso ([VSTSM], [Microsoft], [GS04])

- **Language Workbenches y Language Oriented Programming**: *Language Oriented Programming* se refiere a un estilo de desarrollo que se basa en la idea de construir aplicaciones alrededor de un conjunto de DLSs [Ford06] [Fowler05]. *Language Workbench* es un término genérico para las herramientas que dan soporte a este estilo [Fowler05]. Por lo tanto, *un language workbench es una forma de hacer language oriented programming* [Fowler05].

- **Model Driven Architecture (MDA)**: MDA es un marco de trabajo para el desarrollo de software dirigido por modelos creado por el OMG [OMG]. En MDA el proceso de software es guiado por el modelado. El papel de los DSLs en MDA es similar al que tiene en las *Software Factories*, pero con un matiz muy importante: los DSLs deben crearse a partir de estándares del OMG (por lo tanto, en este caso, deben crearse a partir de MOF [OMG]).

[67] Frameworks (en inglés)
[68] Guidance (en inglés).

Capítulo 8
Modelos de Objetos Adaptativos

"Los objetos son lo permanente, lo subsistente;
la configuración es lo cambiante, lo inestable"
Ludwig Wittgenstein, Tractatus lógico-philosophicus

Un modelo de objetos adaptativo[69] (AOM) es un sistema que representa clases, atributos, relaciones y comportamiento como metadatos [YBJ01]. El sistema es un modelo basado en instancias en lugar de clases. Los usuarios cambian los metadatos (modelo de objetos) para reflejar cambios en el dominio. Estos cambios modifican el comportamiento del sistema. En otras palabras, el sistema almacena su modelo de objetos en una base de datos y lo interpreta en tiempo de ejecución [YJ02]. Esto brinda al modelo una gran adaptabilidad[70] y flexibilidad.

En este capítulo estudiaremos los modelos de objetos adaptativos, comenzando por su definición, un conjunto de características y las consecuencias de su aplicación (tanto positivas como negativas). A continuación analizaremos la arquitectura general de los AOMs. Proseguiremos enumerando los patrones que participan en el AOM. Finalmente mencionaremos ejemplos de implementaciones reales de AOMs en sistemas comerciales, demostrando que esta tecnología puede ser aplicada en forma exitosa en diversos dominios.

❖ ❖ ❖

[69] Adaptive Object Model (AOM) en el texto original
[70] En [YJ03] se compara a los AOM con máquinas virtuales UML (UML Virtual Machines)

8.1 Introducción y Definición

Los usuarios a menudo quieren cambiar las reglas de su negocio sin necesidad de rescribir el código. Los clientes requieren que los sistemas se adapten más fácilmente a las cambiantes necesidades de su negocio [Izquierdo02].

Muchos sistemas de información de hoy en día requieren ser dinámicos y configurables de tal manera que puedan adaptarse. Esto usualmente se hace moviendo ciertos aspectos del sistema, tales como las reglas de negocio, estructura y relaciones a una base de datos de tal manera que puedan ser fácilmente cambiadas. El modelo resultante permite al sistema adaptarse rápidamente a las necesidades de cambio simplemente modificando valores en la base de datos en vez de en el código fuente. Esto promueve el desarrollo de herramientas que permitan a los directivos y administradores introducir nuevos productos sin programar y cambiar sus modelos de negocio en tiempo de ejecución. Esto puede reducir el tiempo de salida al mercado de nuevas ideas de meses a semanas o días. Por tanto la posibilidad de adaptar el sistema se sitúa en las manos de aquellos que tienen el conocimiento del negocio para hacerlo de manera efectiva. [YJ02]

Las arquitecturas que pueden adaptarse dinámicamente a nuevos requisitos del usuario en tiempo de ejecución son llamadas "*meta-arquitecturas*" o "*arquitecturas reflectivas*" [YBJ01]. Los *modelos de objetos adaptativos* (en adelante AOM) son una variante de este tipo de arquitecturas [YBJ01].

8.1.1 Definición

Un modelo de objetos adaptativo[71] (AOM) es un sistema que representa clases, atributos, relaciones y comportamiento como metadatos. El sistema es un modelo basado en instancias en lugar de clases. Los usuarios cambian los metadatos (modelo de objetos) para reflejar cambios en el dominio. Estos cambios modifican el comportamiento del sistema. En otras palabras, el sistema almacena su modelo de objetos en una base de datos y lo interpreta [YJ03]. Esto brinda al modelo una gran adaptabilidad y flexibilidad. Por tanto el modelo de objetos es activo, cuando éste se modifica el sistema cambia inmediatamente.

8.1.2 Evolución Histórica

Este tipo de arquitecturas no es nuevo. Ha ido evolucionando con los años y ha sido referida en distintos trabajos de investigación. A lo largo de su evolución ha sido llamada de diferentes formas: se la ha llamado "*Type instance pattern*" [GHV95] (OOPSLA'95), "*Active-Object Model*" en [FY98], "*Dynamic Object Model*" en [Johnson98], "*User Defined Product Framework*" en [JO98] y finalmente, "*Adaptive Object Model*" en [YBJ01].

[71] Adaptive Object Model (AOM) en el texto original

8.1.3 Características de los AOM

Un sistema con un modelo de objetos adaptativo (AOM) tiene un modelo de objetos explícito que interpreta en tiempo de ejecución. Si se cambia el modelo de objetos, el sistema cambia su comportamiento. Por ejemplo, varios sistemas de workflow tienen un AOM. Los objetos tienen estados que responden a eventos al cambiar de estado. El AOM define los objetos, sus estados, los eventos y las condiciones bajos las cuales un objeto cambia de estado. Personas con privilegios especiales pueden cambiar este modelo de objetos "sin programar". ¿O están programando después de todo? Las reglas de negocio pueden almacenarse en un AOM que hace que sea fácil evolucionar el modo en que una empresa hace negocios. [AOM]

La filosofía detrás de los AOMs se resume perfectamente en la siguiente frase de Ralph Johnson [YJ03]:

> *"Si algo va variar en un modo predecible, almacenar la descripción de la variación en una base de datos para hacer que sea fácil de cambiar"*

De esta frase podemos extraer la siguiente conclusión, *si quieres que algo cambie rápido, debes moverlo a los datos* [YJ03].

Los AOMs tienen las siguientes características principales [YJ03]:

- Son arquitecturas que pueden adaptarse dinámicamente a nuevos requisitos del usuario mediante el uso de metadatos (**capítulo 6**) descriptivos sobre las reglas de negocio y la estructura, que son interpretados en tiempo de ejecución.

- Son muy flexibles (pueden ser modificados por analistas de negocio)

- Son un tipo de "*arquitectura reflectiva*" o "*meta-arquitectura*" (§ 8.1)

8.1.4 Consecuencias

En esta sección analizaremos las principales ventajas y desventajas derivadas de la utilización de modelos de objetos adaptativos (AOMs.)

8.1.4.1 Ventajas

Los AOMs tienen las siguientes ventajas principales:

- **Facilidad de cambio**: esta es la ventaja principal de los AOMs. Este tipo de arquitecturas es adecuada en el escenario de sistemas que cambian constantemente o cuando se quiere dejar a los usuarios que los configuren y

extiendan dinámicamente. Un AOM puede derivar en un sistema que permita a los usuarios "programar sin programar". [YJ02]

- **Menor Cantidad de Clases:** convertir un programa en un modelo de objetos adaptativo reduce el número de clases, haciéndolo más pequeño. La información que estaba fija en el código fuente se almacena en la base de datos. Esta nueva estructura de clases no cambia. En lugar de esto, los cambios en las especificaciones producen cambios en el contenido de la base de datos. [YJ02]

- **Desarrollo Rápido**: el caso de negocio principal para los AOMs es la creación y modificación rápida de software. Los AOMs pueden contribuir a reducir drásticamente el tiempo de mercado[72] de una aplicación [YJ02]

- **Flexibilidad y Adaptabilidad**: los AOMs pueden adaptarse en forma sencilla a cambios en los requisitos originales. Para adaptar el sistema idealmente no es necesario modificar código fuente, sino los metadatos que describen al modelo de objetos que se utilizará en tiempo de ejecución.

8.1.4.2 Desventajas de los AOM

Los AOMs tienen las siguientes desventajas principales [YJ02]:

- **Mayor esfuerzo de construcción**: los AOMs generalmente requieren soporte de herramientas especiales e interfaces gráficas de usuario (GUIs) para definir los objetos en el sistema. Requieren también un sistema para interpretar el modelo. El AOM esta empotrado en el sistema y produce efectos en su ejecución. Por lo tanto, los AOMs requieren un soporte de software adecuado.

- **Más difíciles de entender**: los AOMs pueden ser más difíciles de entender dado que hay dos sistemas de objetos coexistiendo: el intérprete escrito en un lenguaje de programación orientado a objetos y el modelo de objetos adaptativos (codificado en metadatos) que se interpreta. Las clases no representan abstracciones de negocio, debido a que la mayoría de la información sobre el negocio está en la base de datos.

- **Rendimiento**: los AOMs pueden ser más lentos debido a que usualmente se basan en interpretar la representación del modelo de objetos.

- **Mayor dificultad de mantenimiento**: los AOMs pueden ser más difíciles de mantener. Usualmente este es el caso cuando el arquitecto principal deja al equipo y la tarea de mantenimiento recae en desarrolladores que no comprenden este tipo de arquitecturas. De todas formas, los desarrolladores

[72] Time-to-market (en inglés)

que entienden este tipo de arquitecturas las encuentras más fáciles de mantener debido a que hay menos código fuente y a que típicamente un pequeño cambio en la aplicación puede producir un gran cambio en la aplicación en ejecución.

8.2 Arquitectura de un AOM

Los modelos de objetos adaptativos proporcionan una alternativa a los diseños orientados a objeto tradicionales. El diseño tradicional asigna clases a los diferentes tipos de entidades de negocio y les asocia atributos y métodos. Las clases modelan el negocio así que un cambio en el negocio ocasiona un cambio en el código que produce una nueva versión de la aplicación [Izquierdo02] [YBJ01]

Un modelo de objetos adaptativo no modelas las entidades como clases. En vez de eso modelan las descripciones de las clases que son interpretadas en tiempo de ejecución. Por tanto cuando un cambio del negocio es requerido estas descripciones son cambiadas e inmediatamente reflejadas en la aplicación en ejecución [Izquierdo02] [YBJ01].

En esta sección presentaremos los elementos principales que conforman el estilo de arquitectura de los AOMs.

8.2.1 Type Object

La mayoría de los lenguajes orientados a objetos estructuran un programa como un conjunto de clases [YJ02]. Una clase define la estructura y comportamiento de los sus instancias (objetos) [YJ02]. La mayoría de los sistemas orientados a objetos generalmente utilizan una clase diferente para cada tipo de objeto, lo cual implica que para introducir un nuevo tipo de objeto es requisito necesario crear una nueva clase, lo cual requiere programación [YJ02]. Sin embargo, si la diferencia entre los tipos de clases es lo suficientemente pequeña, los objetos pueden ser *generalizados* y las diferencias entre ellos pueden ser descritas utilizando parámetros. [YBJ01] [YJ02]

El patrón TYPE OBJECT [Woolf97] divide una clase en dos clases: una representa un *tipo* y la otra tiene una *referencia* a ese tipo (como se muestra en la figura 8.1) [YBJ01].

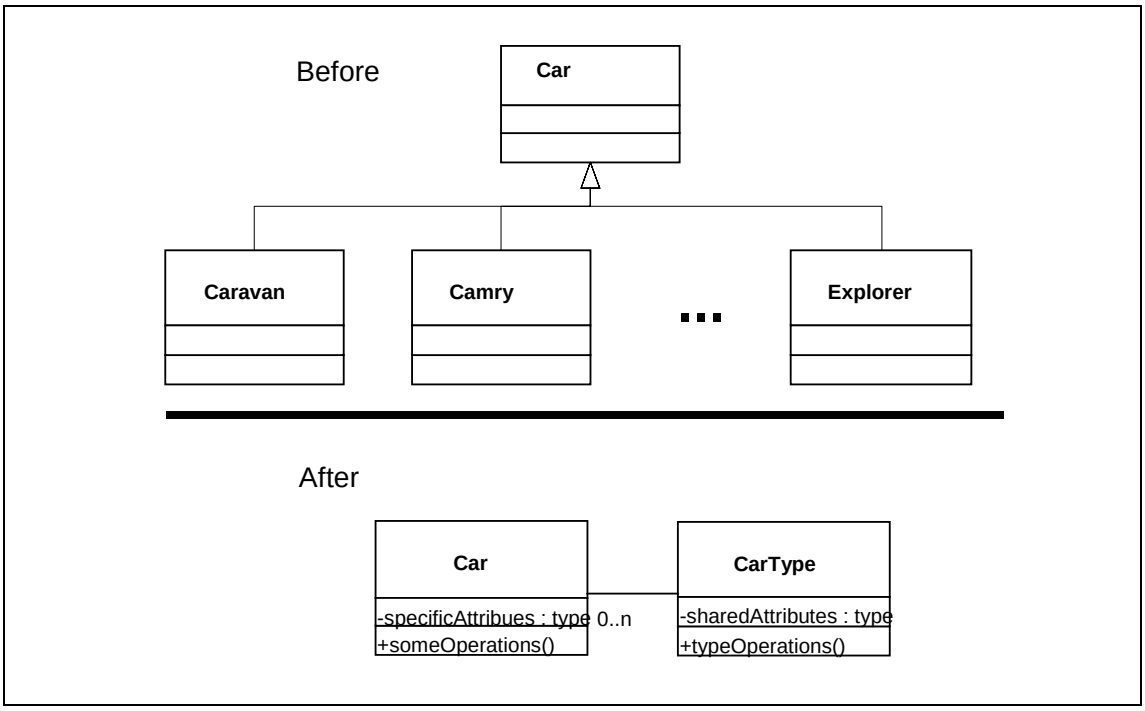


Figura 8.1 – El patrón Type Object

TYPE OBJECT hace de las subclases desconocidas simples instancias de una clase genérica. Nuevas clases pueden ser creadas dinámicamente en tiempo de ejecución instanciando dicha clase genérica. Los objetos creados de la jerarquía tradicional son creados, pero haciendo explícita la relación entre ellos y su tipo. [YJ02]

8.2.2 Property

Los atributos de un objeto usualmente se implementan en sus variables de instancia. Estas variables de definen generalmente en cada subclase. Si objetos de distintos tipos son todos creados a partir de la misma clase genérica se presenta el problema de cómo pueden tener distintos atributos. La solución es aplicar el patrón PROPERTY [FY98] el cual modela cada uno de los atributos de una entidad como un objeto independiente. Por tanto todas las entidades tienen como variable de instancia lo mismo: un contenedor de propiedades. El objeto propiedad contiene tres elementos: el nombre del atributo, su valor y su tipo. [YJ02]

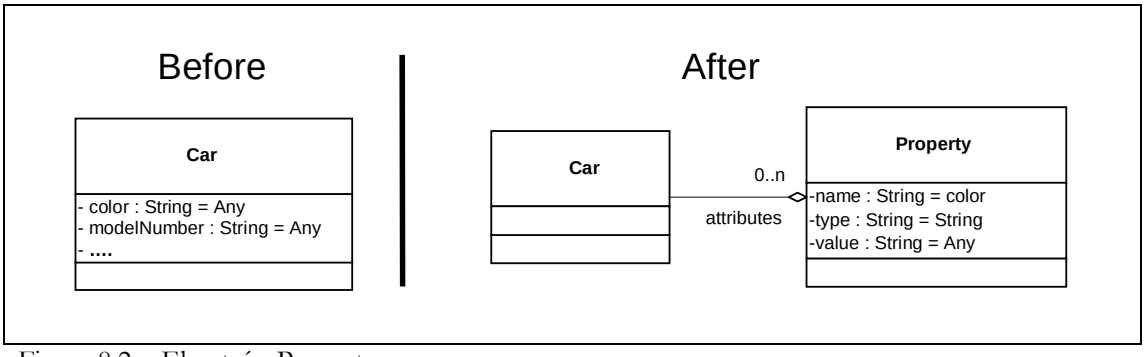


Figura 8.2 – El patrón Property

La mayor parte de las arquitecturas basadas en modelos de objetos adaptativos utilizan los patrones TYPE OBJECT y PROPERTY.

8.2.2.1 Type Square

En la mayoría de los modelos de objetos adaptativos TYPE OBJECT se utiliza dos veces: una antes de utilizar el patrón PROPERTY y otra luego de utilizarlo. TYPE OBJECT divide el sistema en entidades (`Entity`) y tipos de entidades (`EntityType`). Las entidades tienen atributos que pueden ser definidos utilizando propiedades (`Property`). Cada propiedad tiene un tipo (`PropertyType`) y cada tipo de entidad (`EntityType`) puede entonces especificar los tipos de las propiedades para sus entidades. En la figura 8.3 a continuación se muestra el resultado de combinar estos dos patrones, al cual llamamos TYPE SQUARE[73] [YBJ01].

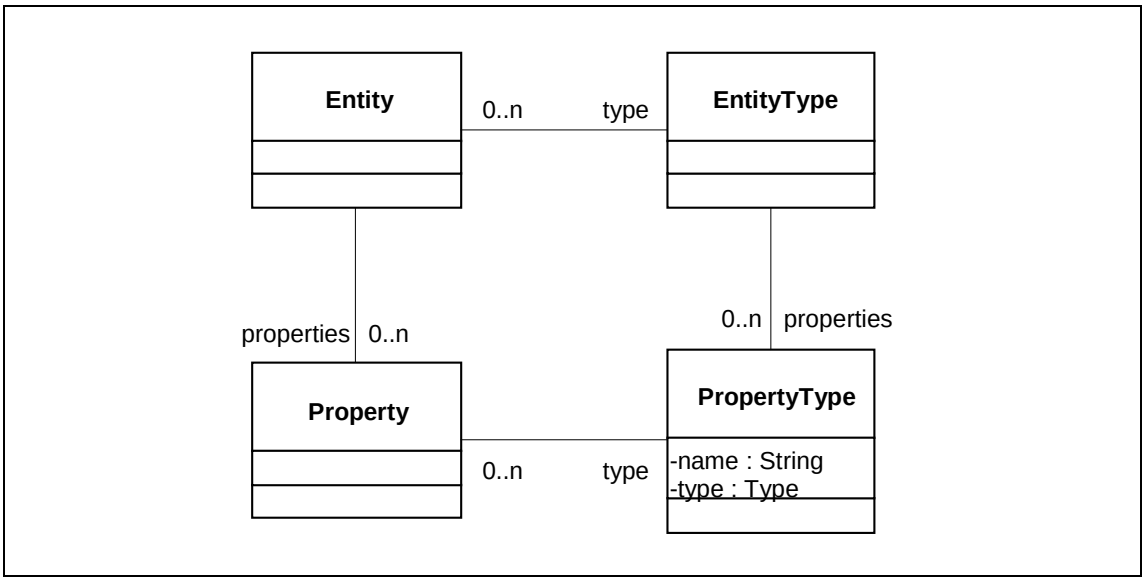


Figura 8.3 – El patrón Type Square

8.2.3 Entidades y Relaciones

Los atributos son propiedades que usualmente se refieren a tipos primitivos. Estas asociaciones son usualmente unidireccionales. Las relaciones son propiedades que se refieren a otras entidades y son generalmente bidireccionales. Esto generalmente se modela mediante dos subclases de propiedades, una para los atributos y otra para las relaciones (como se muestra en la figura 8.4).

Estas dos subclases se suelen modelar de tres formas [YJ03] [YBJ01] [Izquierdo02].

[73] Type Square es un patrón que combina a los patrones Property y Type Object

- La primera es volviendo a aplicar otra vez el patrón PROPERTY para las relaciones al igual que se aplicó para los atributos.

- La segunda manera es utilizar la misma construcción que se hizo para los atributos pero poniendo como clase padre a la clase `Property` y subdividiendo mediante herencia en `Property` y en `Association`. Una `Association` conocería su cardinalidad.

- La tercera forma sería discriminando por el valor de la propiedad. Si dicho valor es otra entidad se supone que es una relación.

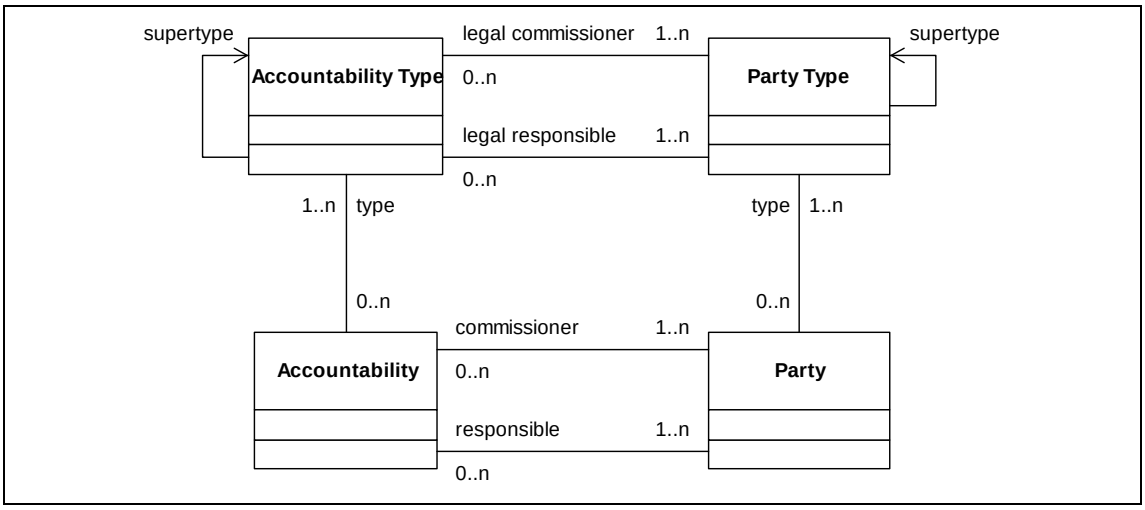


Figura 8.4 – Relaciones entre entidades la utilización del patrón Accountability

8.2.4 Strategy y RuleObjects

Una estrategia es un objeto que encapsula a un algoritmo [GoF95]. El patrón estrategia define una interfaz estándar para una familia de algoritmos para que los clientes puedan trabajar con cualquiera de ellos de la misma manera. Si el comportamiento de un objeto se define en función de una o más estrategias entonces su comportamiento puede ser modificado fácilmente. [YBJ01]

Cada aplicación del patrón STRATEGY produce una interfaz diferente y por lo tanto una jerarquía de clases *estrategia* distinta. Las estrategias pueden evolucionar para llegar a ser complicadas reglas de negocio que son construidas o interpretadas en tiempo de ejecución. Estas pueden ser reglas primitivas o combinación de reglas mediante la aplicación del patrón COMPOSITE [GoF95]. En la siguiente figura se muestra el TYPE SQUARE aumentado con reglas.

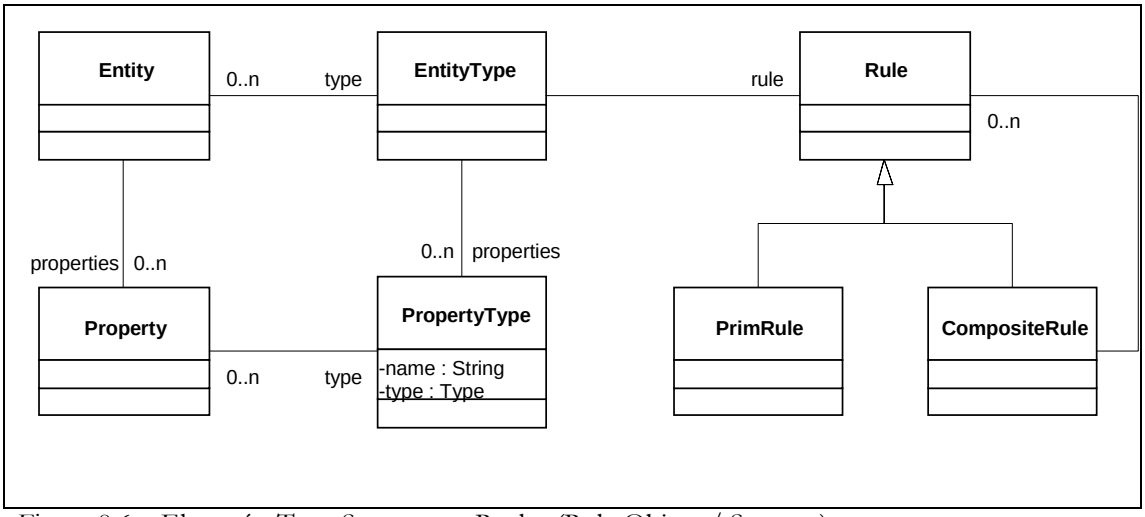


Figura 8.6 – El patrón Type Square con Reglas (Rule Object / Strategy)

El modelado de reglas complicadas junto con el interfaz de usuario dinámico usualmente es la parte más difícil de los modelos de objetos adaptativos y es la razón por la que no son implementables en un marco de trabajo genérico [YBJ01] [Izquierdo02].

8.2.5 Intérpretes de los Metadatos

Los metadatos para describir el modelo de objetos y las reglas de negocio se interpretan en dos sitios. El primero es donde los objetos son construidos, por ejemplo, cuando el modelo de objetos es instanciado. El segundo es durante la interpretación de las reglas en tiempo de ejecución. [YJ02]

La información sobre los tipos de entidades, propiedades, relaciones y comportamientos está almacenada en la base de datos. A pesar de cómo se guardan los datos, debe ser interpretado para construir la instancia del modelo de objetos adaptativo que representa el modelo real de negocio. Si se utiliza una base de datos orientada a objetos, los tipos de objetos pueden ser construidos simplemente instanciando a los `TypeObjects`, `Properties` y `RuleObjects`. De otra manera los metadatos se leen de la base de datos para instanciar estos objetos, los cuales se construyen utilizando en forma conjunta a los patrones BUILDER [GoF95] e INTERPRETER [GoF95] [YJ02].

8.2.6 Interfaz de Usuario para Introducir Tipos

Una de las razones principales para diseñar modelos de objetos adaptativos es permitir a los usuarios y expertos de dominio cambiar el comportamiento del sistema definiendo nuevas entidades, relaciones y reglas. En ocasiones el objetivo es permitir a los usuarios extender el sistema sin programadores. Pero inclusive cuando sólo los desarrolladores definan nuevas entidades y reglas es habitual construir un interfaz de usuario especializado para definirlos.

Los tipos son guardados en una base de datos centralizada. Esto significa que cuando alguien define nuevos tipos las aplicaciones pueden usarlos sin necesidad de ser recompiladas. A menudo las aplicaciones son capaces de usar los nuevos tipos inmediatamente mientras otras veces guardan en una caché la información de tipos y deben ser actualizadas antes de que usen los nuevos.

8.3 Patrones en el AOM

Esta arquitectura es "densa en patrones"[74] [Alexander79] y por tanto puede describirse y documentarse mediante un conjunto de patrones pertenecientes a distintos lenguajes y sistemas de patrones ([FY98], [Fowler96], [Woolf97], [GoF95], [YBJ01], [Arsanjani01]). Podríamos decir que este estilo de arquitectura "integra" a todos estos patrones en un modelo flexible y extensible. A continuación se enumeran los patrones utilizados en el AOM:

- TYPE OBJECT [Woolf97]: para representar los tipos dinámicamente dentro del modelo de objetos adaptativos. Las instancias se vinculan con sus tipos mediante composición (una instancia tiene asociada otra instancia de un `type object`)

- PROPERTY [FY98]: para representar las propiedades asociadas a una entidad. Estas propiedades se asocian a la entidad mediante composición, permitiendo que la estructura de la instancia varíe en tiempo de ejecución dinámicamente.

- TYPE SQUARE [YBJ01]: modelo canónico para los AOMs. A través de este patrón se pueden representar modelos de objetos que varían en función de metadatos (combina TYPE OBJECT, PROPERTY y ACCOUNTABILITY).

- RULE OBJECT [Arsanjani01]: se utiliza para asociar reglas de negocio al TYPE SQUARE (se utiliza junto con STRATEGY).

- STRATEGY [GoF95]: permite hacer variar esas reglas de negocio en tiempo de ejecución (se utiliza conjuntamente con RULE OBJECT).

- INTERPRETER [GoF95]: lo utiliza el analizador para construir la instancia del modelo de objetos a partir de los metadatos (se utiliza conjuntamente con el patrón BUILDER)

- BUILDER [GoF95]: lo utiliza el analizador para construir la instancia del modelo de objetos a partir de los metadatos (se utiliza conjuntamente con el INTERPRETER)

[74] Pattern-dense (en inglés)

- SCHEMA [FY98]: se utiliza para especificar el esquema (estructura y reglas) que tendrá las instancias del modelo de objetos en tiempo de ejecución.

- ACCOUNTABILITY [Fowler96]: se utiliza para modelar las relaciones entre entidades (junto con el TYPE SQUARE).

- COMPOSITE [GoF95]: se utiliza para diseñar propiedades y estrategias que puedan contener a otras mediante composición recursiva.

Una interesante observación que puede hacerse tras leer esta lista es que la flexibilidad del modelo se debe a la composición dinámica de elementos (todos los patrones que se componen en este estilo de arquitectura utilizan en un alto grado la composición).

8.4 Casos de Implementación de AOMs

A continuación, enumeraremos brevemente cuatro proyectos existentes donde se han utilizado AOMs: [YJ02]

- **User-Defined Product Framework (UDP)**: aplicación desarrollada para la compañía *The Hartford* [Hartford]. El AOM ha sido utilizado en este caso para representar pólizas de seguros. UDP es un marco de trabajo para crear objetos compuestos basados en atributos. El objetivo es poder crear o modificar pólizas de seguro en forma ágil, sin necesidad de programar. Estas modificaciones pueden ser realizadas por expertos del negocio.

- **Argo Document Workflow Framework**: desarrollado para dar soporte a la administración de Argo en Bélgica. Argo es una organización semi-gubernamental que gestiona varios cientos de escuelas públicas. Utiliza este marco de trabajo para desarrollar sus aplicaciones, las cuales comparten un modelo de negocio común y requieren bases de datos, documentos electrónicos, flujos de trabajo y funcionalidades de Internet.

- **Objectiva's Telephony Billing System**: este sistema permite a los desarrolladores construir aplicaciones reutilizando clases existentes y no los fuerza a crear nuevas. *Objectiva* permite crear sistemas de facturación para todo tipo de servicios de telecomunicaciones, incluyendo telefonía móvil, PCS, portabilidad de números locales, llamadas locales y de larga distancia y sistemas satelitales. Permite también personalizar rápidamente un sistema existente para responder a condiciones cambiantes y para proveer nuevos servicios. Es un sistema de facturación que permite que un único sistema gestione cualquier servicio de telecomunicaciones.

- **IDPH Medical Domain Framework**: aplicación desarrollada para el Departamento de Salud Pública de Illinois para gestionar información médica de pacientes.

En todos los casos se utiliza una variante de TYPE SQUARE, aunque luego cada uno tiene sus particularidades. Todas estas implementaciones son analizadas en detalle en [YJ02].

Capítulo 9
Sistemas Emergentes

"El creciente interés por la descentralización significa
mucho más que nuevos tipos de organizaciones.
Significa una nueva forma de ver el mundo,
nuevos modos de pensar y nuevos modos de conocer."
Mitchel Resnick, Turtles, Termites and Traffic Jams (1994)

En los sistemas emergentes, la combinación comportamientos sencillos a nivel local produce comportamientos complejos a nivel global. No existe una componente central, una "mano invisible"[75] que regule el comportamiento global del sistema, no hay un coordinador, no hay un líder.

En este capítulo analizaremos a los sistemas emergentes comenzando por el estudio de la emergencia incluyendo conceptos como la concepción centralizadora, las reglas de la emergencia y un conjunto de heurísticas para el pensamiento descentralizador. Proseguiremos con un conjunto de ejemplos de sistemas emergentes. Finalmente aplicaremos los conceptos presentados a la disciplina de la ingeniería del software.

❖ ❖ ❖

[75] La mano invisible es un concepto introducido por el economista Adam Smith (1723-1790) en la obra "La riqueza de las naciones", publicada en 1776. Según esa obra, la mano invisible guía el mercado hacia la eficiencia.

9.1 Emergencia

Emergencia es lo que ocurre cuando un sistema de elementos relativamente simples se organiza espontáneamente y sin leyes explícitas hasta dar lugar a un comportamiento inteligente. Los agentes de un nivel inferior adoptan comportamientos propios de un nivel superior: las hormigas crean colonias; los urbanistas, vecindarios [Johnson01]. Es lo que sucede cuando "*el todo es mas inteligente que la suma de sus partes*" [OR02].

Los sistemas emergentes tienen las siguientes características:

> 1. No hay un control jerárquico de arriba hacia abajo (top-down) que diga al sistema que es lo que debe hacer (por ejemplo, "armar una pila de madera").
>
> 2. Cada una de las entidades involucradas es en sí bastante tonta. Sigue unas pocas reglas sencillas y locales que sólo ella conoce.
>
> 3. La interacción de reglas locales simples y azar producen un diseño emergente global que no es inherente a las partes.

A continuación, enumero algunos ejemplos de sistemas emergentes muy populares:

- Colonias de hormigas
- Atascos de tráfico
- Ciudades
- El Cerebro humano
- La economía
- La Web

Mitchell Resnick del MIT[76] ha desarrollado un software de modelado masivamente paralelo llamado StarLogo [Starlogo] que permite hacer simulaciones de este tipo de sistemas para explorar comportamientos descentralizados. En [Resnick94] se detallan una serie de experimentos[77] realizados con StarLogo[78], por ejemplo, colonias de hormigas, atascos de tráfico, incendios forestales, construcción de termiteros, etc.

9.1.1 La Concepción Centralizadora

[76] Massachusetts Institute of Technology

[77] Estos ejemplos se incluyen con su código fuente en StarLogo y corresponden a experimentos pedagógicos que realizo Resnick con alumnos de colegios secundarios, siguiendo un enfoque construccionista "a la Piaget". Uno de sus objetivos fue "observar como lo estudiantes pensaban sobre los sistemas descentralizados" [Resnick94]

[78] StarLogo continúa evolucionando y la última versión es la 2.22, liberada el 8 de Junio de 2006.

Los seres humanos tienen una fuerte tendencia intrínseca a buscar un elemento coordinador detrás de todos los sistemas y cuando observan un patrón existente en el mundo, con frecuencia suponen que existe algún tipo de control centralizado [Resnick94]. Esto se conoce como la concepción centralizadora: la necesidad de encontrar un ente controlador en todo conjunto de elementos que interactúan para obtener un fin.

Pese a esto, en los últimos años los conceptos de emergencia y autoorganización se han hecho más populares y actualmente existen múltiples grupos de investigación activos trabajando en estos temas.

9.1.2 Reglas de la Emergencia

En el libro Sistemas Emergentes [Johnson01], se enumeran los principios fundamentales de la emergencia que son las condiciones necesarias para que se produzcan fenómenos emergentes. Siguiendo estos principios, la inteligencia y adaptabilidad del sistema total derivan del conocimiento local de sus elementos.

- **Más es diferente:** diez hormigas deambulando en el suelo desierto no podrán juzgar adecuadamente las necesidades de roles según su entorno, pero dos mil si. Se aplica también a la distinción entre micromotivos y macroconducta: cada hormiga individualmente "no sabe" que esta estableciendo prioridades entre caminos posibles al dejar un gradiente de feromona.

- **La ignorancia es útil:** cuanto más sencillos sean los agentes que conforman el sistema, mejor. Es mejor construir un sistema densamente interconectado con elementos simples y dejar que la conduzca más sofisticada surja paulatinamente.

- **Alentar los encuentros casuales:** estos sistemas dependen fuertemente de las interacciones entre sus agentes que exploran un espacio sin órdenes definidas. Sin encuentros casuales, una colonia de hormigas sería incapaz de encontrar nuevas fuentes de alimentos.

- **Buscar patrones en los signos:** las hormigas, por ejemplo, dependen de los patrones semioquímicos que detectan. Estos patrones se generan a partir de los gradientes de feromonas que excretan. La feromona excretada por una hormiga recolectora no dice nada, pero la de cincuenta hormigas sí.

- **Prestar atención a tus vecinos:** según [Johnson01], es la lección más importante que nos dejan las hormigas y la de más vastas consecuencias. Podría reformularse como "la información local conduce a la sabiduría global". Si las hormigas no se encontraran unas con otras, la colonia sería un ensamblaje sin sentido de organismos individuales.

Para una explicación mas detallada de estos principios, ver el Capítulo II de la segunda parte del libro "*Sistemas Emergentes*" [Johnson01].

9.1.3 Heurísticas Orientadoras para el Pensamiento Descentralizador

En capítulo "Reflexiones" de [Resnick94], se plantean una serie de heurísticas orientadoras para ayudar a pensar sobre los mundos descentralizados. Las cinco heurísticas analizadas son las siguientes:

1. **La realimentación positiva no es siempre negativa**: la realimentación positiva con frecuencia desempeña un papel importante en la creación y extensión de patrones y estructuras.

2. **La aleatoriedad puede ayudar a crear orden**: la mayoría de las personas consideran que la aleatoriedad es destructiva, pero en algunos casos ayuda a construir sistemas mejor organizados.

3. **Una bandada no es un pájaro grande**: es importante no confundir niveles. Con frecuencia, las personas confunden las conductas de los individuos con las de los grupos.

4. **Un atasco de tráfico no es sólo una colección de automóviles**: es importante advertir que algunos objetos ("objetos emergentes") tienen una composición siempre cambiante.

5. **Los montes están vivos**: las personas se concentran frecuentemente en las conductas de los objetos individuales y pasan por alto el ambiente que rodea a los objetos.

Para comprender mejor las heurísticas es recomendable leer "*Tortugas, Termitas y Atascos de Tráfico*" [Resnick94], ya que de alguna manera son conclusiones tomadas de los experimentos realizados y detallados en esa obra.

9.2 Ejemplos de Sistemas Emergentes

Los sistemas emergentes tienen aplicaciones en una amplia variedad de campos. A continuación enumeraremos algunos de ellos:

- Bioinformática
- Análisis de problemas complejos (por ejemplo, el problema del viajante)
- Desarrollo de software
- Planificación Urbanística
- Mundo empresarial
- Modelos sociales

- Modelos de aprendizaje
- Organizaciones sociales

9.2.1 Los Sistemas Emergentes en las Organizaciones

Desde el punto de vista de las organizaciones, hay muchas pequeñas normas que pueden tener efectos tremendos. En "*Hacia la Empresa en Red*" [Cornella03], se plantea una aplicación de sistemas emergentes al mundo empresarial, planteando un conjunto de doce reglas simples que al ser seguidas por los integrantes de una organización pueden tener un impacto altamente positivo, a saber [Cornella03]:

- Chequeo personal de conocimiento, actual y necesidades
- Entrenador personal
- Un tiempo para aprender, cada día
- Dónde generamos valor para el cliente
- Premiar a quién pregunta y a quién responde
- Premiar a los grupos por sus éxitos
- Organizar lo que ya tenemos (ontologías)
- Todo con su síntesis
- Compartir lo que encuentras
- Enseñar lo que sabes
- Orientación a objetos, reutilizables
- Traductores entre comunidades focalizadas

Si se busca en esta lista, no encontrará ninguna regla que diga directamente "ser más eficiente", "trabajar más", "dar más ganancias a la empresa" o "trabajar mejor" ni nada por el estilo, pero la aplicación de estas reglas hará posible su emergencia.

9.2.2 ¿Cómo Consiguen las Hormigas sus Alimentos?

Las comunidades de hormigas son un excelente ejemplo de sistemas emergentes autoorganizados. Para conseguir alimento en forma eficiente, siguen un conjunto de reglas muy sencillas, a saber:

- Salir del hormiguero en búsqueda de alimento
    - Si hay un rastro de feromona, seguirlo
    - Si hay mas de un rastro de feromona, seguir al más fuerte
    - Si no hay rastro de feromona, continuar con la búsqueda
- Si encuentra alimento, llevarlo nuevamente al hormiguero
- Durante el viaje de retorno al hormiguero, excretar feromonas

Cuando una hormiga sale del hormiguero, si hay un rastro de feromona, lo sigue, sino, busca comida. Al encontrarla, retorna al hormiguero, dejando un rastro de feromona. Aquí se produce un fenómeno de retroalimentación positiva: cuantas más

hormigas sigan un rastro, más fuerte se hace, dado que todas excretan feromonas al volver sobre él para retornar al hormiguero. Cuando se agota la fuente de alimento, este rastro se evapora al cabo de un tiempo, dado que las hormigas sólo excretan las feromonas cuando llevan alimento.

Como podemos apreciar, ninguna conoce el objetivo general y está altamente especializada en un conjunto de tareas sencillas. A partir de la interacción de los comportamientos de agentes que siguen estas reglas surge un comportamiento extremadamente complejo: conseguir alimentos en forma eficiente.

Adicionalmente, siguiendo reglas sencillas, las hormigas también se organizan para construir y mantener cementerios y basurales en sus colonias.

9.2.3 Otro Ejemplo: Las Ciudades

Las ciudades, al igual que las colonias de hormigas, poseen un tipo de inteligencia emergente: la habilidad de almacenar y recuperar información, de reconocer y responder a patrones en el comportamiento humano. Los habitantes de la ciudad contribuimos a esa inteligencia emergente, pero no es casi imposible de percibirlo, porque nuestras vidas se desarrollan a otra escala temporal. [Johnson01]

9.3 La Emergencia en la Ingeniería del Software

A modo de ejemplo, en [C2ME] se presenta el siguiente ejemplo: en Extreme Programing, el buen diseño es el resultado emergente de la aplicación de reglas simples de calidad de código, las cuales se enumeran en el libro "*Extreme Programming Explained: Embrace the Change"* [Beck00]:

1. Que todas las pruebas se ejecuten correctamente
2. Que el código exprese las ideas que uno desea expresar
3. Que no exista código duplicado
4. Contener el mínimo número de clases y métodos.

Para los practicantes de XP, a partir de aplicar estas sencillas reglas se puede obtener una aplicación de calidad. Es decir, la calidad y el buen diseño (comportamiento a nivel macro) "emergen" a partir de la aplicación de un grupo de reglas sencillas (comportamiento a nivel micro).

9.3.1 Revisión de Conceptos Informáticos a Través de la Visión Emergente

La emergencia nos ofrece una visión alternativa a conceptos clásicos de la Ingeniería del Software y de la Teoría de General de los Sistemas[79] [Bertalanffy68]. A modo de ejemplo, podemos citar el análisis de la *Falacia Operacional* desde el punto de vista emergente realizado en el breve ensayo "*La falacia operacional y los sistemas emergentes*"[80] [Welicki06d].

La falacia operacional es un axioma introducido por John Gall en su obra "*Systemantics*"[Gall75], y se refiere a que "*La gente en los sistemas no hace lo que el sistema dice que hacen"*. Desde el punto de vista de la emergencia, esto no es algo negativo o problemático, dado que en los sistemas emergentes ninguno de sus componentes conoce el objetivo completo del sistema (en *Systemantics* se analiza por qué los sistemas no funcionan). Esta situación de excepción de la falacia operacional parece ser muy común en los sistemas emergentes, donde existen elementos con comportamientos sencillos que producen comportamientos complejos a partir de sus interacciones locales. Por lo tanto, ningún componente del sistema hace lo que el sistema dice hacer.

A primera vista, la falacia operacional parece un concepto universal y cuando se combina con los demás teoremas y axiomas presentados en *Systemantics* tiene una cierta connotación negativa. Pero como hemos visto en el apartado anterior esto no es siempre verdadero: en los sistemas emergentes, la gente no hace lo que el sistema dice que hace.

9.3.1.1 Cohesión y Acoplamiento

La cohesión y el acoplamiento son dos conceptos básicos y fundamentales de la Ingeniería del Software. Su correcta aplicación permite un alto grado de independencia funcional entre los elementos que conforman un sistema.

La cohesión es una medida de la fuerza relativa funcional de un módulo [Pressman02] y se refiere al grado de especialización de una función (método) para realizar una tarea. Un elemento con alta cohesión realiza una sola tarea. Se busca maximizar el nivel de cohesión.

El acoplamiento es la medida de la interconexión entre los módulos dentro de una estructura de software [Pressman02]. Cuando menor es el acoplamiento, los módulos son más independientes y es más sencillo reemplazarlos por otros. Se intenta minimizar el acoplamiento.

[79] De hecho, nos ofrece una visión de distinta de cualquier tema.
[80] Este es un trabajo disponible desde la Web, pero que no ha sido publicado formalmente.

En los sistemas emergentes sus elementos tienen alta cohesión y bajo acoplamiento, dado que están altamente especializados en la realización de una tarea y sólo se comunican a través de interacciones locales sencillas bien definidas.

Capítulo 10
Enfoques Existentes

"En teoría, no existe diferencia
entre teoría y práctica;
en la práctica sí la hay."
Jan L.A. van de Snepscheut

La descripción de patrones, su catalogación y categorización para su posterior compartición es un problema complejo que ha generado un gran interés tanto en la industria como en ámbitos académicos. Existen distintas iniciativas que atacan al problema en formas diversas, con el fin de resolverlo total o parcialmente. Nuestro estudio de los enfoques existentes se centra en tres ámbitos:

- **Modelos de descripción y catalogación**: actualmente existen varios lenguajes para describir a los patrones con diferentes grados de especialización para esta tarea. Complementariamente a los lenguajes, hay modelos que permiten clasificar y catalogar a los patrones, a efectos de facilitar su recuperación y posterior utilización.

- **Catálogos públicos existentes en la Web**: en los últimos años se han publicado en la Web varios catálogos públicos de patrones. En este capítulo analizaremos varios de los catálogos más populares.

- **Herramientas de modelado**: casi todas las herramientas de modelado en la actualidad incluyen funcionalidades para utilizar patrones en el modelado que permiten navegar por un catálogo de patrones y generar artefactos de software[81].

En este capítulo analizaremos diversos modelos, catálogos y herramientas para demostrar finalmente que ninguno de ellos es suficiente para resolver el problema planteado al inicio de esta tesis (**capítulo 1**). Finalmente a modo de conclusión enumeraremos los problemas recurrentes encontrados en todos los enfoques estudiados.

❖ ❖ ❖

[81] Estos artefactos no son necesariamente ejecutables. En el caso de las herramientas de modelado, pueden ser diagramas a partir de los cuales puede generarse código fuente a posteriori.

10.1 Descripción y Catalogación

Existen varios enfoques para describir y/o catalogar patrones. En esta sección describiremos los más significativos y utilizados y en cada caso expondremos sus limitaciones.

10.1.1 Wiki

Según Wikipedia[82] [Wikipedia], un wiki (del hawaiano wiki wiki, «*rápido*») es:

> *Una colección de páginas hipertexto, que pueden ser visitadas y editadas por cualquier persona (aunque en algunos casos se exige el registro como usuario) en cualquier momento. Una versión Web de un wiki también se llama WikiWikiWeb. Se trata de un simple juego de palabras, ya que las iniciales son «WWW» como las de la World Wide Web.*

Un wiki es una aplicación de informática colaborativa en un servidor que permite que los documentos allí alojados (las *páginas wiki*) sean escritos de forma colaborativa a través de un navegador, utilizando una notación sencilla para dar formato, crear enlaces, etc. Cuando alguien edita una página wiki, sus cambios aparecen inmediatamente en la Web, sin pasar por ningún tipo de revisión previa. [Wikipedia]

La principal utilidad de un wiki es que permite crear y mejorar las páginas de forma instantánea dando una gran libertad al usuario y por medio de una interfaz muy simple. Esto hace que más gente participe en su edición a diferencia de los sistemas tradicionales, donde resulta más difícil que los usuarios del sitio contribuyan a mejorarlo [Wikipedia] [PPR]. Los principios de diseño en los que se fundan los wikis se presentan en [C2WDP].

10.1.1.1 Los Wikis y los Patrones

Los orígenes de los wikis están en la comunidad de patrones de diseño, que los utilizaron para escribir y discutir patrones. El primer *WikiWikiWeb* fue creado por Ward Cunningham, quien inventó y dio nombre al concepto y produjo la primera implementación de un servidor wiki para el Portland Pattern Repository [PPR] en 1995. En palabras del propio Ward Cunningham, un wiki es "*la base de datos en línea más simple que pueda funcionar*"[83].

10.1.1.2 Limitaciones de los Wikis

Los wikis tienen una serie de inconvenientes que los hacen insuficientes para resolver el problema planteado al inicio de esta tesis. El primero tiene que ver con

[82] Actualmente, el wiki más grande que existe es la versión en inglés de Wikipedia, seguida por varias otras versiones del proyecto [Wikipedia]

[83] "*the simplest online database that could possibly work*" en el texto original (notar cierto paralelismo con los principios de XP [Beck00], del cual Ward Cunningham es uno de los principales creadores junto a Kent Beck)

la estructura: la información se almacena como texto no estructurado (con algunas "instrucciones especiales"[84]), sin el grado de formalidad necesario para poder ser explotada, manipulada y transformada por herramientas automáticas. Otro problema relevante es que las relaciones entre las páginas son sintácticas (con las limitaciones que eso conlleva para relacionar, clasificar y categorizar información). Finalmente, la visualización y usabilidad en estos sistemas suele ser bastante precaria (siendo el caso extremo es el Portland Pattern Repostory [PPR]).

10.1.2 HTML

HTML (Hyper Text Markup Language) es un lenguaje de marcas para representar el contenido de páginas de hipertexto [BernersLee00]. Es el lenguaje en que están escritas la mayoría de las páginas Web [BernersLee00]. Contiene un conjunto de etiquetas básicas para representar documentos. De esta forma se puede estructurar el texto en cabeceras, listas, enlaces, etc.

HTML es un lenguaje de marcas no propietario basado en SGML (es una variante simplificada de éste) [W3C]. Una de las claves de su éxito frente a SGML es su facilidad de uso [BernersLee00] y corta curva de aprendizaje. Puede producirse con herramientas de baja tecnología (cualquier editor de textos es suficiente para crear documentos HTML).

XHTML (Lenguage de Marcado de Texto Extensible) es una versión más estricta y limpia de HTML, que nace precisamente con el objetivo de reemplazar HTML ante su limitación de uso con las cada vez más abundantes herramientas basadas en XML. XHTML extiende la sintaxis de HTML 4.0 combinando la sintaxis de HTML (diseñado para mostrar datos) con la de XML (diseñado para describir datos). [W3CES05]

La especificación y evolución de HTML y XHTML es controlada por el World Wide Web Consortium (W3C) [W3C].

10.1.2.1 Limitaciones de HTML

HTML está centrado en la visualización [W3CES05]. Los elementos que conforman la estructura de un documento HTML se refieren a secciones muy ligadas a la representación de hipertexto y están fuertemente relacionados con la presentación.

HTML puede describir con bastante eficiencia el nivel de conocimiento, pero tiene graves problemas para describirlo en forma abstracta. De hecho, dado que HTML tiene un enfoque de "publicación" (y un chequeo de consistencia sintáctica muy permisivo[85]), el nivel de abstracción y grado de formalidad con que se representa la información no es el adecuado para describir a los patrones en el modo que precisamos en esta tesis.

[84] Nos referimos a la sintaxis wiki, sobre la que se puede encontrar más información en [WPW]

[85] Los navegadores HTML tienen la capacidad de trabajar con documentos incompletos o mal formados. Los navegadores ignoran todos los errores e intentan inferir los resultados adecuados (en la medida de lo posible)

10.1.3 PLML y PLMLx

En la conferencia CHI 2003 [CHI2003], 15 participantes se reunieron en un workshop para discutir las perspectivas sobre los patrones de interacción persona ordenador[86] (HCI) [Interfaces03]. Un resultado significativo de este workshop es la especificación de *Pattern Language Markup Language* (en adelante, PLML). El objetivo de PLML es traer orden a las muchas e inconsistentes formas de describir los patrones (plantillas) utilizadas por los autores de patrones [PLML] (recordar la discusión sobre plantillas en el **capítulo 4**).

PLML se basa en un DTD[87] [W3C] donde se define un conjunto de elementos estándar que permiten describir los elementos más significativos de los patrones (soportando así un gran número de las plantillas existentes). En el siguiente bloque de código se muestra el DTD completo de la versión 1.1 de PLML (tomado de [PLML]):

```
<!ELEMENT pattern (name?, alias*, illustration?, problem?, context?, forces?,
solution?, synopsis?, diagram?, evidence?, confidence?, literature?,
implementation?, related-patterns?, pattern-link*, management?)>
<!ATTLIST pattern patternID CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT alias (#PCDATA)>
<!ELEMENT illustration ANY>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT context ANY>
<!ELEMENT forces ANY>
<!ELEMENT solution ANY>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT diagram ANY>
<!ELEMENT evidence (example*, rationale?)>
<!ELEMENT example ANY>
<!ELEMENT rationale ANY>
<!ELEMENT confidence (#PCDATA)>
<!ELEMENT literature ANY>
<!ELEMENT implementation ANY>
<!ELEMENT related-patterns ANY>
<!ELEMENT pattern-link EMPTY>
<!ATTLIST pattern-link
  type CDATA #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #REQUIRED
  label CDATA #REQUIRED>
<!ELEMENT management (author?, credits?, creation-date?, last-modified?,
revision-number?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT creation-date (#PCDATA)>
<!ELEMENT last-modified (#PCDATA)>
<!ELEMENT revision-number (#PCDATA)>
```

Código 10.1 – DTD de PLML

Existe una versión extendida de PLML llamada PLMLx (Extended PLML) [PLMLX], que añade una serie de características al lenguaje. Si bien PLMLx añade

[86] Human Computer Interaction (HCI) en el texto original
[87] Document Type Definition

funcionalidades a la especificación original, también restringe el espectro de patrones que pueden ser definidos utilizándolo (por ejemplo, excluye a todos los patrones de UI de Jennifer Tidwell [Tidwell05]) y añade una serie de campos obligatorios que pueden resultar inconvenientes cuando se definen el resto de los patrones [Fincher04].

Un aspecto importante a destacar sobre PLML y PLMLx es que han sido diseñados para describir patrones de HCI, con lo cual pueden tener limitaciones para describir otros tipos de patrones (esto se explica mejor en la siguiente sección).

10.1.3.1 Limitaciones de PML

PLML ayuda a traer orden a la descripción de patrones, pero tiene varios inconvenientes y limitaciones, tales como tener un conjunto predefinido de elementos para describir el nivel de conocimiento del patrón y la falta de soporte para describir en forma abstracta el nivel de implementación.

Puede existir una distancia considerable entre el DTD de PLML y la plantilla utilizada para describir ciertos patrones. Por ejemplo, hay varias secciones de los patrones de "*Design Patterns*" [GoF95] que no están incluidas en el DTD. Lo mismo sucede para los patrones de "*Patterns of Software Architecture, Volume 1*" [POSA96] o para los de "*Patterns of Enterprise Application Architecture*" [Fowler02]. Describir estos patrones utilizando PLML implica un importante esfuerzo de traducción (que en el peor de los casos puede ser en vano). Adicionalmente, PLML no incluye construcciones para describir el nivel de implementación a un nivel alto de abstracción.

Finalmente, PLML y PLMLx han sido creados para describir patrones. Por lo tanto, no pueden expresar conceptos soporte y de ingeniería del software como principios de orientación a objetos, lenguajes de patrones, categorías de patrones, etc. [Welicki06]. Esto dificulta la creación de redes de entidades de distintos tipos.

10.1.3.2 Adopción de PLML

En el estudio sobre gestión de colecciones[88] de patrones titulado "*Managing UI Patterns Collections*" [DKT05] realizado por investigadores de la Universidad Massey [MU] de Nueva Zelanda y publicado por la ACM [ACM] en 2005, se analizan 15 catálogos públicos de patrones de UI[89]. Según ese trabajo ninguno de los catálogos estudiados utilizaba PLML.

10.1.4 XMI

El objetivo principal de XMI es facilitar el intercambio de metadatos entre herramientas de modelado (basadas en UML [UML] de OMG [OMG]) y

[88] Estas colecciones pueden ser lenguajes o sistemas de patrones.
[89] UI se refiere a User Interface (Interfaz de Usuario)

repositorios de metadatos (basados en el MOF [OMGMOF] de OMG [OMG]) en entornos heterogéneos distribuidos [OMGXMI]. XMI integra tres estándares clave [OMGXMI]:

- **XML** (Extensible Markup Language), recomendación del W3C.
- **UML** (Unified Modelling Language), estándar de modelado del OMG.
- **MOF** (Meta Object Facility), estándar del OMG para metamodelado y repositorios de metadatos.

La integración de estos tres estándares en XMI combina lo mejor del W3C [W3C] y del OMG [OMG] en referencia a tecnologías de modelado y metadatos, permitiendo a los desarrolladores de sistemas distribuidos compartir modelos de objetos y metadatos en Internet [OMGXMI].

XMI, junto con UML y MOF forman el núcleo de la arquitectura de repositorios de metadatos del OMG, definiendo una herramienta de modelado orientado a objetos (UML) y un mecanismo para compartir metadatos de estos modelos (MOF). XMI disminuye la barrera de entrada para utilizar los estándares de metadatos del OMG. [OMGXMI]

10.1.4.1 Limitaciones de XMI

Si bien XMI es muy potente, está muy orientado a la representación de modelos y tiene un acoplamiento muy fuerte con las tecnologías del OMG (su uso implica el de UML y MOF). XMI permite representar modelos UML en un formato común, pero es relativamente limitado respecto a la información adicional que puede tolerar utilizando la sintaxis estándar [EAXL]. Por ejemplo, en el caso de los modelos de *Enterprise Architect* [EA], mucha de la información sobre el modelo debe ser convertido a lo que se conoce como "*Tagged Values*" los cuales se importarán a otros sistemas de modelado como información adicional o serán ignorados completamente [EAXL].

Lo expuesto en párrafo anterior dificulta la descripción del nivel de conocimiento, enfocando la solución a la descripción del nivel de implementación (estructura de la solución). Este camino descarta gran parte de la información más importante de los patrones (el nivel de conocimiento). También dificulta la descripción de conceptos de soporte, dado que en la mayoría de los casos no tienen un modelo de objetos asociado, sino que son un conjunto de descripciones con metadatos descriptivos (para clasificación y búsquedas).

Un hecho concreto que refuerza lo expuesto en este párrafo es que ninguno de los once catálogos públicos de patrones analizados posteriormente en este capítulo (§ 10.2) utiliza XMI. En contrapartida, es utilizado por algunas de las herramientas de modelado analizadas más adelante (§ 10.3).

10.1.5 ODOL (Proyecto WOP)

El proyecto "*Web of Patterns Project*" tiene como objetivo representar diseños de software utilizando tecnologías de Web Semántica [BHL01] [DOS03], en particular RDF[90] y OWL[91] [WOP]. Su propósito es representar patrones, antipatrones y refactorizaciones utilizando los conceptos desarrollados en una ontología [WOP]. Sus objetivos incluyen la creación de la ontología y un conjunto de herramientas para luego compartirlas con la comunidad de ingeniería del software. [WOP]

El proyecto consta de tres partes [WOP]:

- La ontología de diseño de software orientado a objetos *ODOL* (*Object-Oriented Software Design Ontology*).
- Descripciones de patrones y catálogo basados en ODOL.
- El "*WOP Pattern Scanner*", que es una herramienta que encuentra patrones de diseño en programas Java.

La ontología ODOL (que es el núcleo del proyecto) es un grupo pequeño de conceptos y relaciones utilizados en los lenguajes de programación orientados a objetos más importantes [WOP]. Incluye conceptos básicos (como clases y métodos) los cuales están implementados directamente en los lenguajes orientados a objetos, conceptos de diseño como "asociación" y conceptos de alto nivel como patrones, categorías de patrones y sus relaciones (como por ejemplo refinamiento) [WOP]. Estos conceptos han sido modelados utilizando el lenguaje de ontologías OWL [WOP]. En la figura a continuación se muestra la representación del patrón ABSTRACT FACTORY [GoF95].

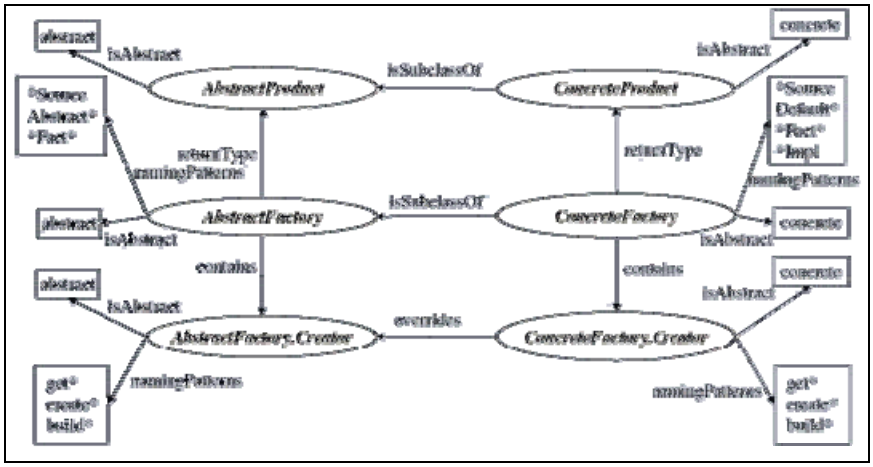


Figura 10.1 – Representación de Abstract Factory en ODOL, tomada de [WOP].

[90] Resource Description Framework
[91] Ontology Web Language

10.1.5.1 Limitaciones de ODOL

Este enfoque está más bien orientado a describir el nivel de implementación, dejando de lado la descripción detallada del nivel de conocimiento. Adicionalmente, la creación de la ontología es un proceso complejo que requiere un enfoque "bottom-up". Esto hace que la producción de nuevos patrones utilizando este enfoque sea más compleja.

10.1.6 Lattice Based Classification

Hasso and Carlson, del *Illinois Institute of Technology* [IIT], han propuesto y en el PLoP[92] 2005 [PLoP05] un modelo de clasificación basado en teoría lingüística y estructuras algebraicas complejas [HC05]. Este modelo permite clasificar y organizar patrones para poder identificar y diferenciar los patrones adecuados en un contexto específico. En la siguiente figura se muestra un ejemplo de organización del cluster de patrones PROXY (en [HC05] se explica detalladamente la técnica de clasificación y la forma de obtener este gráfico).

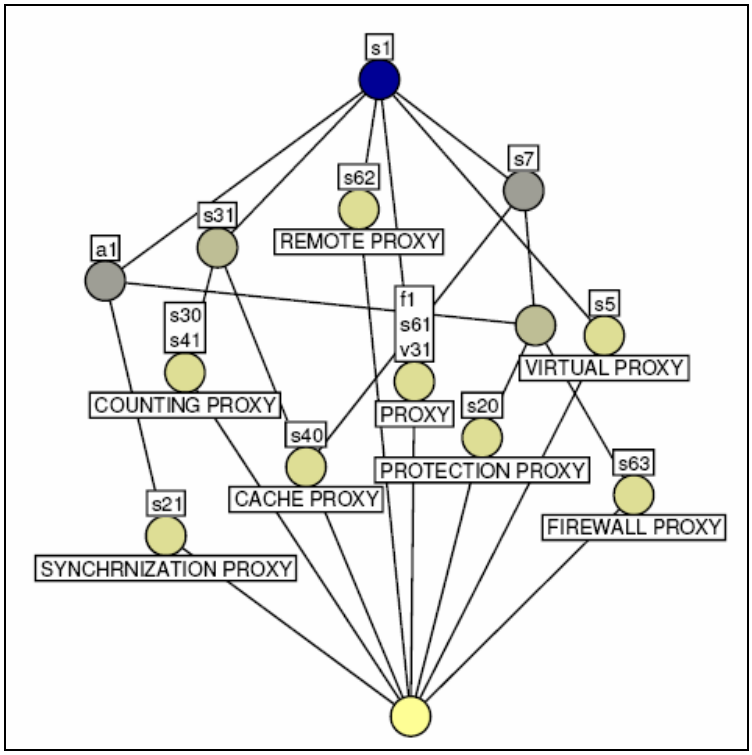


Figura 10.2 – Sub-diagrama de Lattice de las variantes del patrón Proxy, tomada de [HC05].

10.1.6.1 Limitaciones de Lattice Based Classification

Si bien el modelo es muy potente, es muy complejo de poner en práctica. La clasificación de los patrones en grupos requiere un importante trabajo de análisis (como puede observarse en la figura 10.1, arriba). Otro problema más grave aún es

[92] Pattern Lanaguages of Programs Conference

la dificultad para evolucionar las categorías creadas (que requiere tanto o más análisis que la clasificación inicial).

10.2 Catálogos Públicos de Patrones en la Web

En la actualidad existe una amplia variedad de catálogos públicos de patrones en la Web. En esta sección exploraremos algunos de los más populares. Una gran cantidad de estos catálogos Web contienen principalmente patrones de interacción persona ordenador (esta área es una de las más activas en el campo de la descripción y catalogación de patrones, siendo la impulsora de iniciativas como PLML, § 10.1.3). En cada caso realizaremos un breve análisis sobre las tecnologías utilizadas para describir a los patrones, de las fortalezas y de las debilidades y carencias de cada aplicación.

10.2.1 Portland Pattern Repository

http://c2.com/ppr/index.html

El *Portland Pattern Repository* [PPR] ha sido el primer repositorio de patrones en la Web y también el primer *wiki* (creado por Ward Cunnigham, § 10.1.1). Contiene una amplia variedad de contenido introducido y mantenido por varias de las figuras de referencia de la comunidad de patrones[93]. Contiene información sobre una gran variedad de temas referentes a los patrones y a los conceptos de ingeniería del software que los rodean. Toda esta información hace que sea más fácil comprender realmente a los patrones (dado que se puede acceder mediante hipervínculos a los conceptos en los que se fundan los patrones, obras en que están incluidos, etc.). En la siguiente figura se muestra una página del Portland Patterns Repository.

Figura 10.3 – Página de ejemplo del Portland Pattern Repository

10.2.1.1 Fortalezas

- Puede representar patrones y cualquier tipo de concepto (autores, refactorizaciones, libros, conceptos generales, etc.).
- Permite edición colaborativa.

[93] La lista de colaboradores incluye a Kent Beck, Ward Cunningham, Ralph Johnson, Robert C. Martin, etc.

- Es el repositorio más antiguo, conteniendo información y experiencia de ingenieros destacados durante más de una década.
- Tiene un buscador sencillo.
- Esta "vivo". Tiene constante actividad.
- Modelo de "navegación por descubrimiento".
- Cubre una amplia variedad de temas.
- Fácil de usar y de aprender.

10.2.1.2 Debilidades

- Las entidades no tienen estructura. Son bloques de textos con enlaces.
- No se pueden representar en forma abstracta los niveles de conocimiento e implementación.
- No se pueden generar artefactos de software a partir de las descripciones.
- Las relaciones son sintácticas (basadas en hipervínculos).
- No tiene un buen sistema de navegación.
- Buscador muy básico.
- Interfaz de usuario muy primitiva.
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.

10.2.2 Patterns Share

http://www.patternshare.org

PatternShare [Patternshare] es una iniciativa de Microsoft [Microsoft] con el objeto de agrupar patrones de distintos autores y lenguajes de patrones. Esta creado sobre una plataforma de wiki, lo cual permite la edición colaborativa. El modelo de navegación por los catálogos está fundado principalmente en la visión de patrones de Microsoft. En la siguiente figura se muestra la página inicial de Patternshare.

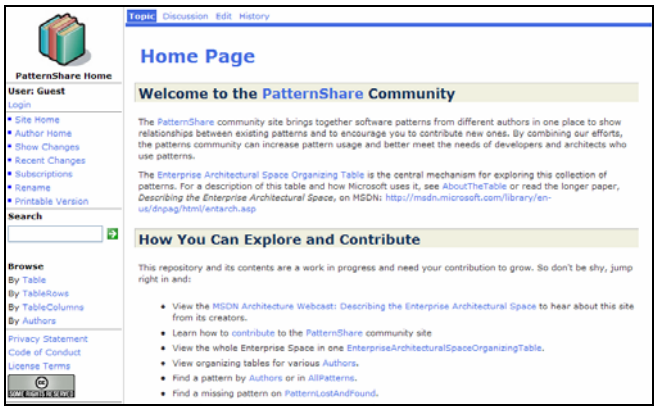


Figura 10.4 – Página de inicio de Patternshare.org

10.2.2.1 Fortalezas

- Permite edición colaborativa

- Contiene patrones de varios catálogos ([GoF95], [POSA96], [Microsoft03b], etc.).
- Tiene un de actividad considerable.
- Provee un mecanismo de búsqueda sencillo.
- Muestra las novedades en la página de inicio.
- Soporte institucional de una gran empresa de software.
- Buscador.
- Permite navegar por el catálogo a partir de varios criterios.
- Soporte para discusiones sobre cualquier elemento (página wiki) en el sistema.

10.2.2.2 Debilidades

- Si bien las plantillas de patrones pueden definirse, una vez que se crean son fijas.
- No se puede representar en forma abstracta los niveles de conocimiento e implementación.
- No se pueden generar artefactos de software a partir de las descripciones.
- Las relaciones son sintácticas (basadas en hipervínculos).
- El sistema de navegación por el catálogo no es claro ni intuitivo. Está basado en las vistas de agrupación propuestas por Microsoft en [Microsoft03b].
- Sólo hay una única forma de visualizar el patrón.

10.2.3 Sun's Core J2EE Patterns

http://www.corej2eepatterns.com/Patterns2ndEd/index.htm

Este catálogo [CJ2P] contiene una versión electrónica de extractos de los contenidos del libro "*Core J2EE Patterns*" [ACM01]. Es un conjunto de páginas HTML que pueden ser visualizadas en forma sencilla utilizando un navegador de Internet. Está orientado a un tipo específico de patrones (que son los contenidos en la obra anteriormente mencionada). En las siguientes figuras se muestran el mapa de patrones (que permite navegar en forma visual por el catálogo) y la plantilla de un patrón concreto.

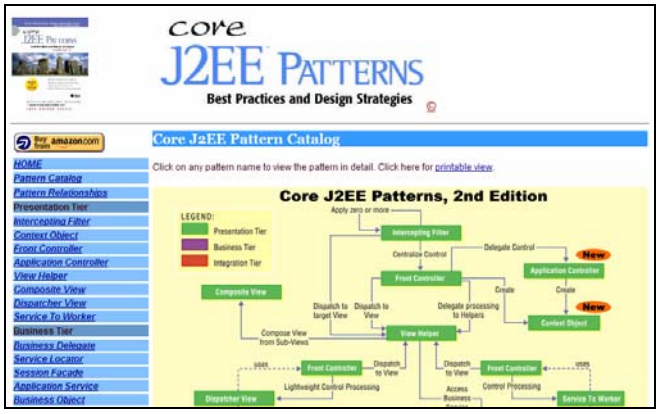


Figura 10.5 – Mapa de patrones de "Core J2EE Patterns"

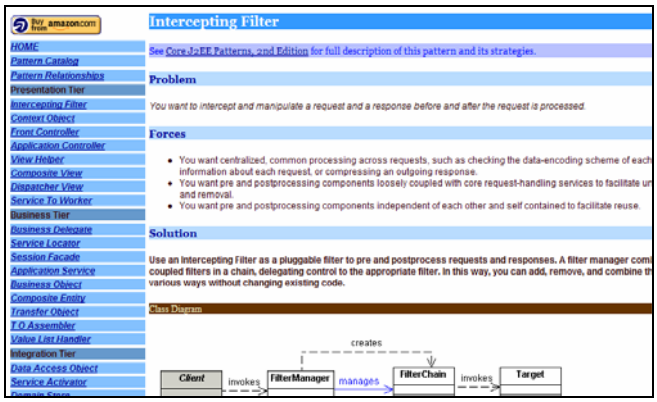


Figura 10.6 – Plantilla de un patrón de "Core J2EE Patterns"

10.2.3.1 Fortalezas

- El modelo de navegación es muy claro: los patrones están categorizados y es fácil manejarse a través de los patrones relacionados.
- La plantilla es clara y contiene información relevante.
- La plantilla utilizada es homogénea.
- Buena usabilidad.
- Está fundado en contenidos ampliamente aceptados por la comunidad.
- Contiene un mapa para navegar por los contenidos del catálogo. Ese mapa permite visualizar en forma rápida los patrones del catálogo junto con sus relaciones.
- Soporte institucional de una gran empresa de software.
- Sólo hay una única forma de visualizar el patrón.

10.2.3.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Sólo muestra patrones de un único tipo y no incluye información sobre conceptos de soporte (categorías, refactorizaciones, etc.).
- Las entidades no tienen una estructura que pueda ser explotada automáticamente. Están definidas mediante etiquetas HTML.
- No se puede representar en forma abstracta los niveles de conocimiento e implementación.
- No se pueden generar artefactos de software a partir de las descripciones.
- Las relaciones son sintácticas (basadas en hipervínculos).
- No tiene buscador.
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.4 Montreal Online Usability DIgital Library (MOUDIL)

http://hci.cs.concordia.ca/moudil/

MOUDIL (*Montreal Online Usability DIgital Library*) es un sistema para compartir patrones de interfaz de usuario [MOUDIL]. Ha sido creado con dos objetivos: 1) Ofrecer acceso a los desarrolladores a un amplio conjunto de patrones de Interfaz de Usuario (UI) y 2) Actuar como un foro donde los investigadores puedan explorar colecciones de patrones para descubrir más sobre la naturaleza de esas colecciones. Si bien las funcionalidades de este catálogo son prometedoras, su gran mayoría está en construcción y aparentemente no se actualiza[94].

Figura 10.7 – Pagina inicial del catálogo Web "MOUDIL"

Figura 10.8 – Plantilla de un patrón en el catálogo Web "MOUDIL"

10.2.4.1 Fortalezas

- El modelo de navegación es muy claro: los patrones están categorizados y es fácil manejarse a través de los patrones relacionados.
- Permite navegar los patrones del catálogo en varias formas (alfabética, por aplicación y por contribuidor).
- La plantilla es clara y contiene información relevante.
- La plantilla utilizada es homogénea.
- Permite a los usuarios armar su propio espacio de patrones (seleccionando patrones).
- Ofrece varios niveles de búsqueda (básica y avanzada).
- Soporte para discusión.

[94] La última fecha de actualización es de finales de 2004. La última fecha de revisión de este documento es mediados de 2006.

- Muestra los patrones más accedidos.

10.2.4.2 Debilidades

- Gran parte del sitio está en construcción.
- No incluye información sobre conceptos de soporte (categorías, refactorizaciones, etc.).
- Los patrones no incluyen el nivel de implementación.
- No se pueden generar artefactos de software a partir de las descripciones.
- No se muestran las relaciones entre patrones.
- No hay figuras asociadas a los patrones, lo cual dificulta su compresión.
- Aparentemente, no se actualiza hace tiempo[95].
- Sólo hay una única forma de visualizar el patrón.

10.2.5 Patterns Almanac (version Web)

http://www.smallmemory.com/almanac/index.html

"*Patterns Almanac*" [Rising00] ha sido una iniciativa para la creación un directorio (con referencias a las fuentes originales) de los patrones existentes en un momento determinado. Lamentablemente luego de su lanzamiento impreso [Rising00] no ha sido actualizado[96]. La versión Web del "Patterns Almanac" [PAW] está desarrollada utilizando un conjunto de páginas HTML con la información de cada patrón en la versión impresa[97]. En la siguiente figura se muestra la pantalla inicial del "Patterns Almanac"

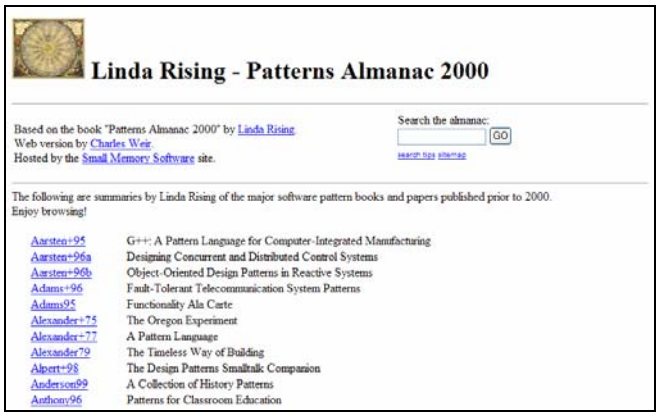


Figura 10.9 – Página inicial de la versión Web del "Patterns Almanac"

10.2.5.1 Fortalezas

- Incluye referencias a una amplia variedad de patrones (todos los patrones publicados antes del año 2000)

[95] En la sección discusión hay un hilo titulado "is this site alive" (con fecha del 28 de enero de 2004) sin respuesta....
[96] La última fecha de actualización es Noviembre de 2004
[97] Para cada patrón sólo se incluye información de catalogación y una frase que lo resume.

- Muestra referencias a patrones de todos los tipos (diseño, arquitectura, procesos, UI, etc.).
- Incluye todos los contenidos de "*Patterns Almanac*" [Rising00].
- La interfaz de usuario es muy fácil de aprender y usar.
- Tiene búsquedas y mapa del sitio.
- Los patrones están relacionados entre sí.
- Se pueden ver todos los patrones de un libro específico, junto con los datos de referencia de la obra.

10.2.5.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Las entidades no tienen una estructura que pueda ser explotada automáticamente. Están definidas mediante etiquetas HTML.
- No se puede representar en forma abstracta los niveles de conocimiento e implementación.
- Solo muestra un breve resumen del patrón y un apuntador a la fuente donde ha sido descrito. No incluye los niveles de conocimiento e implementación.
- No se pueden generar artefactos de software a partir de las descripciones.
- Las relaciones son sintácticas (basadas en enlaces de hipertexto).
- Desactualizado. Los patrones a los que se hace referencia son los que han sido publicados antes de 2000. Hay secciones con enlaces a documentos que no son actualizadas desde Noviembre de 2004[98].
- La navegación es muy rudimentaria: sólo se presenta una lista alfabética con los nombres de patrones.
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.6 Martin Fowler's Enterprise Architecture Catalog

http://www.martinfowler.com/eaaCatalog/

El catálogo de patrones de arquitectura de aplicaciones empresariales [EAAC] es un complemento al libro "*Patterns of Enterprise Application Architecture*" [Fowler02]. Contiene versiones resumidas de los patrones incluidos en esa obra. Los patrones están descritos utilizando HTML y relacionados mediante hipervínculos. En cada patrón se incluye además de su nombre la categoría a la que pertenece y la página donde comienza su descripción en el libro. En las siguientes figuras se muestra la página inicial y la descripción de un patrón en este catálogo.

[98] Este documento se ha completa en el ultimo trimestre de 2006.


A short summary of the patterns in *Patterns of Enterprise Application Architecture (P of EAA)*.

These pages are a brief overview of each of the patterns in P of EAA. They aren't intended to stand alone, but merely as a quick aide-memoire for those familiar with them, and a handy link if you want to refer to one online. In the future I may add some post-publication comments into the material here, but we'll see how that works out.

Many of these diagrams demonstrate the rather poor GIF output of Visio. The nice diagrams were redrawn for me by David Heinemeier Hansson

Domain Logic Patterns: Transaction Script (110), Domain Model (116), Table Module (125), Service Layer (133).

Data Source Architectural Patterns: Table Data Gateway (144), Row Data Gateway (152), Active Record (160), Data Mapper (165).

Object-Relational Behavioral Patterns: Unit of Work (184), Identity Map (195), Lazy Load (200)

Object-Relational Structural Patterns: Identity Field (216), Foreign Key Mapping (236), Association Table Mapping (248), Dependent Mapping (262), Embedded Value (268), Serialized LOB (272), Single Table Inheritance (278), Class Table Inheritance (285), Concrete Table Inheritance (293), Inheritance Mappers (302).

Object-Relational Metadata Mapping Patterns: Metadata Mapping (306), Query Object (316), Repository (322).

Web Presentation Patterns: Model View Controller (330), Page Controller (333), Front Controller (344), Template View (350), Transform View (361), Two-Step View (365), Application Controller (379).

Distribution Patterns: Remote Facade (388), Data Transfer Object (401)

Offline Concurrency Patterns: Optimistic Offline Lock (416), Pessimistic Offline Lock (426), Coarse Grained Lock (438), Implicit Lock (449).

Figura 10.10 – Página inicial del catálogo Web de "*Patterns of Enterprise Application Architecture*"

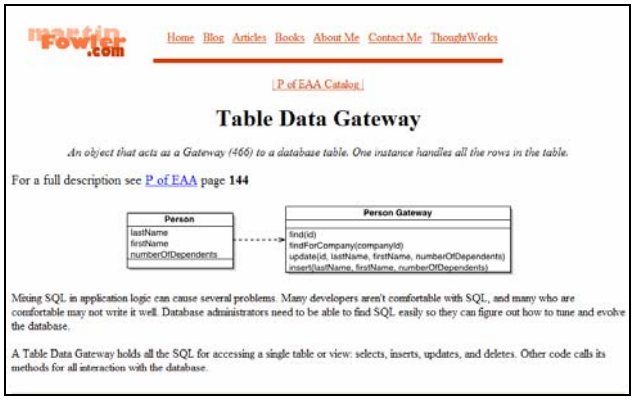


Figura 10.11 – Plantilla de un patrón en el catálogo Web de "*Patterns of Enterprise Application Architecture*"

10.2.6.1 Fortalezas

- Incluye los resúmenes de todos los patrones de un catálogo concreto.
- Los patrones son fácilmente legibles en la Web.
- La interfaz de usuario es muy fácil de aprender y usar.
- Buen complemento de la obra impresa ([Fowler02]).

10.2.6.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Las entidades no tienen una estructura que pueda ser explotada automáticamente. Están definidas mediante etiquetas HTML.
- No se puede representar en forma abstracta los niveles de conocimiento e implementación.
- Sólo muestra un breve resumen del patrón y un apuntador a la fuente donde ha sido descrito. Incluye un breve resumen del nivel de conocimiento.
- No se pueden generar artefactos de software a partir de las descripciones
- Sólo describe patrones. No hay descripciones de otros conceptos como categorías, refactorizaciones, principios, etc.
- Los patrones no están relacionados entre sí. No se puede navegar desde un patrón a otro relacionado.

- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.7 Enterprise Integration Patterns Catalog

http://enterpriseintegrationpatterns.com/eaipatterns.html

El catálogo de patrones de integración de aplicaciones empresariales [EIPC] es un complemento al libro "*Enterprise Integration Patterns*" [HW03]. Contiene versiones resumidas de los patrones incluidos en esa obra. Los patrones están descritos utilizando HTML y relacionados mediante hipervínculos. En las siguientes figuras se muestra la página inicial y la descripción de un patrón en este catálogo.

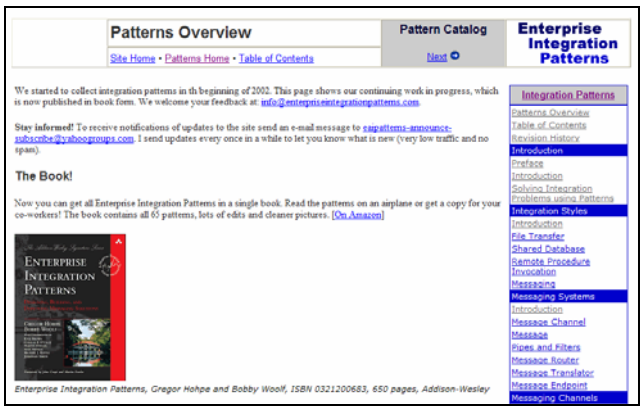


Figura 10.12 – Página inicial del catálogo Web de "Enterprise Integration Patterns"

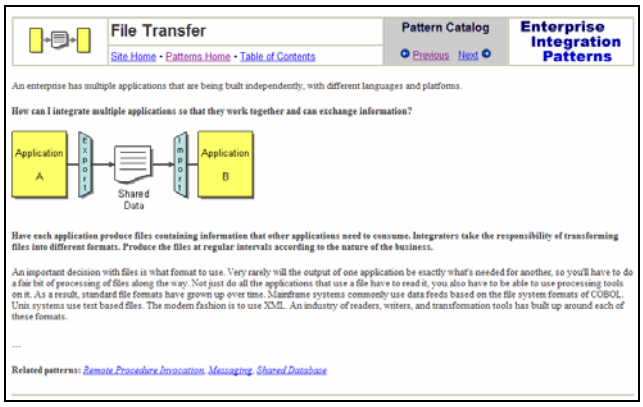


Figura 10.13 – Plantilla de un patrón en el catálogo Web de "Enterprise Integration Patterns"

10.2.7.1 Fortalezas

- Incluye los resúmenes de todos los patrones de un catálogo concreto.
- Los patrones son fácilmente legibles en la Web.
- La interfaz de usuario es muy fácil de aprender y usar.
- El catálogo es fácil de navegar.
- Contiene información contextual sobre el catálogo (qué es, para qué sirve, etc.).

- Los patrones están relacionados, permitiendo navegar el catálogo a través de las relaciones.
- Pueden discutirse los patrones utilizando un Wiki creado a tales efectos.
- Buen complemento de la obra impresa ([HW03]).

10.2.7.2 Debilidades

- Las entidades no tienen una estructura que pueda ser explotada automáticamente. Están definidas mediante etiquetas HTML.
- No representa en forma abstracta los niveles de conocimiento e implementación.
- Sólo muestra un breve resumen del patrón y un apuntador a la fuente donde ha sido descrito. Incluye un breve resumen del nivel de conocimiento.
- No se pueden generar artefactos de software a partir de las descripciones.
- Las relaciones son sintácticas (basadas en enlaces de hipertexto).
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.8 Patterns en el Handbook of Software Architecture

http://www.booch.com/architecture/patterns.jsp

La sección de patrones [PHSA] del "*Handbook of Software Architecture*" [Booch] agrupa diversos patrones de varias obras. Es una iniciativa liderada por Grady Booch similar a la que ha llevado a cabo varios años atrás Linda Rising [Rising00]. Contiene más de 450 patrones de distintos libros, conferencias y trabajos de investigación.

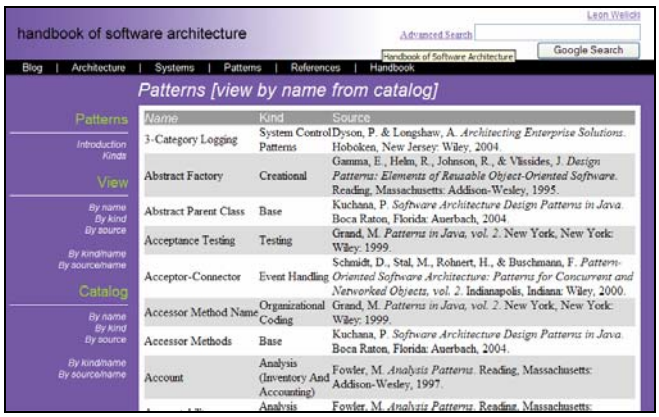


Figura 10.14 – Lista de patrones en el "Handbook of Software Architecture"

10.2.8.1 Fortalezas

- Incluye referencias a una amplia variedad de patrones.
- Incluye patrones de más de 45 obras distintas.

- Muestra referencias a patrones de todos los tipos (diseño, arquitectura, procesos, UI, etc.).
- La interfaz de usuario es muy fácil de aprender y usar.
- Permite navegar por los patrones en diversas formas (por nombre, por autor, etc.).
- Tiene búsquedas (utilizando Google).
- Pueden verse todos los patrones de un libro específico, junto con los datos de referencia de la obra.

10.2.8.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Sólo se describen patrones (no se describen conceptos de soporte).
- No incluye información de implementación.
- No se muestran relaciones entre patrones.
- Sólo muestra el resumen muy básico de cada patrón (por ejemplo, en el caso del GoF la intención).
- No se pueden generar artefactos de software a partir de las descripciones.
- Sólo hay una única forma de visualizar el patrón.

10.2.9 Patterns in Interaction Design (PID)

http://www.welie.com/

Patterns in Interaction Design [PID] es un sitio Web que contiene publicados un conjunto de patrones orientados a la construcción de interfaces de usuario y diseño de interacción. Las categorías de patrones que se presentan en este sitio incluyen patrones para la Web, para interfaz gráfica de usuario (GUI) y para dispositivos móviles. La información se almacena en XML y el código HTML de salida se genera mediante una transformación XSL[99]. En las siguientes figuras se muestran la página de entrada de la sección "*Web Design Patterns*" y la plantilla de un patrón.

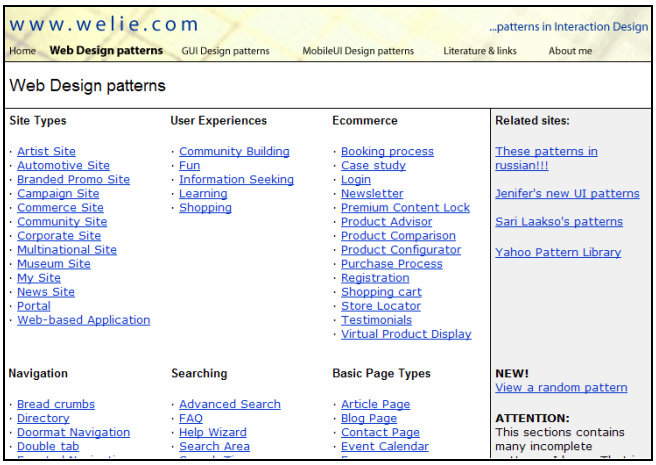


Figura 10.15 – Pagina principal con la lista de patrones en "Welie.com"

[99] Según nos ha explicado el creador del sitio en un correo electrónico.

Figura 10.16 – Plantilla de un patrón en "Welie.com"

10.2.9.1 Fortalezas

- Incluye un amplio catalogo de patrones de UI, Movilidad y Web.
- La interfaz de usuario es muy fácil de aprender y usar.
- La navegación es simple y clara.
- La plantilla utilizada es consistente y fácil de leer.
- Los ejemplos utilizados son fácilmente identificables y comprensibles.
- Contiene ejemplos de sitios reales, con referencias contrastables.
- La información se almacena en un formato abstracto independiente de la visualización.
- Contiene referencias a libros, sitios Web y artículos donde puede encontrarse más información sobre el patrón.

10.2.9.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Sólo se describen patrones (no se describen conceptos de soporte).
- No tiene buscador.
- La plantilla tiene secciones que no se utilizan (por ejemplo, las secciones "*related patterns*" e "*implementation*" suelen estar vacías).
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.10 UI Patterns

http://designinginterfaces.com/

El catálogo de patrones de interfaz de usuario [Tidwell] es un complemento al libro "*Designing Interfaces*" [Tidwell05]. Contiene versiones resumidas de los patrones incluidos en esa obra. Los patrones están descritos utilizando HTML y relacionados mediante hipervínculos. En las siguientes figuras se muestra la página inicial y la descripción de un patrón en este catálogo.

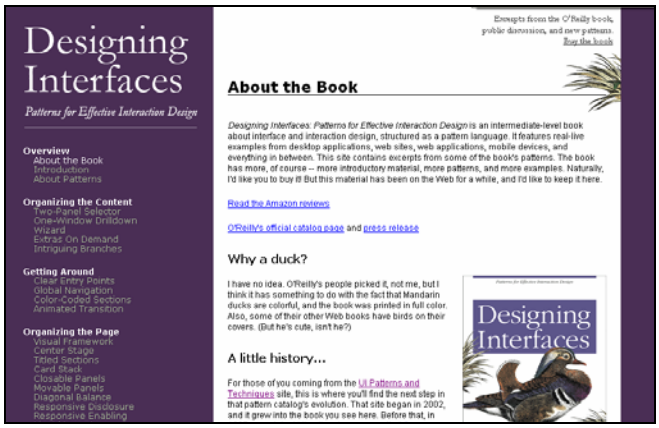


Figura 10.17 – Pagina principal del catálogo "Designing Interfaces"

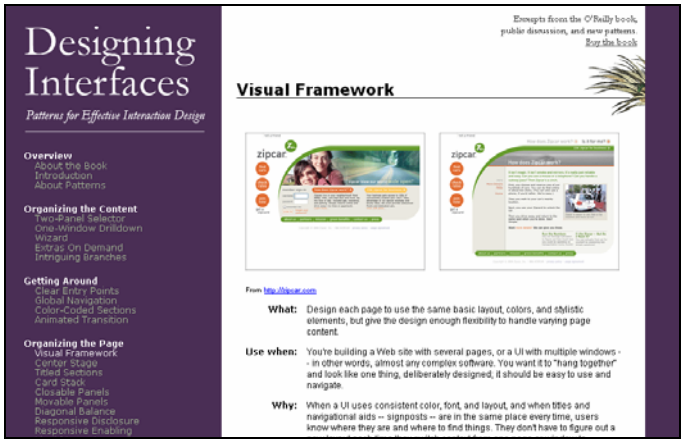


Figura 10.18 – Plantilla de un patrón en el catálogo "Designing Interfaces"

10.2.10.1 Fortalezas

- Incluye un amplio catalogo de patrones de UI.
- La interfaz de usuario es muy fácil de aprender y usar.
- La navegación es simple y clara.
- El texto esta preparado para ser leído en la Web.
- La plantilla utilizada es consistente y fácil de leer.
- Los ejemplos utilizados son fácilmente identificables y comprensibles.
- El modelo de navegación es muy claro: los patrones están categorizados y es fácil manejarse a través de los patrones relacionados.
- Los patrones están relacionados, permitiendo navegar el catálogo a través de las relaciones.

10.2.10.2 Debilidades

- No tiene funcionalidades para discusión / comunidad.
- Sólo se describen patrones (no se describen conceptos de soporte).
- No tiene buscador.
- No se pueden generar artefactos de software a partir de las descripciones.

- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.
- Sólo hay una única forma de visualizar el patrón.

10.2.11 DoFactory GoF Patterns

http://www.dofactory.com/Patterns/Patterns.aspx

Esta página [DoFactory] es muy popular en el mundo de desarrollo en ambientes .NET. Contiene las implementaciones y descripciones estructurales de los 23 patrones del GoF. La página pertenece a una empresa dedicada a consultoría y formación en desarrollo, especializada en temas relacionados con ingeniería del software. Desde esta página ofrecen servicios y productos a los usuarios. En la siguiente figura se muestra la página que permite navegar por el catálogo de patrones.

Figura 10.19 – Lista de patrones en el catálogo "DoFactory"

10.2.11.1 Fortalezas

- Incluye ejemplos de implementaciones para los 23 patrones del GoF.
- Contiene una clasificación por "frecuencia de uso" de cada patrón.
- Además de la implementación general muestra un ejemplo del patrón aplicado en un contexto concreto.
- Contiene implementaciones optimizadas para .NET (aunque para acceder a esta opción es necesario pagar previamente una tarifa).

10.2.11.2 Debilidades

- La descripción del nivel de conocimiento es incompleta.
- No tiene funcionalidades para discusión / comunidad.
- Sólo se describen patrones (no se describen conceptos de soporte).
- No se puede representar en forma abstracta los niveles de conocimiento e implementación.
- No se muestran relaciones entre patrones.

- Sólo incluye información de implementaciones concretas del patrón (no se muestra información relacionada con el contexto y las fuerzas).
- No se pueden generar artefactos de software a partir de las descripciones.
- El sistema de categorización para navegar por el catálogo es fijo. No permite al usuario utilizar categorizaciones alternativas.

10.3 Representación de Patrones en Herramientas Comerciales

Actualmente las herramientas modernas de modelado tienen funcionalidades orientadas al uso de patrones. En todos los casos es posible navegar por un catálogo de patrones y utilizarlos en los modelos. En la siguiente tabla se analiza brevemente la descripción y uso de patrones en tres de las herramientas comerciales más populares (Togheter [Borland], Rational XDE [Rational] y Enterprise Architect [EA]).

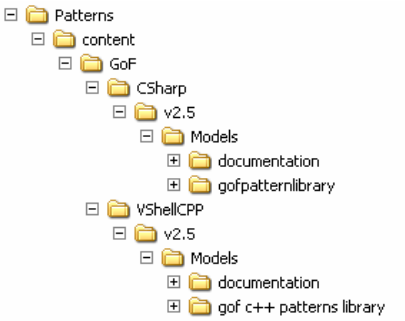
Herramienta	Cómo se representan los patrones…
Rational XDE [Rational]	Los patrones se especifican utilizando una combinación de ficheros XML, XSD y HTML. Es necesario crear un conjunto de ficheros para cada implementación del patrón orientada a un lenguaje de programación específico. El fichero más importante es un documento XML (cuyo nombre generalmente coincide con el del patrón) que tiene toda información necesaria para utilizar el patrón en XDE. En la siguiente figura se muestra la estructura de carpetas de los ficheros XDE para representar los patrones (en la versión .NET). Obsérvese que para cada implementación destino (C#, C++, etc.) existe una carpeta donde están alojados los ficheros necesarios. 
Borland Together [Together]	Los patrones se especifican utilizando un conjunto de ficheros XML con un formato propietario. El nivel de conocimiento se modela en forma incompleta (sólo se incluye información general sobre el patrón). Respecto al nivel de implementación, deben crearse ficheros especiales para cada lenguaje destino (la representación no es totalmente abstracta). La siguiente

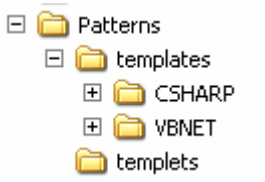
	figura muestra la estructura de carpetas para la versión de .NET. 
Enterprise Architect [EA]	Los patrones se especifican en un fichero XML, utilizando XMI. La definición del patrón no incluye la información completa del nivel de conocimiento. Respecto al nivel de implementación, sólo se incluyen las interfaces de los participantes (información estructural), pero no el comportamiento de sus métodos.

Tabla 10.1 – Representación y utilización de patrones en las herramientas de modelado

En [Welicki04c] y [Welicki05] se incluye un estudio más detallado sobre la representación de los patrones en estas herramientas.

10.4 Conclusión: Problemas Recurrentes

A lo largo de este análisis hemos encontrado problemas recurrentes en todos los enfoques revisados. Presentaremos estos problemas en cada una de las categorías establecidas en la introducción de este capítulo:

- **Modelos de descripción y catalogación**: la mayoría de los enfoques tienen problemas para representar el nivel de conocimiento en forma apropiada. En muchos casos el nivel de conocimiento no se incluye (o sólo se incluye parcialmente). Un aspecto interesante es que los modelos de descripción que mejor representan el nivel de conocimiento están a un nivel muy bajo de abstracción y no pueden representar en forma abstracta el nivel de implementación. La gestión y representación de múltiples plantillas de patrones en forma dinámica es otro problema importante que no ha sido resuelto adecuadamente por ninguno de los enfoques. Respecto a los modelos de catalogación, en la mayoría de los casos el problema es la complejidad de producción de los catálogos. Los lenguajes y modelos con mayor especialización para la descripción de entidades a un alto nivel de abstracción (PLML, PLMLx, XMI, ODOL, etc.) tienen graves dificultades para representar entidades que no sean patrones[100].

- **Catálogos públicos existentes en la Web**: en la mayoría de los casos nos hemos encontrado con que la navegación por el catálogo es fija (utilizando una única metáfora de navegación) y que no es posible tener diferentes vistas

[100] De hecho en la mayoría de los casos no pueden representarlas. PLML sólo puede incluir algo de información sobre la fuente, categoría y sistema en que está incluido el patrón. ODOL y XMI pueden representar elementos que tengan estructura o implementación (dado que tienen serias carencias para representar entidades que sólo tienen nivel de conocimiento)

de una misma entidad en función de los intereses del usuario[101]. En este último caso cada catálogo provee una única vista, mostrando al usuario la información en función de los intereses del publicador de la información. Otro problema es el grado de "vida" de los catálogos: en pocos casos se proveen mecanismos para fomentar la discusión y creación de comunidad para evolucionar los catálogos almacenados en el catálogo o generar conocimiento. Las capacidades de búsquedas de los catálogos suelen ser muy básicas e inclusive en muchos casos no están presentes.

- **Herramientas de modelado**: ninguna de las herramientas analizadas tiene un modelo abstracto y unificado para representar a los patrones a un alto nivel de abstracción incluyendo los niveles de conocimiento e implementación (como hemos propuesto al inicio de esta tesis). En todos los casos la descripción del nivel de conocimiento es incompleta (recordar que el nivel de conocimiento es el que permite realmente entender al patrón) y la representación del nivel de implementación no es abstracta y está asociada a una plataforma concreta (son necesarios tantos ficheros de implantación como lenguajes destino hayan).

Por lo tanto a modo de conclusión final podemos establecer que:

1. No existe un enfoque de descripción que permita describir los patrones a un alto nivel de abstracción incluyendo los niveles de conocimiento e implementación.

2. No existe un enfoque de descripción lo suficientemente flexible y extensible que permita describir patrones a un nivel de abstracción elevado utilizando cualquiera de las plantillas existentes.

3. No existe un enfoque de descripción que permita describir en forma uniforme patrones y conceptos de soporte

4. Los modelos de catalogación existentes con características avanzadas son muy complejos y requieren un gran esfuerzo de producción para obtener resultados. Por lo tanto ninguno permite obtener resultados rápidos con un grado de esfuerzo razonable.

5. El mantenimiento en los modelos de catalogación existentes con características avanzadas es complejo y laborioso

6. No existe un modelo de catalogación con características avanzadas que permita establecer relaciones semánticas complejas ad-hoc en forma emergente. Todos requieren trabajo de elaboración previa (desarrollo de una ontología, creación de modelos matemáticos, etc.)

[101] El usuario debe poder elegir en qué aspecto desea enfocarse al visualizar un patrón. El sistema de vistas dinámicas desarrollado en el prototipo y presentado posteriormente en los capítulos 14 y 16 permite resolver este problema en forma eficiente, dinámica y flexible.

7. No existe un modelo de catalogación con características avanzadas que permita al usuario abstraerse de las tecnologías y modelos que utiliza

8. Los catálogos públicos de patrones en la Web imponen al usuario modelos rígidos de navegación

9. Los catálogos públicos de patrones en la Web imponen al usuario modelos rígidos de visualización de entidades en función de los intereses del publicador de la información. El usuario no puede elegir en qué forma desea visualizar una entidad.

10. La mayoría de los catálogos públicos de patrones no incluyen infraestructura para la creación de comunidades o grupos de discusión para evolucionar sus contenidos. Por lo tanto no están "vivos" y no son "generadores de conocimiento"

11. Todos los catálogos públicos de patrones son contenedores pasivos de información: no exponen sus contenidos a través de APIs para que éstos puedan ser utilizados por otros catálogos o herramientas.

12. Las capacidades de búsqueda en los catálogos públicos de patrones en la Web suelen ser muy básicas, inclusive en muchos casos no están presentes

13. La mayoría de los catálogos públicos de patrones en la Web sólo muestran patrones, dejando de lado a los conceptos de soporte.

14. Los catálogos públicos de patrones en la Web exponen relaciones muy básicas entre sus elementos (relaciones sintácticas basadas en enlaces de hipertexto). En muchos casos las relaciones no están presentes.

15. Las herramientas de modelado no incluyen el nivel de conocimiento en forma adecuada cuando representan a los patrones (se especializan en la estructura y descripción general del comportamiento a nivel de interfaz)

16. Las herramientas de modelado no permiten describir el comportamiento del patrón a un alto nivel de abstracción, teniendo elementos descriptivos para cada implementación concreta objetivo.

A modo de conclusión final podemos afirmar que no existe un meta-modelo que sirva de guía para la descripción de patrones y conceptos de soporte a un alto nivel de abstracción, la gestión y compartición de de estas descripciones y cómo presentarlas a los usuarios finales a través de la Web o de interfaces de servicios débilmente acopladas.

Parte III
Solución Propuesta

"Si no conozco una cosa,
la investigaré"
Louis Pasteur

Capítulo 11
Arquitectura General de la Solución

"La única forma de conocer los límites
de lo posible es aventurarse un poco
más allá de ellos, en lo imposible"
Arthur C. Clarke

En el **capítulo 1** de esta tesis, se presenta el contexto (§ 1.1) y el problema general que intentamos resolver (§ 1.2). En ese mismo capítulo se establece un conjunto de objetivos generales (§ 1.4) que sirven como guía para delinear la estrategia de nuestra solución, la cual resumimos en los siguientes puntos:

- **Crear un lenguaje de meta-especificación** para representar a las entidades (patrones y conceptos) a un alto nivel de abstracción incluyendo los niveles de conocimiento e implementación.

- **Crear un catálogo** de entidades (patrones y conceptos asociados) descriptos con el lenguaje creado en el paso anterior.

- **Crear una herramienta de visualización** que permita navegar por el catálogo mencionado en el punto anterior.

En este capítulo se presenta una visión de alto nivel de la solución completa donde se propone una solución al problema planteado al inicio de esta tesis. Se expone una visión general a nivel arquitectónico que sirve como guía para comprender la solución propuesta y desarrolla en capítulos posteriores. Finalmente se presenta el modelo de gestión de conocimiento asociado al sistema propuesto, tomando como concepto central el ciclo de vida de los patrones [Welicki06b]. Al terminar de leer este capítulo el lector tendrá una idea general de la arquitectura y estrategia de solución propuesta en esta tesis.

❖ ❖ ❖

11.1 Contexto y Argumentación de la Solución

11.1.1 Estableciendo el Significado del Término "Patrón"

Como hemos podido comprobar anteriormente en el **capítulo 3**, es muy difícil establecer una definición taxativa del concepto "patrón" (§ 3.1). En las diversas obras de referencia existen diferentes definiciones con ligeras variaciones ([GoF95], [Alexander79], [POSA96], [Fowler02], [BFVY96]). A efectos de poder definir nuestro lenguaje de meta-definición de patrones, es condición necesaria establecer una definición clara y precisa de este concepto[102].

Por tanto, definimos a un patrón como "*una pieza de conocimiento que incluye información sobre un problema y su solución en un contexto, con sus ventajas e inconvenientes y toda la información necesaria para poder tener un buen entendimiento de los aspectos relacionados con él. Eventualmente puede contener información específica sobre su comportamiento, la cual permite generar artefactos de software*"[103] [Welicki05].

11.1.2 ¿Cómo Describir un Patrón?

Como hemos mencionado en el párrafo anterior, un patrón es una pieza de conocimiento que contiene una combinación de información descrita en prosa (contexto, motivación, problema, solución, etc.) y mediante métodos formales (código fuente y diagramas).

Por tanto, hemos dividido la descripción de un patrón en dos niveles [Welicki05]:

- **Nivel de Conocimiento**: incluye toda la información literaria, a saber, plantilla del patrón, metadatos para búsqueda, definición y responsabilidades de los participantes, relaciones con otras entidades y cualquier otro tipo de meta-información.

- **Nivel de Implementación**: incluye la información sobre el comportamiento del patrón, a saber, estructura y detalles de implementación, algoritmos, relaciones estáticas y dinámicas entre participantes. En muchos casos puede no estar presente (es opcional).

[102] Es necesario tener totalmente claro un concepto antes de pasar al nivel meta. Sería el equivalente a establecer cimientos fuertes antes de construir un edificio. Ralph Johnson hace un interesante análisis y reflexión sobre la necesidad de tener claros los conceptos antes de pasar al nivel meta y sobre los peligros de ir al nivel meta sin tener claro el nivel general en [C2Meta]

[103] "(a pattern is) a *piece of knowledge that includes information about a problem and its solution in a specific context, with the trade-offs and all the literary information needed to have a good understanding of the issues related with it. Eventually, it may contain behavior specific information, which will allow generating software artifacts for the proposed solution to the problem*" (Texto original de [Welicki05])

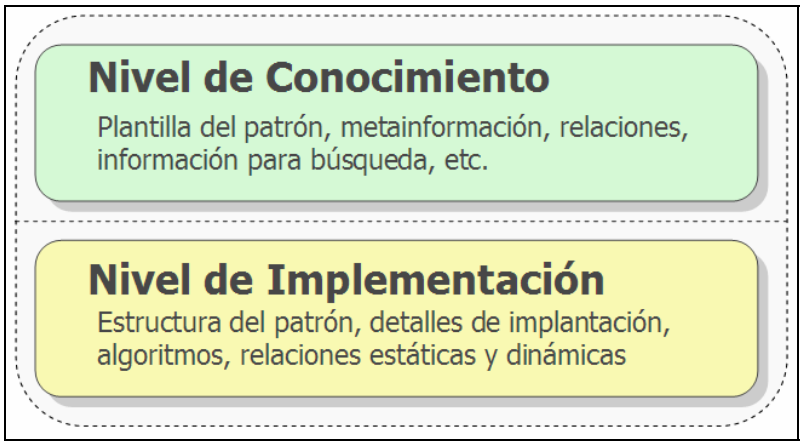


Figura 11.1 – Niveles de descripción de un patrón.

Es importante destacar que estos niveles no son monolíticos, sino que se componen a su vez de elementos más pequeños especializados en la descripción de un aspecto concreto (§ 12.2.3).

11.1.3 ¿Este Enfoque Sólo es Aplicable a los Patrones?

No, el enfoque propuesto puede ser aplicado a **cualquier concepto**. Por ejemplo, para describir el principio abstracto "*Dependency Inversion Principle*" (DIP) [Martin02], podríamos utilizar los dos niveles mencionados:

- **Nivel de Conocimiento**: incluye toda la información literaria que permite comprender el concepto, las relaciones con otros conceptos, metadatos para clasificación, etc. En el caso del DIP, incluiría su definición, etiquetas para búsquedas y relaciones con otras entidades.

- **Nivel de Implementación**: incluye un ejemplo donde se muestra como se pueden eliminar las dependencias con entidades concretas mediante el acoplamiento abstracto [Martin02].

Como hemos mencionado en la sección anterior, el nivel de implementación es opcional. Por tanto, para entidades que no lo tengan (como lenguajes de patrones, autores, etc.) es suficiente el nivel literario.

11.1.4 ¿Es Suficiente Describir Patrones para Transmitir Eficientemente el Conocimiento?

No, no es suficiente. Es deseable poder describir otros tipos de entidades que puedan aumentar la información que tenemos sobre el patrón y contribuir a una mejor compresión del concepto que el patrón quiere transmitir[104]. A modo de ejemplo, podemos citar entidades como los lenguajes de patrones y categorías en que un patrón está incluido, las refactorizaciones que pueden aplicarse, los

[104] Esto está alineado a la frase que dice que "un patrón es una solución a un problema en un contexto" [Hillside03]

conceptos de orientación a objetos en que se funda, los autores, las fuentes de información de donde ha sido obtenido (libros, artículos, Web.), etc. Toda esta información nos permite establecer un contexto claro del patrón, facilitando una mejor compresión del mismo.

A modo de ejemplo, podemos citar el caso del patrón ABSTRACT FACTORY [GoF95], respecto del cual podríamos decir que:

- Ha sido publicado en el libro "*Design Patterns: Elements of Reusable Object Oriented Software*" [GoF95]
- Esta incluido en la categoría "*Patrones Creacionales*" del sistemas de patrones del GoF [GoF95]
- Ha sido escrito por el GoF (Gamma, Helm, Johnson y Vlissides)
- Cumple con el principio "*Dependency Inversion Principle*" [Martin02]
- Es un patrón de diseño
- Es un patrón con ámbito de objeto [GoF95]
- Se refiere al desarrollo de software
- Se pueden aplicar las refactorizaciones "*Move Creation Knowledge to Factory*" [Kerievsky04] y "*Encapsulate Classes with Factory*" [Kerievsky04]
- Está relacionado con FACTORY METHOD [GoF95], SINGLETON [GoF95], PROTOTYPE [GoF95] y CREATOR [Larman99]

De esta forma podemos conocer la información esencial del patrón, sus orígenes, sobre qué principios de diseño se funda, cómo llegar a el, qué tipo de patrón es, etc.

11.1.5 Formalización del Dominio de Definición de Patrones

Los patrones no existen aislados de un contexto. De la misma forma en que una implementación no es suficiente para transmitir un patrón, un patrón puede no ser suficiente para transmitir el conocimiento que intenta expresar. Para subsanar esta situación hemos definido una ontología sencilla para formalizar el dominio de la definición de patrones.

¿Qué es una ontología para el propósito de nuestra investigación? Hemos utilizado la definición de Tomas Gruber que dice que "*una ontología es la especificación de una conceptualización*"[105][Gruber93] y que "*una ontología es una descripción (como la especificación forma de un programa) de los conceptos y relaciones que pueden existir para un agente o comunidad de agentes*" [106] [Gruber93]. Según estas definiciones el objetivo de la ontología es habilitar la compartición y reutilización de conocimiento [Gruber93], que es uno de los pilares sobre los que se funda nuestro modelo.

[105] "an ontology is a specification of a conceptualization" [Gruber93]
[106] "an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents" [Gruber93]

En la figura 11.2 a continuación se muestra la estructura de una ontología de ejemplo[107], donde el patrón es el concepto central y está relacionado con otros conceptos que a su vez están relacionados entre sí.

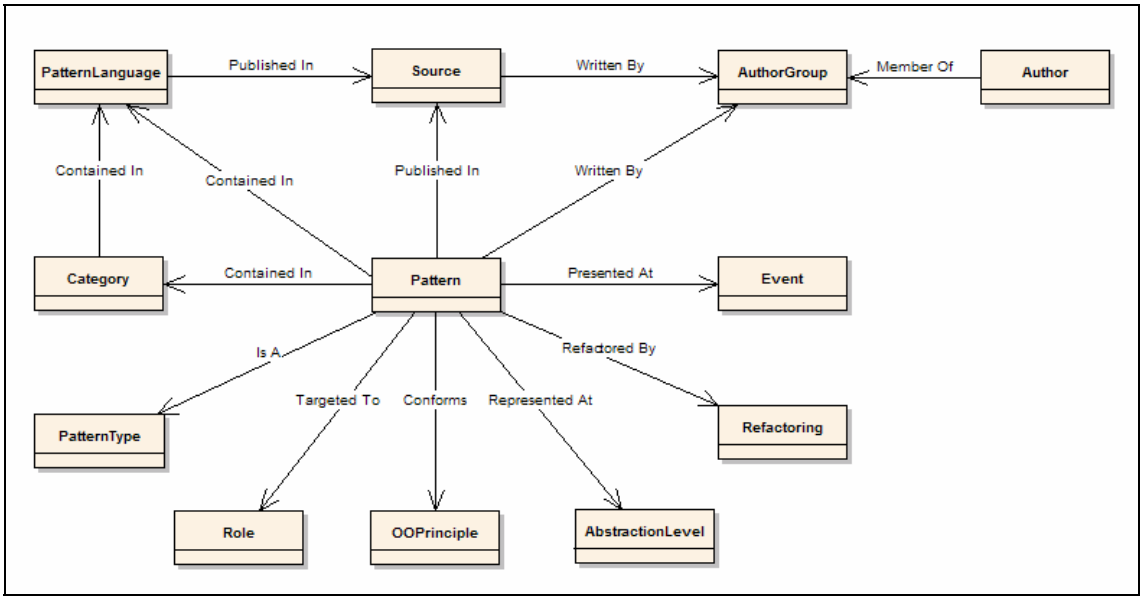


Figura 11.2 – Ontología sencilla para definir el dominio de la definición de patrones.

Es importante destacar que existen otras ontologías especializadas en la descripción de patrones como [WOP] y [W3CSWE] para conceptos de ingeniería del software. El problema que hemos encontrado con ellas es la rigidez[108]. En nuestro caso, la ontología no es determinante en el dominio y puede definirse y modificarse ad-hoc, dado que las relaciones se construyen dinámicamente a partir de EML-RDL, uno de los sub-lenguajes de EML (§ 12.2.4.2).

11.1.6 Ejemplo de Relaciones entre Entidades

Para explicar mejor el concepto, utilizaremos un ejemplo. En este caso, nos valdremos de los patrones creacionales del GoF [GoF95] y del patrón CREATOR de GRASP[109] (*General Responsibility Assignment Patterns*) [Larman99]. En la figura 11.3 a continuación se muestra como es posible relacionar entidades de distintos lenguajes de patrones. Adicionalmente, las entidades relacionadas no tienen que ser del mismo tipo: el patrón CREATOR (del lenguaje de patrones GRASP [Larman99]) está relacionado con la categoría de *Patrones Creacionales* del catálogo del GoF [GoF95]. A su vez, cada patrón creacional del catálogo del GoF está contenido en la categoría *Creacional* (el patrón tiene una relación "isContained" con la categoría).

[107] Esta ontología es la que hemos utilizado en el prototipo desarrollado en esta tesis. Existen otras ontologías, como por ejemplo [WOP] y [W3CSWE]
[108] Un análisis sobre este tema puede encontrarse en el capítulo 10 de esta tesis.
[109] GRASP es un lenguaje de patrones que se presenta en "Aplying UML and Patterns" [Larman99]

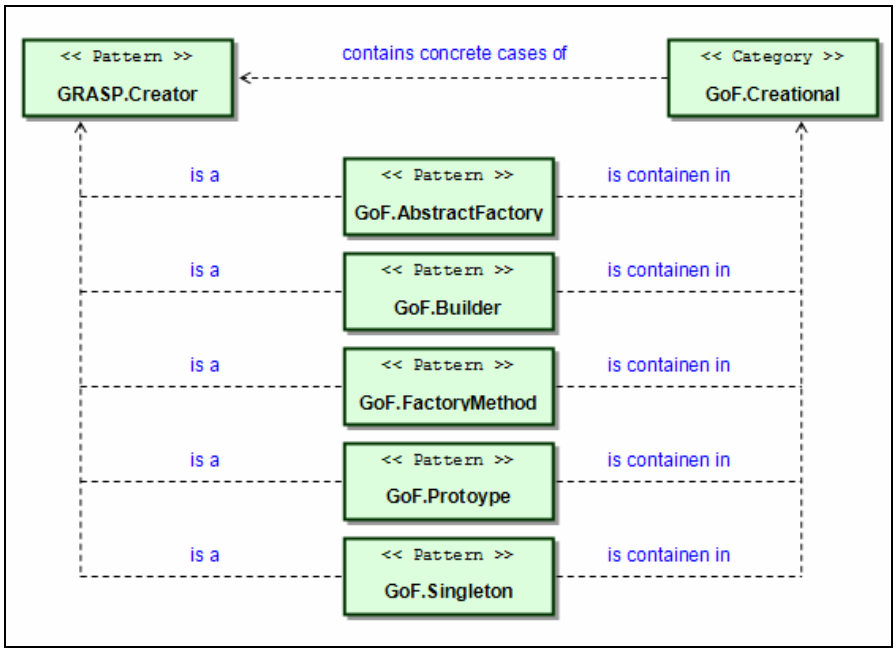


Figura 11.3 – Relación entre entidades EML.

11.1.7 Pattern Language Markup Language (PLML)

En la conferencia CHI 2003 [CHI2003], 15 participantes se reunieron en un workshop para discutir las perspectivas sobre los patrones de interacción persona ordenador[110] (HCI). Un resultado significativo de este workshop es la especificación de Pattern Lanague Markup Language (PLML). El objetivo de PLML [PLML] es traer orden a las muchas e inconsistentes formas de describir los patrones (plantillas) utilizadas por los autores de patrones [PLML03] (hemos analizado a las plantillas para descripción de patrones anteriormente en el **capítulo 4**).

PLML se basa en un DTD[111] [W3C] con una serie de elementos estándar que permiten describir los elementos más significativos de los patrones (soportando así un gran número de las plantillas existentes). En el siguiente bloque de código se muestra el DTD completo de la versión 1.1 de PLML (tomado de [PLML03]):

[110] Human Computer Interaction (HCI) en el texto original
[111] Document Type Definition

```
<!ELEMENT pattern (name?, alias*, illustration?, problem?, context?, forces?,
solution?,   synopsis?,   diagram?,   evidence?,   confidence?,   literature?,
implementation?, related-patterns?, pattern-link*, management?)>
<!ATTLIST pattern patternID CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT alias (#PCDATA)>
<!ELEMENT illustration ANY>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT context ANY>
<!ELEMENT forces ANY>
<!ELEMENT solution ANY>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT diagram ANY>
<!ELEMENT evidence (example*, rationale?)>
<!ELEMENT example ANY>
<!ELEMENT rationale ANY>
<!ELEMENT confidence (#PCDATA)>
<!ELEMENT literature ANY>
<!ELEMENT implementation ANY>
<!ELEMENT related-patterns ANY>
<!ELEMENT pattern-link EMPTY>
<!ATTLIST pattern-link
  type CDATA #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #REQUIRED
  label CDATA #REQUIRED>
<!ELEMENT  management  (author?,  credits?,  creation-date?,  last-modified?,
revision-number?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT creation-date (#PCDATA)>
<!ELEMENT last-modified (#PCDATA)>
<!ELEMENT revision-number (#PCDATA)>
```

Código 11.1 – DTD de PLML [PLML03]

Existe una versión extendida de PLML llamada PLMLx (Extended PLML) [PLMLX], que añade una serie de características al lenguaje. Si bien PLMLx añade funcionalidades a la especificación original, también restringe el espectro de patrones que pueden ser definidos utilizándolo (por ejemplo, excluye a todos los patrones de UI de Jennifer Tidwell [Tidwell] y añade una serie de campos mandatarios que pueden resultar inconvenientes cuando se definen el resto de los patrones [Fincher04].

Un aspecto importante a destacar sobre PLML y PLMLx es que han sido diseñados para describir patrones de HCI, con lo cual pueden tener limitaciones para describir otros tipos de patrones (esto se explica mejor en la siguiente sección).

11.1.8 ¿PLML es Suficiente para Describir a los Patrones?

No, no lo es. PLML ayuda a traer orden a la descripción de patrones, pero tiene varios inconvenientes y limitaciones, tales como tener un conjunto predefinido de elementos para describir el nivel de conocimiento del patrón y la falta de soporte para describir el nivel de implementación.

Puede existir una distancia considerable entre el DTD de PLML y la plantilla utilizada para describir ciertos patrones. Por ejemplo, hay varias secciones de los patrones de [GoF95] que no están incluidas en el DTD. Lo mismo sucede para los patrones de [POSA96] o para los de [Fowler02]. Describir estos patrones utilizando PLML implica un importante esfuerzo de traducción. Adicionalmente, PLML no incluye construcciones para describir el nivel de implementación a un nivel alto de abstracción.

Finalmente, PLML y PLMLx han sido creados para describir patrones. Por lo tanto, no pueden expresar conceptos de soporte y de ingeniería del software como principios de orientación a objetos, lenguajes de patrones, categorías de patrones, etc. [Welicki06]. Esto dificulta la creación de redes de entidades como la que se muestra en la figura 11.3 de la página anterior (§ 11.1.6).

11.1.9 ¿Qué Hace Falta para Describir Correctamente a las Entidades?

Es necesario crear un lenguaje que permita definir a los patrones y conceptos de soporte en forma estándar y uniforme, a un alto nivel de abstracción. La descripción no debe tener un fin específico más que describir el concepto. El modo de utilización debe ser determinado por el consumidor de la especificación. El objetivo del lenguaje es describir a las entidades en un nivel meta (recordar la discusión sobre el nivel meta en **capítulo 6**), convirtiéndose en un lenguaje de meta-especificación de entidades[112].

El lenguaje de meta-especificación debe proveer las construcciones sintácticas y semánticas adecuadas para representar los niveles de conocimiento e implementación en forma homogénea. La descripción del nivel de conocimiento debe soportar todas las plantillas de patrones (§ 4.1) existentes[113]. El nivel de implementación debe ser independiente de los lenguajes de programación y entornos de despliegue y el comportamiento de los participantes debe poder ser definido a un alto nivel de abstracción. Una vez obtenida la definición de un patrón o concepto, esta debe poder ser relacionada con cualquier otra y anotada con meta-datos para búsquedas y clasificación.

En el **capítulo 12** se presenta una lista completa y detallada de los requisitos para el lenguaje de meta-especificación de entidades (§ 12.1).

11.1.10 ¿Describir Correctamente las Entidades es Suficiente para Compartir Conocimiento?

No, la mera descripción de las entidades no garantiza que el conocimiento pueda ser compartido en forma eficiente, aunque es un gran paso hacia su descripción y

[112] Si bien puede describir cualquier entidad, el foco principal está establecido en los patrones de software.
[113] Hemos analizado las plantillas más utilizadas y populares en el capítulo 4.

formalización. Es necesario articular de alguna forma el conocimiento expresado con el lenguaje de meta-especificación para que éste pueda ser utilizado por otros.

Para este fin proponemos necesaria la creación de un catálogo[114] de entidades descritas con el lenguaje de meta-especificación mencionado en la sección anterior. El catálogo representa una base de conocimiento (donde existen términos y relaciones) que puede ser consumida por diversos clientes y evolucionar con el tiempo.

11.1.11 ¿Un Conjunto de Definiciones Forma un Catálogo?

No, un catálogo es un elemento más complejo que un conjunto de definiciones almacenadas en forma persistente[115]. El catálogo debe proveer la infraestructura necesaria para añadir, eliminar, modificar, vincular y recuperar patrones. Debe soportar la edición iterativa e incremental y tener la capacidad de trabajar con información incompleta (no es necesario tener toda la información de una entidad para poder introducirla en el catálogo, puede ser completada a posteriori). No debe limitarse a ser un repositorio pasivo, exponiendo sus contenidos (mediante un API de objetos distribuidos o servicios Web) permitiendo a los clientes utilizarlos en forma débilmente acoplada. Debe proveer también un motor de búsquedas que permita encontrar los elementos alojados en el catálogo.

En el **capítulo 13** se presenta una lista completa y detallada de los requisitos para el catálogo (§ 13.1).

11.1.12 ¿Es Suficiente el Catálogo para Exponer sus Contenidos?

No, no es suficiente. Si bien el catálogo tiene mecanismos activos para exponer sus contenidos, no tiene asociada ninguna estrategia de presentación. Por lo tanto, el catálogo es un proveedor de información, pero no tiene ningún tipo de responsabilidad sobre cómo se muestran los datos. Los clientes pueden consumir esos datos desde sus aplicaciones y hacer con ellos lo que consideren adecuado. Para que nuestro modelo de compartición de conocimiento sea completo, es necesario crear una herramienta de visualización, que permita a los usuarios interactuar con los contenidos del catálogo. Esta herramienta es el visor del catálogo, al que llamamos PatternsBrowser.

11.1.13 El Visor del Catálogo (PatternsBrowser)

La herramienta de visualización ***PatternsBrowser*** expone los contenidos del catálogo a través de una interfaz Web. Debe ser fácilmente accesible al público

[114] Las entidades de ese catálogo se convierten en "palabras" dentro de un "lenguaje" que es representado por la totalidad de elementos incluidos en el catálogo.
[115] Aún cuando sea un conjunto de especificaciones (que no es el caso), vale la pena traer la definición de sinergia que dice que "el todo es más que la suma de las partes"

general y no debe presentar problemas de distribución. Debe proveer una fachada sencilla para acceder a los contenidos del catálogo, facilitando a los usuarios la tarea de buscar, navegar y utilizar a los patrones y conceptos almacenados. Debe permitir al usuario elegir la vista que quiere aplicar sobre el contenido, permitiendo que este se centre en el aspecto que más le interese de la entidad. Debe incluir mecanismos de feedback y comunidades para convertirse en un "espacio vivo de información"[116].

En el **capítulo 14** se presenta ver una lista completa y detallada de los requisitos para la herramienta de visualización de catálogo (§ 14.1).

11.2 Estrategia General de Solución

En el **capítulo 1** de esta tesis, expusimos el contexto (§ 1.1) y el problema general que intentamos resolver (§ 1.2). En ese mismo capítulo hemos establecido un conjunto de objetivos generales (§ 1.4) que nos ha servido como guía para delinear la estrategia de nuestra solución y a partir de los cuales proponemos: [Welicki05] [Welicki06] [Welicki06b]:

- **Crear un lenguaje de meta-especificación** para representar a los patrones. Este lenguaje debe poder representar los niveles de descripción e implementación. La definición del nivel de conocimiento debe ser flexible, extensible y debe poder representar cualquier plantilla de patrones. La definición del nivel de implementación debe ser independiente del lenguaje y plataforma destino. Este lenguaje describe a los patrones a un nivel muy alto de abstracción, sin un propósito definido. El propósito de utilización puede ser establecido a posteriori por quien use la meta-definición y abarca fines tan amplios como visualización de la plantilla del patrón, generación de código, generación de diagramas, generación de grafos de relaciones, etc.

- **Crear un catálogo** de patrones (y conceptos asociados) descriptos con el lenguaje creado en el paso anterior. Diseñar una arquitectura funcional y técnica que permita gestionar en forma sencilla el catálogo y exponer sus contenidos. El catálogo no debe limitarse a ser un elemento pasivo, exponiendo sus contenidos en diversas formas explotables por sistemas de información y personas.

- **Crear una herramienta de visualización** que permita navegar por el catálogo mencionado en el punto anterior. Idealmente, esta herramienta debe estar basada en tecnologías Web, haciéndola accesible al público general a través de un navegador Web y sin la necesidad de ningún software adicional.

[116] Un problema recurrente que hemos encontrado en muchos catálogos es que "mueren" por falta de actividad. Creemos que dotar al visor de mecanismos de interacción permitirá mantener "vivo" al catálogo, convirtiéndolo en un espacio donde la comunidad puede compartir información y discutir sobre ésta.

Si bien estos objetivos son los que nos guiarán a resolver el problema de la descripción, utilización y compartición de patrones de software, estos mismos pasos pueden ser aplicados en otro dominio para crear una solución de gestión de conocimiento vertical. Podemos decir entonces que estamos ante un patrón de arquitectura [POSA96] [POSA00] [POSA03] que nos servirá como guía para la construcción de nuestra solución.

El conjunto de tecnologías y modelos propuestos permite acercar a los usuarios conocimientos de ingeniería del software que se encuentra disperso y distribuido en fuentes heterogéneas [Welicki06b], como se muestra en la figura 11.4 a continuación.

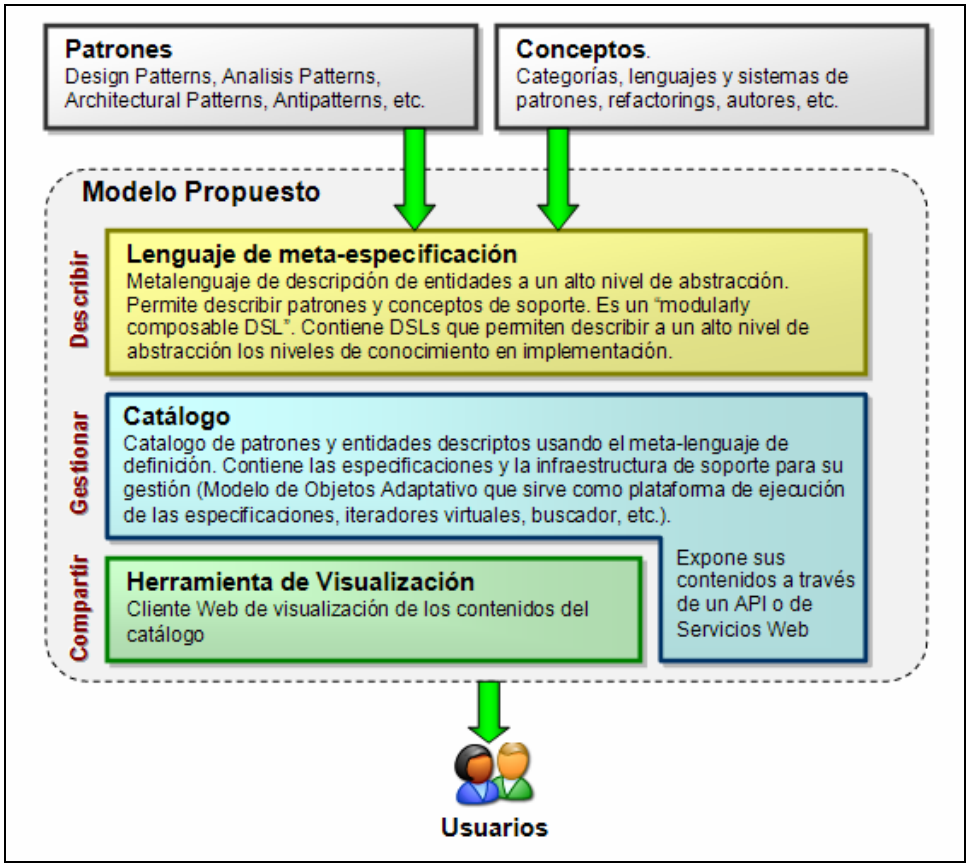


Figura 11.4 – Arquitectura general de la solución propuesta. Notar como el modelo en su totalidad acerca los patrones y los conceptos a los usuarios.

11.3 Solución Desarrollada

Los elementos que conforman la solución[117] propuesta son:

- **EML (Entity Meta-Specification Languaje):** EML es un lenguaje de meta-especificación de patrones (y conceptos de soporte) basado en XML. Permite definir a estos elementos en un único fichero auto-contenido. Incluye construcciones que permiten representar en forma abstracta los

[117] Estos elementos se derivan de la estrategia general de solución propuesta anteriormente en la sección 11.2.

niveles de descripción e implementación. Permite también describir relaciones entre entidades. El nivel de conocimiento incluye a todas las plantillas [C2PatternForm] existentes. El nivel de implementación permite describir el comportamiento del patrón en forma abstracta, independiente de los lenguajes de programación y entornos de ejecución. EML se presenta en detalle en el **capítulo 12**.

- **Catálogo de Patrones:** catálogo de patrones descriptos utilizando EML. Contiene también conceptos asociados a los patrones descriptos en EML (lenguajes de patrones, categorías, fuentes de información, autores, conceptos de orientación a objetos, etc.). Tiene facilidades para poder registrar, vincular, anotar y buscar a las entidades que están incluidas en él. Expone los contenidos del catálogo a través de un API local (FREP, presentado en § 11.3.2.1 y § 13.4.2) o de una interfaz de servicios Web. El catálogo se presenta en detalle en el **capítulo 13**.

- **Visor del Catalogo**: aplicación de visualización del catálogo. Permite navegar por el catálogo de patrones en diversas formas. Permite ver a un patrón o concepto de muchas formas, permitiendo al usuario concentrarse en un aspecto específico de cada entidad. Registra los elementos más populares y contiene foros de discusión multi-hilo asociados a cada concepto, a efectos de desarrollar el sentido de comunidad y dar el sentido de "espacio vivo de información". El Visor del Catálogo se presenta en detalle en el **capítulo 14**.

En los apartados restantes de esta sección profundizaremos sobre cada uno de los grandes bloques de construcción de nuestra solución, mencionados en la lista superior. Haremos especial énfasis en EML. Presentaremos también el modelo de objetos adaptativo (Adaptive Object Model [YJ03]) que sirve como plataforma de ejecución de EML. Como podremos observar, los lenguajes de dominio específico (Domain Specific Languages [Czarnecki00) y los modelos de objetos adaptativos son conceptos centrales en la implementación de nuestra solución.

11.3.1 El Lenguaje de Meta-Especificación (EML)

EML es el acrónimo de "Entity Meta-Specification Languaje". EML es un "modularly composable DSL" [Czarnecki00] basado en XML, creado con el objeto de describir todo tipo de patrones de software y conceptos asociados que permitan su compresión y contextualización. Las entidades descriptas con EML deben contener el nivel de conocimiento (información literaria) y pueden tener información sobre el comportamiento de los elementos que contienen (nivel de implementación).

EML provee la infraestructura necesaria para describir los niveles de descripción e implementación. Tiene también construcciones para describir relaciones entre patrones y conceptos y capacidades de anotación para clasificación y búsquedas rápidas. En su primera [Welicki04c] y segunda versión [Welicki05] sólo permitía

describir patrones definidos con plantillas previamente conocidas (era muy rígido respecto al nivel de conocimiento). En la versión actual [Welicki06] [Welicki06b] permite describir cualquier tipo de entidad utilizando plantillas dinámicas. Puede describir también relaciones entre entidades de los mismos o diferentes tipos (por ejemplo, "ABSTRACT FACTORY contiene a FACTORY METHOD" o "ABSTRACT FACTORY implementa el principio de inversión de dependencias [Martin02].

Para la descripción del nivel de implementación se ha creado un DSL que permite representar comportamiento de artefactos de código a un nivel alto de abstracción. En el **apéndice B** se puede encontrar la especificación completa de EML 1.0 (que incluye este DSL).

EML no fuerza la utilización de ningún tipo de plantilla particular [C2PatternForm] u elementos fijos para poder describir un patrón (que es un problema que hemos encontrado en PLML y PLMLx, en § 10.1.3.1). Las entidades se pueden definir con un mínimo de información y luego evolucionar.

Las entidades se clasifican a partir de sus relaciones. Estas pueden evolucionar con el tiempo, lo cual es una de las características más potentes de EML[118]. Un elemento puede variar su contexto con sólo modificar las entidades con las que se relaciona. De esta manera, para decir que "un patrón está incluido en un lenguaje de patrones" es necesario crear una relación de tipo "está incluido" entre el patrón y el lenguaje de patrones. Las relaciones se pueden crear, modificar y eliminar dinámicamente en cualquier momento.

La siguiente figura presenta una visión panorámica de alto nivel de EML. En ella se muestra al lenguaje general como un contenedor y a los sub-lenguajes que lo componen (recordar que EML es un "modularly composable DSL" [Welicki06]).

[118] Esta característica esta completamente alineada con la filosofía de la comunidad de patrones: los patrones evolucionan constantemente y periódicamente se publican versiones revisadas de los mismos patrones. De hecho, todos los patrones publicados en la serie POSA habían sido publicados previamente en las conferencias PLoP y EuroPLoP.

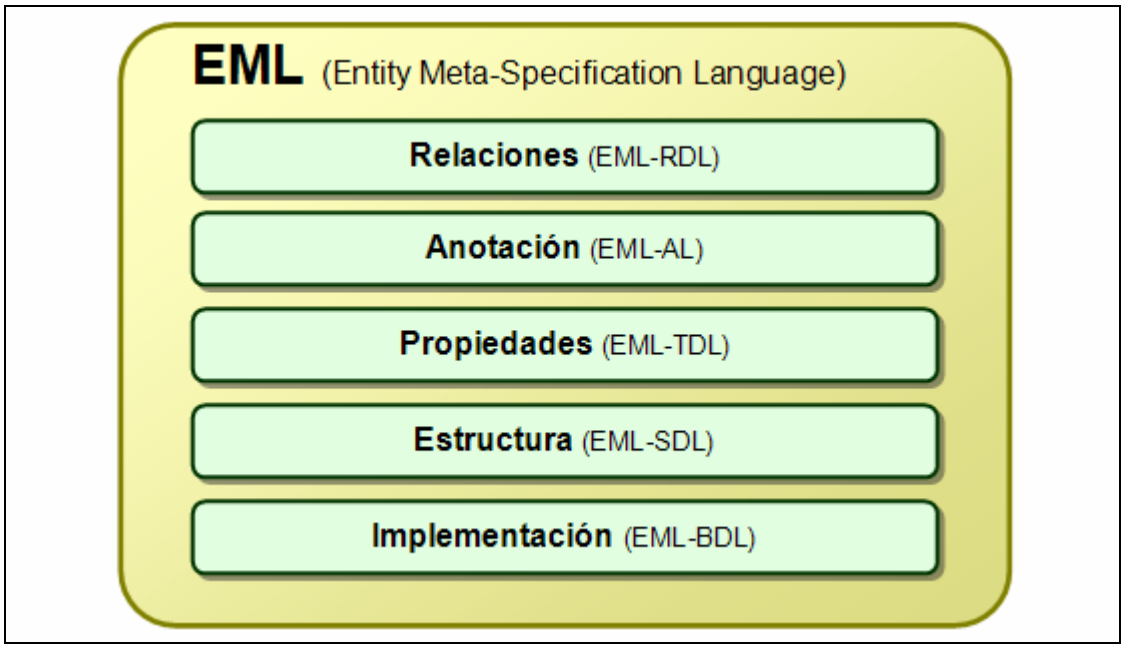


Figura 11.5 – Visión general de EML. En esta figura se pueden ver los elementos de alto nivel que conforman EML,

EML se presenta con mayor detalle en el **capítulo 12** y en el **apéndice B**.

11.3.2 Catálogo de Patrones y Conceptos

El catálogo de patrones es el elemento más complejo de la solución. Se compone de un conjunto de especificaciones de patrones y conceptos de soporte escritos utilizando EML, de la infraestructura necesaria para poder gestionar esas especificaciones y de una plataforma de ejecución para poder exponer esas especificaciones en forma de objetos o mensajes (sobre una arquitectura orientada a servicios)

El catálogo no se limita a un contenedor pasivo de información. Además de la gestión de la información provee la infraestructura necesaria para que esta sea explotada por diversos clientes en forma débilmente acoplada[119].

Todas las entidades del catálogo han sido definidas utilizando EML. Los componentes que conforman al catálogo se dividen en dos grandes grupos:

- **Componentes Pasivos**: conjunto de ficheros donde cada uno contiene la definición de una entidad en EML (cada fichero es totalmente auto-contenido). Contiene también la definición de los iteradores virtuales e imágenes asociadas a las entidades.

- **Componentes Activos**: infraestructura que permite exponer las entidades definidas en EML en forma de objetos (mediante un AOM) o mediante

[119] Loosely coupled (en inglés)

servicios Web. Infraestructura de registro para gestionar el alta, modificación, eliminación y vinculación de elementos en el catálogo.

El componente pasivo se construye mediante la combinación de una estructura de datos compleja (donde se registra la información sobre las entidades, sus metadatos, sus relaciones, etc.) y un conjunto de meta-especificaciones (las definiciones EML). Para introducir una nueva entidad EML en el sistema es necesario "registrarla", lo cual implica almacenar la entidad en todos los espacios de información que conforman al catálogo.

La siguiente figura presenta una visión panorámica de alto nivel de los principales elementos que conforman el catálogo.

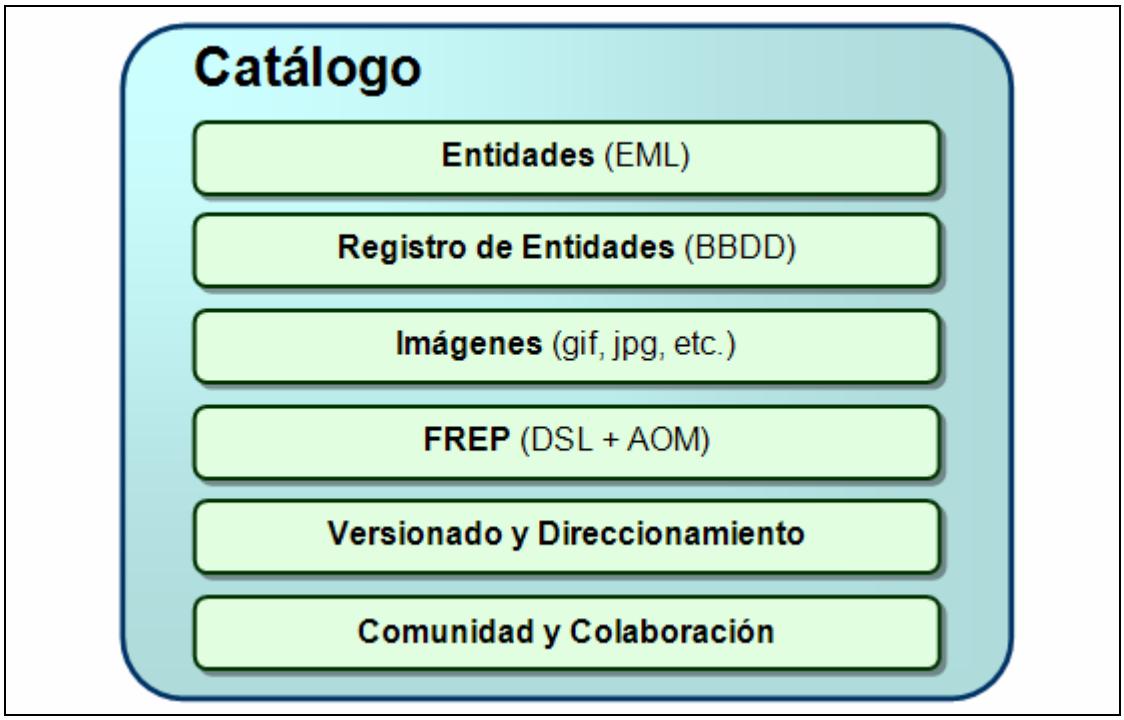


Figura 11.6 – Visión general del catálogo. En esta figura se pueden ver todos los elementos que conforman el catálogo (tanto elementos "activos" como "pasivos").

El catálogo se presenta con mayor detalle en el **capítulo 13** (en § 13.2.2 se muestra una visión gráfica de la arquitectura en forma más detallada). En el **capítulo 2** (§ 2.2) se presenta una breve reseña sobre la evolución de la infraestructura de catalogación en el transcurso de la investigación. La implementación del catálogo realizada en el prototipo se presenta en el **capítulo 16**.

11.3.2.1 DSL + AOM = FREP[120] (Plataforma Flexible de Tiempo de Ejecución)

Un lenguaje de dominio específico[121] (DSL) es un lenguaje especializado para un problema específico [Czarnecki00]. La idea básica de un DSL es un lenguaje de

[120] Flexible Runtime Execution Platform
[121] Domain Specific Language (DSL) en el texto original

ordenador orientado a resolver un tipo particular de problemas a diferencia de un lenguaje de propósito general[122] que es creado para resolver cualquier clase de problema [Fowler02]. Los DSLs pueden ser textuales o gráficos y pueden tener diferentes niveles de especialización. En general, son necesarios varios DSLs para especificar una aplicación completa. [Czarnecki00]. Para más detalles sobre DSLs, ver el **capítulo 7** de esta tesis.

Un modelo de objetos adaptativo[123] (AOM) es un sistema que representa clases, atributos, relaciones y comportamiento como metadatos. El sistema es un modelo basado en instancias en lugar de clases. Los usuarios cambian los metadatos (modelo de objetos) para reflejar cambios en el dominio. Estos cambios modifican el comportamiento del sistema. En otras palabras, el sistema almacena su modelo de objetos en una base de datos y lo interpreta [YJ03]. Esto brinda al modelo una gran adaptabilidad[124]. Para más detalles sobre AOMs, ver el **capítulo 8** de esta tesis.

Si bien hemos incluido este componente en el contexto del catálogo (§ 13.4.2), le dedicaremos una sección debido a su alta importancia en el modelo. Este es el componente más importante y complejo en la infraestructura técnica de la solución. Es el entorno virtual donde "viven" las descripciones de las entidades EML.

FREP combina DSLs y AOM para crear una solución flexible, extensible y que puede ser evolucionada en forma sencilla. Cada componente de la solución puede evolucionar en forma independiente[125], dado que el acoplamiento es mínimo y las reglas de composición de las instancias residen en metadatos. Los DSLs que conforman EML son analizados por un analizador que da como resultado una instancia ejecutable de un AOM que puede ser utilizada por los clientes del catálogo.

Nuestro AOM es una extensión del modelo original propuesto en [YJ03] y contiene tres niveles en lugar de los dos niveles tradicionales:

- Nivel de Conocimiento
- Nivel de Implementación
- Nivel de Visualización

FREP esta basado totalmente en metadatos, permitiendo modificar la mayor parte de sus componentes sin tocar código. Su mecanismo de extensibilidad esta basado en una arquitectura de plug-ins [Fowler02] extendida, fundada en los principios de "*Dependency Inversion Principle*" [Martin02] y "*Open-Closed Principle*" [Martin02] [JCJO92]. De esta forma, podemos añadir en forma sencilla elementos al sistema implementando interfaces y registrando componentes o modificar algún aspecto

[122] General Purpose Language en el texto original
[123] Adaptive Object Model (AOM) en el texto original
[124] En [YJ03] se compara a los AOM con máquinas virtuales UML (UML Virtual Machines)
[125] Esta independencia tiene sus límites. Por ejemplo, si se modifica un DSL y esta modificación afecta al analizador, se deberá modificar la pieza de código que realice el análisis de esa sección del DSL para poder introducir los cambios en el sistema.

concreto del sistema modificando metadatos de configuración. En la figura 11.7 a continuación se presenta un diagrama de arquitectura general del modelo.

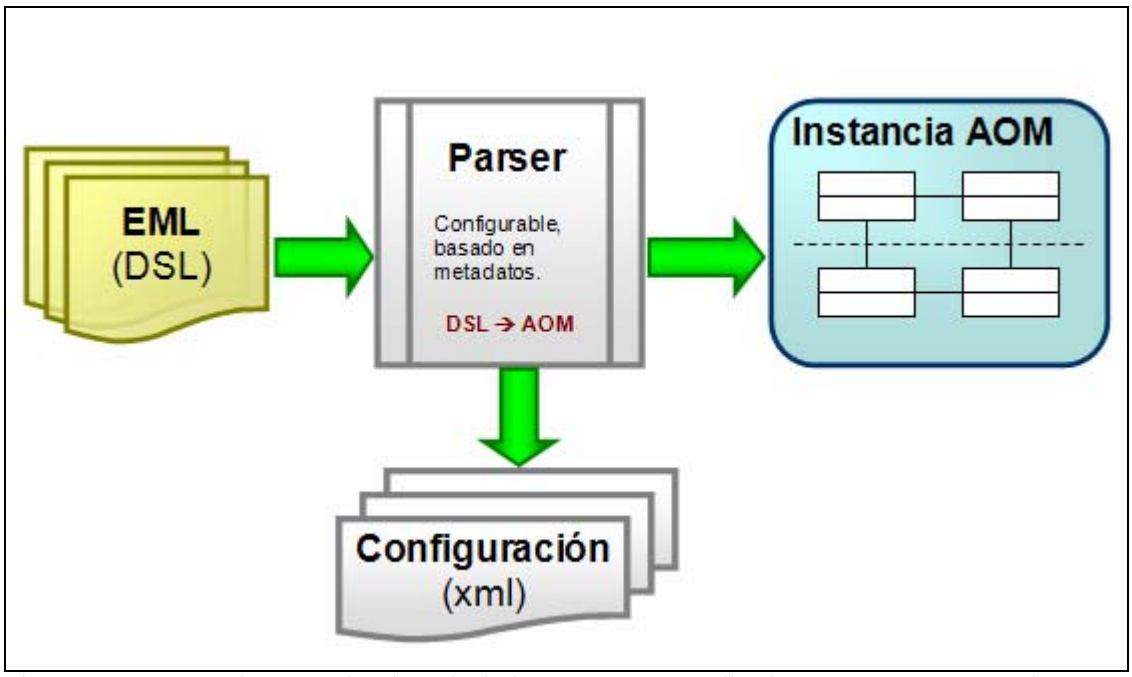


Figura 11.7 – Arquitectura de alto nivel de FREP. En analizador toma como entradas meta-especificaciones EML y a partir de un proceso basado en metadatos de configuración retorna una instancia de un modelo de objetos adaptativo como salida.

Una vez realizado el análisis del DSL y obtenido el AOM, este puede ser utilizado en la forma que el usuario considere apropiada. FREP se presenta en mayor detalle en el **capítulo 13** (§ 13.4.2).

11.3.2.2 ¿Por qué DSL y AOM?

Dada la complejidad de ambos conceptos, el lector en este punto puede preguntarse porque hemos decidido combinarlos y si realmente vale la pena. La combinación de estos dos conceptos permite crear una plataforma ágil, flexible y robusta muy fácil de evolucionar [Welicki04]. Cada una de estas tecnologías permite gestionar en forma óptima su ámbito de incumbencia:

- Los DSLs nos permiten describir en forma precisa los elementos, utilizando lenguajes especializados en describir un dominio específico y acotado.

- Los AOM nos permiten crear un modelo de objetos dinámico que varia en función de metadatos, permitiendo que la plataforma evolucione sin modificar su código fuente y evolucionar la plataforma añadiendo nuevos metadatos o nuevas piezas de código con responsabilidades bien definidas.

Por lo tanto, consideramos que hemos hecho una buena elección porque…

- Utilizamos la expresividad de los DSLs para describir a las entidades (a partir de descripciones puntuales de las partes que las componen).

- Utilizamos la flexibilidad, agilidad y potencia de AOM para ofrecer una plataforma de ejecución para las entidades.

Esta combinación permite resolver un problema complejo que no ha sido resuelto previamente por ningún otro enfoque existente (ver **capítulo 10**): dar soporte a todas las plantillas existentes para describir patrones e inclusive permitir crear nuevas plantillas dinámicamente[126] manteniendo una estructura formal.

11.3.3 Visualizador de Patones (Patterns Browser)

El visualizador de patrones (Patterns Browser) es una aplicación Web que permite navegar por los contenidos del catalogo. Su objetivo es proveer un cliente sencillo de visualización sobre los contenidos del catálogo mencionado anteriormente.

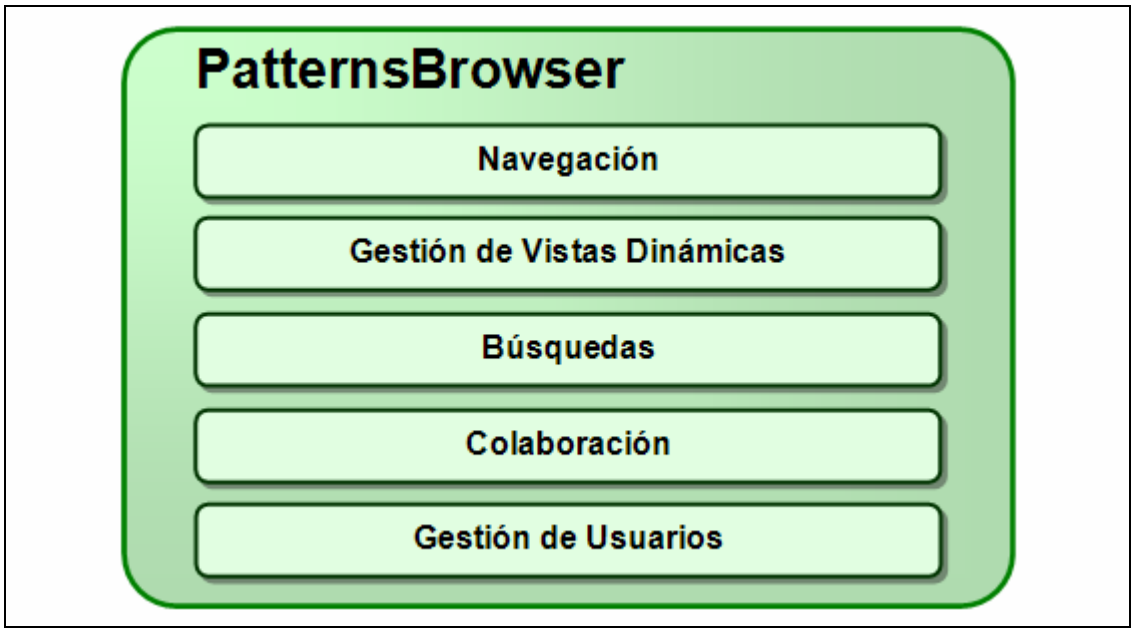


Figura 11.8 – Visión general de los componentes que conforman el visor del catálogo.

PatternsBrowser permite navegar por las entidades registradas en el sistema. Para cada una hay varias vistas disponibles, las cuales se vinculan dinámicamente mediante metadatos. Por lo tanto, tomando una entidad como el "modelo" en una arquitectura tipo MVC [POSA96] se le pueden asignar múltiples vistas, como se muestra en la figura 11.8. Por lo tanto, estamos ante una extensión al MVC original, implementada sobre un modelo de plug-ins [Fowler02] basado en reflectividad [Ortin02] [POSA96]. Las vistas se vinculan dinámicamente a la entidad en tiempo de ejecución, permitiendo cambiar el estilo de visualización a partir de criterios arbitrarios.

[126] Estas nuevas plantillas se pueden crear en forma arbitraria. Por ejemplo, en el prototipo hemos creado una plantilla especial para describir patrones del GoF que incluye contenidos de otras fuentes y elementos de otras plantillas, con el objeto de aumentar la información sobre el patrón y hacer que sea más fácil de comprender.

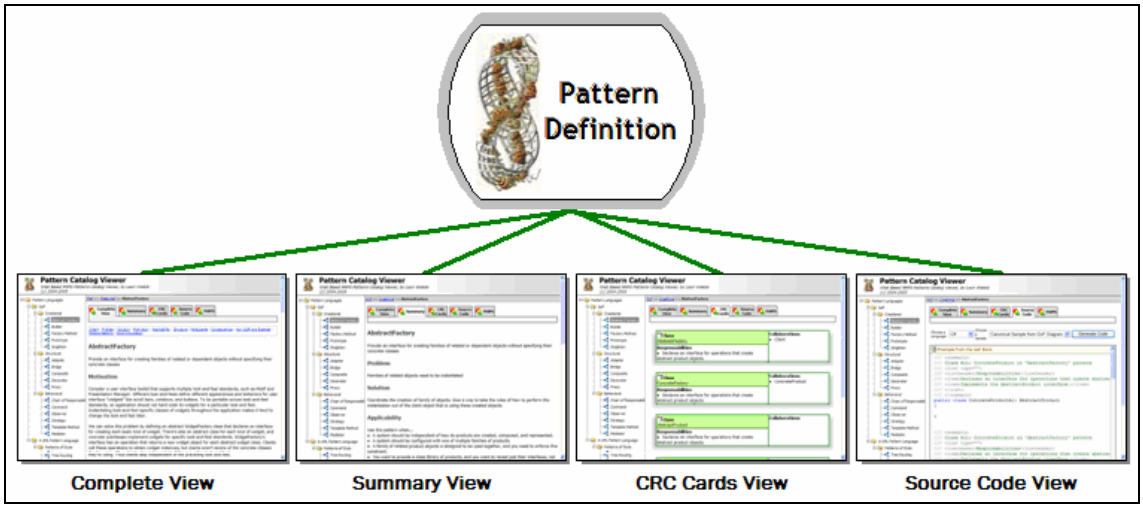


Figura 11.9 – Múltiples vistas sobre un patrón. Las vistas se vinculan a las entidades dinámicamente, en tiempo de ejecución. La posibilidad de contar con varias vistas permite al usuario enfocarse en el aspecto que considera adecuado.

El visor del catálogo se presenta con mayor detalle en el **capítulo 14**. La implementación del visor del catálogo realizada en el prototipo se presenta en el **capítulo 16**.

11.4 Creación de un Prototipo

A efectos de verificar empíricamente la factibilidad del modelo teórico propuesto hemos creado un prototipo de investigación que implementa todas las funciones mencionadas en este capítulo y a lo largo de esta disertación. El prototipo se compone de los siguientes componentes:

- Una especificación completa de EML (especificación utilizable de todos sus DSLs).
- Una implementación del catálogo, incluyendo…
  - Un catálogo de patrones y conceptos de soporte creados utilizando la especificación mencionada en el punto anterior.
  - Una implementación completa de la infraestructura de catalogación (incluyendo FREP).
- Una implementación de la interfaz de servicios Web.
- Una herramienta Web de visualización del catálogo.

El prototipo se describe en detalle en los **capítulos 15** y **16**.

11.5 Gestión del Conocimiento

Anteriormente (§ 3.1.7) hemos dicho que los patrones son un gran mecanismo de comunicación para transmitir la experiencia de los ingenieros y diseñadores experimentados a los más noveles [GoF95], convirtiéndose en unas de las vías para

la gestión del conocimiento [Welicki06b] [Welicki05d]. En el caso de nuestro modelo, no sólo se asigna un significado a un concepto, sino que ese significado puede ser serializado, transmitido a través de un cable y compartido en diversos contextos, convirtiéndose en una vía para la gestión de conocimiento empírico (el empirismo esta dado por la naturaleza intrínseca de los patrones).

Los patrones permiten establecer un vocabulario común de diseño [GoF95], cambiando el nivel de abstracción a colaboraciones entre entidades a un mayor nivel de abstracción y permitiendo comunicar experiencia sobre dichos problemas y soluciones [Welicki05d]. Nuestro modelo permite dar soporte a ese vocabulario, actuando como un "tesauro" que de soporte a un proceso de construcción de software, gestión de conocimiento o gestión de experiencias basadas en patrones.

La tecnología es sólo un facilitador[127] y no la solución en sí [Joyanes03]. En nuestro caso esta afirmación se cumple en su máxima expresión, dado que la mayor complejidad en la solución general es establecer el lenguaje de descripción y crear el catálogo (tareas que no requieren tecnología). La creación de nuestro entorno de ejecución FREP ha facilitado aún mas la inclusión de nuevos elementos (si bien la construcción de FREP ha sido compleja), relegando a los demás aspectos tecnológicos a un 15% de la solución total[128].

Alfons Cornella, de Infonomía dice en [Cornella03] que "*sin espacio social, no existe el espacio digital*"[129]. Es por esto que junto con la infraestructura de compartición de conocimiento formal hemos habilitado un espacio de colaboración para que los usuarios puedan fortalecer el sentido de comunidad y mantener "vivo" al catalogo. En la primer versión de nuestro modelo hemos incluido foros de discusión multi-hilo asociados a cada entidad EML, a efectos de favorecer el intercambio de información y la interacción entre los usuarios del sistema, que pertenecen a grupos muy diversos probablemente divididos geográficamente[130].

11.5.1 Los Patrones y la Gestión del Conocimiento

Un patrón es una forma explícita de conocimiento de ingeniería del software [Welicki06b]. Compartir patrones es compartir conocimiento [Welicki06]. Los patrones son una vía ideal para compartir experiencia en el ámbito de la ingeniería del software. Ayudan a aumentar (y mejorar) el vocabulario de sus usuarios, permitiendo aumentar el nivel abstracción en las discusiones en un contexto [Welicki05d] (por ejemplo, los patrones de diseño [GoF95] cambian el nivel de abstracción en las discusiones de diseño y arquitectura de software de clases a colaboraciones entre clases [GoF95]). Existe un punto de convergencia muy

[127] Enabler (en inglés)

[128] De todos modos, es importante aclarar que la construcción de FREP ha sido una tarea compleja en los niveles de análisis, diseño e implementación. La creación de FREP supuso un desafió técnico importante, pero una vez creada nos ha permitido abstraer muchos factores técnicos.

[129] Siguiendo con esta tesis, se recomienda el artículo "*La infoestructura: Un concepto esencial en la sociedad de la información*" [Cornella98], del mismo autor donde profundiza en esta aserción

[130] Esta es una característica intrínseca de las aplicaciones Web [Cueva03] y que hereda nuestro front-end de visualización por ser una de ellas.

interesante entre la gestión del conocimiento aplicada y los patrones, que puede ser expresada con esta simple analogía: en "*La Organización Creadora de Conocimiento*" [NT94] se presenta un modelo para crear valor a partir de la generación de conocimiento, utilizado en las compañías japonesas. En esa misma obra se demuestra como varias compañías japonesas de renombre (Toyota, Matsushita Corp., etc.) han obtenido resultados exitosos a partir de la utilización del modelo de generación de conocimiento propuesto en esa misma obra. En "*Patrones de Diseño*" [GoF95], se crea una herramienta de edición de textos con características avanzadas (el editor de textos *Lexi*) a partir de la composición de los patrones presentados en esa misma obra. En ambos casos se está obteniendo valor y resolviendo problemas no triviales a partir de la correcta gestión y utilización del conocimiento. [Welicki06b]

11.5.2 Si Supiéramos lo que Sabemos…

Existe una frase muy famosa en la comunidad de gestión del conocimiento, atribuida a Lewis Platt, un ex presidente de Hewlett-Packard: "*Si HP supiera lo que HP sabe, seria tres veces más rentable*" [DP99]. Los patrones ayudan a codificar y formalizar conocimiento, haciendo posible su compartición dentro de una organización o en la industria en su totalidad. Los patrones no se limitan a diseño orientado a objetos o algoritmos [Welicki06b]: existen en una amplia variedad de dominios (a modo de ejemplo podemos citar a [MR04], donde se presenta un lenguaje de patrones para introducir nuevas ideas en las organizaciones).

Los patrones pueden ayudar a una organización a codificar su conocimiento tácito en formas concretas de conocimiento explícito que puede ser compartido y reutilizado a distintos niveles (intra o inter-organizacional) [Welicki06b]. Los lenguajes de patrones pueden hacer lo mismo pero a una escala mayor, estableciendo un marco de trabajo[131] para organizar y clasificar grupos de patrones [Welicki06b].

11.5.3 Los Patrones en el Modelo de Nonaka y Takeuchi

En la obra "*La Organización Creadora de Conocimiento*", Nonaka y Takeuchi presentan un modelo donde se crea valor a partir de la correcta gestión del conocimiento en las organizaciones. En el modelo de creación de conocimiento presentado la espiral de conocimiento [NT94] se da en dos dimensiones: la epistemológica y la ontológica.

Anteriormente hemos postulado que los patrones son una forma empírica de gestión del conocimiento. Fundamos esta aserción en la ubicación que podemos dar a los patrones dentro de las dimensiones de creación conocimiento (§ 5.1.1.2) establecidas en [NT94]. En la figura 11.9 a continuación podemos observar que los patrones son conocimiento explícito que se comparte a nivel interorganizacional [Welicki06b].

[131] Framework (en inglés)

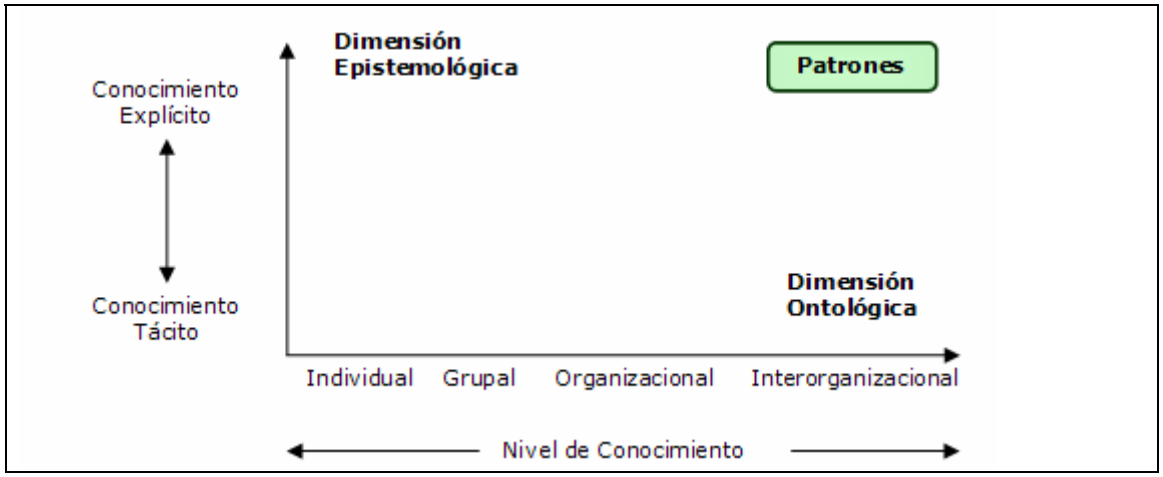


Figura 11.10 – Los patrones en el modelo de conocimiento de Nonaka y Takeuchi.

Los patrones son conocimiento explícito que se formaliza y describe mediante plantillas (**capítulo 4**) y que se comparte a través de su publicación en libros ([GoF95], [POSA96], [Fowler02], [Fowler96], [Larman99], etc.), conferencias (por ejemplo, todas las variantes de las conferencias PLoP [Hillside]) y mediante catálogos públicos en la Web (en el **capítulo 10** se estudian 12 catálogos públicos de patrones).

11.5.4 El "Ciclo de Vida" de un Patrón

Los patrones emergen de la experiencia [Welicki06b]. Tienen un ciclo de vida que comienza con conocimiento tácito en la cabeza de una persona o grupo de individuos y termina con una descripción explícita de ese conocimiento que puede ser compartida. Es muy importante destacar la importancia de este proceso, dado que el conocimiento tácito reside en la cabeza de un grupo de individuos y si no se formaliza de alguna forma puede perderse cuando éstos dejan la organización [Welicki06b] [Welicki03b]. En la figura 11.10 a continuación se presenta el ciclo de vida del patrón.

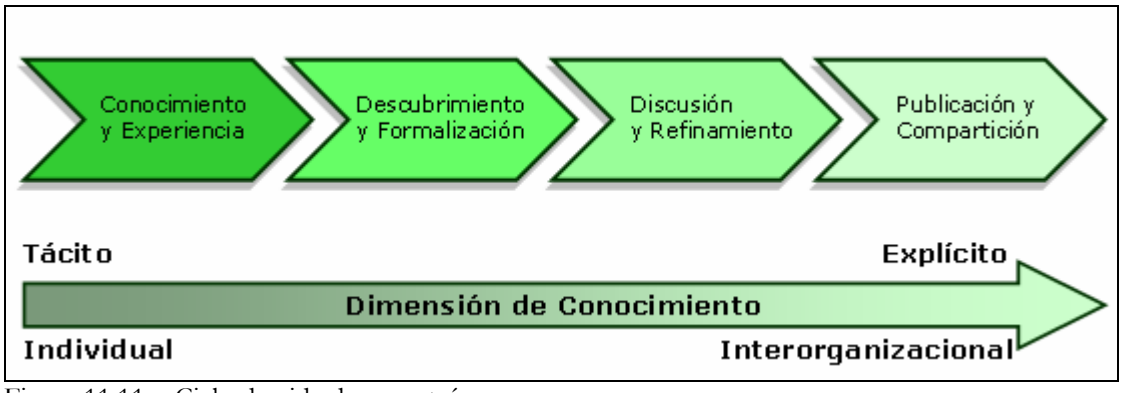


Figura 11.11 – Ciclo de vida de un patrón.

El ciclo de vida de un patrón tiene las siguientes etapas [Welicki06b]:

- ***Conocimiento y Experiencia***: el patrón es sólo conocimiento tácito residente en la cabeza de un individuo o grupo, proveniente de la experiencia.

- ***Descubrimiento y Formalización***: el patrón comienza a emerger (a partir de ideas y prácticas existentes o mediante "minería de patrones"[132] en sistemas existentes). Los individuos involucrados inician su documentación (utilizando alguna plantilla). El producto final de esta etapa es una descripción del patrón pendiente de revisión[133].

- ***Discusión y Refinamiento***: los patrones se discuten en la comunidad. Las conferencias PLoP[134] [Hillside] (en todas sus variantes) son el mejor foro para discutir y refinar patrones y lenguajes de patrones. El producto final de esta etapa es una versión refinada del patrón a partir de su análisis y discusión en la comunidad.

- ***Publicación y Compartición***: el patrón refinado se publica y comparte en la comunidad de software (los medios de publicación pueden ser libros, revistas[135], actas de conferencias, Web, etc.). El producto de esta etapa es una versión final del patrón que puede ser compartida.

11.6 Reflexión: El Nuevo Ciclo de Conocimiento de los Patrones

La creación de un lenguaje de meta-especificación para descripción de patrones, una infraestructura de catalogación y un catálogo de patrones y conceptos (incluyendo las relaciones existentes entre ellos) descriptos con ese lenguaje es un paso claro hacia recoger, gestionar y compartir conocimiento sobre ingeniería del software [Welicki06b]. La creación de una herramienta de visualización pone este conocimiento a disposición del mundo.

El modelo propuesto en esta tesis produce un cambio en cada uno de los pasos del ciclo de vida de los patrones. El lenguaje de meta-especificación permite a los autores formalizar su conocimiento tácito en una forma estándar que puede ser compartida mediante una infraestructura de compartición de conocimiento. El catálogo les permite organizar y clasificar sus patrones, relacionarlos con otras entidades y compartirlos con otros usuarios. El visor hace accesible al mundo en forma sencilla todo este conocimiento y provee funcionalidades de comunidad que dan soporte a la discusión y trabajo colaborativo sobre un patrón para refinarlo y/o evolucionarlo. [Welicki06b]

[132] Pattern Minning (en inglés)
[133] La revisión se refiere a la discusión y validación por parte de la comunidad de patrones y en menor medida al proceso de mejora del patron (shepherding [Hillside]) previo a la publicación del patrón para su discusión.
[134] PLoP, EuroPLoP, VikingPLoP, ChilliPLoP y SugarLoafPLoP.
[135] Journals (en inglés)

Todo esto cambia el ciclo de vida, haciéndolo más dinámico e interactivo: una vez que se crea una descripción, esta puede ser introducida en el catálogo y compartida con el mundo entero. La discusión y refinamiento no es más un paso en el medio del ciclo de vida, dado que la herramienta de visualización provee mecanismos para discutir en forma interactiva sobre cualquier entidad del catálogo. La publicación tampoco es un paso estático al final del ciclo: una vez que la meta-especificación existe y se introduce en el catálogo se "publica", estando disponible para todos los usuarios. Los autores pueden refinar sus patrones utilizando como entrada los comentarios y las discusiones sobre las entidades, producidos por usuarios del mundo entero. [Welicki06b]

Por lo tanto el nuevo ciclo, que se muestra en la figura 11.11, es más dinámico e interactivo, soportando la evolución y refinamiento constante de los patrones [Welicki06b].

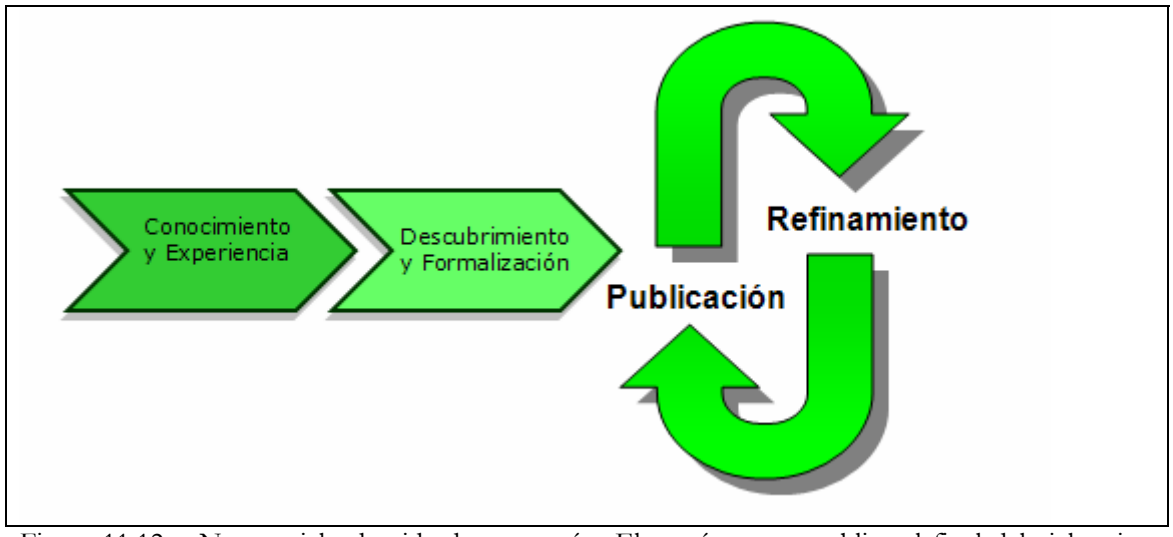


Figura 11.12 – Nuevo ciclo de vida de un patrón. El patrón no se publica al final del ciclo, sino apenas tengamos una versión inicial. La discusión y refinamiento se dan constantemente, realimentando y evolucionando constantemente a la versión publicada del patrón.

Capítulo 12
EML - El Lenguaje de Meta-Especificación de Patrones

"No hay peldaño que nos eleve más,
ni que sea más trascendental en
la historia de la evolución de la mente
que la invención del lenguaje"
Daniel Dennet

En el capítulo anterior hemos realizado un análisis y argumentación del contexto de la solución (§ 11.1) al problema planteado al inicio de esta tesis (§ 1.2). En esa misma sección hemos establecido en forma inductiva la necesidad para la creación de un lenguaje de meta-especificación y una serie de requisitos generales para éste[136]. Más adelante, presentamos brevemente a EML como solución para la descripción de entidades a un alto nivel de abstracción, a un nivel meta[137].

En este capítulo presentaremos EML (*Entity Meta-specification Language*), el lenguaje de meta-especificación de entidades (patrones y conceptos de soporte) creado en esta tesis. Este lenguaje es un "*high-level modularly composable DSL*" [Czarnecki00] textual basado en XML cuyo objetivo principal es describir entidades, con especial énfasis en la descripción de patrones y conceptos de ingeniería del software.

En la primera sección del capítulo se establecen los requisitos detallados para el lenguaje de meta-especificación. A continuación se describe EML, que es el lenguaje que hemos creado en esta tesis y que satisface esos requisitos. Finalmente, a modo de conclusión revisaremos los requisitos establecidos al inicio, contrastándolos con todo lo expuesto a lo largo del capítulo.

❖ ❖ ❖

[136] En ese mismo capítulo, prometimos ampliar esos requisitos generales. Los requisitos ampliados se presentan en este capítulo.
[137] El nivel Meta es explicado en el capítulo 6 de esta tesis.

12.1 Requisitos del Lenguaje de Meta-Especificación

En el **capítulo 11** presentamos en forma resumida y coloquial los requisitos para la definición de un lenguaje de meta-especificación de patrones y conceptos de ingeniería del software (§ 11.1.9), que se establecen a partir de los problemas recurrentes detectados en los enfoques existentes en el **capítulo 10** (§ 10.4). A continuación se refinan esos requisitos y se presentan en forma detallada en la siguiente lista:

RL-1 **Alto nivel de abstracción**: la descripción de las entidades (patrones y conceptos de soporte) debe realizarse a un alto nivel de abstracción. Esta descripción debe ser independiente de cualquier pauta de utilización.

RL-2 **Soportar todas las plantillas de patrones**: poder describir patrones independientemente de la plantilla que utilicen (§ 4.1). Soportar todas las plantillas existentes de patrones y permitir crear nuevas dinámicamente en función de criterios arbitrarios (por ejemplo, una plantilla que contenga componentes de distintas plantillas[138]).

RL-3 **Descripción de relaciones:** debe tener la posibilidad de incluir relaciones entre entidades en forma sencilla. Las entidades relacionadas (los extremos de las relaciones) deben poder ser de diferentes tipos.

RL-4 **Descripción del comportamiento independiente de plataforma:** la descripción del comportamiento de las entidades involucradas en la solución debe ser totalmente independiente de los lenguajes y entornos de programación.

RL-5 **Anotación mediante etiquetas:** debe proveer la infraestructura necesaria para anotar a las entidades con etiquetas[139] [Tag] que pueden ser utilizadas para búsquedas y clasificación.

RL-6 **Describir entidades con y sin implementación:** representar en forma indistinta patrones que tengan los niveles de conocimiento e implementación (por ejemplo los de [GoF95] y [POSA96]) y los que sólo tengan el de conocimiento (por ejemplo los de [Evitts00] y [Fowler96]).

RL-7 **Reaprovechar el conocimiento existente:** debe tener la posibilidad de agregar apuntadores a sitios donde pueda conseguirse mas información sobre un patrón, aprovechando otras fuentes de conocimiento existentes.

[138] En el prototipo (presentado en los capítulos 15 y 16) hemos creado una plantilla extendida para los patrones del GoF [GoF95] que incluye propiedades adicionales como ejemplos problema, solución y ejemplo en el mundo real (no software).
[139] Tags (en inglés)

RL-8 **Metadatos para búsquedas y clasificación:** deebe tener metadatos para búsqueda y clasificación. Estos metadatos deben poder describirse en forma independiente del contenido de la entidad, permitiendo que ésta pueda ser encontrada por un buscador semántico o que participe en esquemas de clasificación emergente (entre otros usos).

RL-9 **Facilidad de producción:** debe ser fácil escribir meta-especificaciones de entidades con este lenguaje. Las meta-especificaciones deben poder crearse, editarse y modificarse con cualquier editor de textos[140].

RL-10 **Uniformidad:** utilizar un lenguaje uniforme para describir a los niveles de conocimiento e implementación[141].

12.2 El Lenguaje EML

EML es el acrónimo de *Entity Meta-specification Language*. EML es un lenguaje de dominio específico textual, modularmente componible[142] [Czarnecki00], basado en XML, creado con el objetivo de describir todo tipo de patrones de software y conceptos de soporte (lenguajes de patrones, categorías, refactorizaciones, autores, principios de orientación a objetos, etc.).

EML provee la infraestructura necesaria para describir los niveles de conocimiento e implementación junto con las relaciones entre patrones y conceptos de soporte. En sus primeras versiones [Welicki04c] [Welicki05] sólo estaba preparado para describir patrones utilizando plantillas fijas estáticas previamente definidas (basadas en elementos estandarizados[143]). En la versión actual [Welicki06] [Welicki06b] (que es la que se presenta en esta tesis) puede describir cualquier tipo de entidad utilizando plantillas dinámicas. También puede describir relaciones entre entidades del mismo o diferentes tipos (por ejemplo, puede expresar que "*un patrón está contenido en un lenguaje de patrones*", "*un patrón conforma un principio de orientación a objetos*", etc.). Las entidades descriptas utilizando EML deben tener información literaria (nivel de conocimiento) y pueden tener información de comportamiento (nivel de implementación).

Para la descripción de comportamiento (utilizado en las posibles implementaciones de la solución en un patrón en los casos apropiados), se ha creado un DSL especial (EML-BDL, § 12.x) que contiene una representación abstracta de construcciones comunes existentes en varios lenguajes orientados a objetos. La definición completa

[140] En un caso extremo, deberían poder ser manipulados con el Bloc de Notas de Windows.
[141] En los enfoques existentes, el nivel de conocimiento se describe en modo textual, el código fuente en algún lenguaje de programación y la estructura en un lenguaje de modelado (como OMT o UML). Este requisito apunta a que toda esta información pueda ser expresada con un mismo lenguaje.
[142] "modularly composable domain specific language" en el texto original
[143] En [Welicki04c] se presenta la plantilla general que se utiliza para describir a los patrones en las primeras versiones de EML (llamado entonces M4PS). Si bien permitía describir un amplio número de patrones no era flexible y no permitía describir a todas las plantillas existentes. Adicionalmente no servía para describir conceptos de soporte (dado que había sido diseñada sólo para describir patrones).

de este sub-DSL de EML puede encontrarse más adelante en el **apéndice B** de esta tesis.

12.2.1 EML es un DSL

Un lenguaje de dominio específico (DSL) es un lenguaje especializado para un problema específico [Czarnecki00]. La idea básica de un DSL es un lenguaje de ordenador orientado a resolver un tipo particular de problemas a diferencia de un lenguaje de propósito general que es creado para resolver cualquier clase de problema [Fowler02]. Los DSLs pueden ser textuales o gráficos y pueden tener diferentes niveles de especialización. En general, son necesarios varios DSLs para especificar una aplicación completa. [Czarnecki00]. Para más detalles sobre DSLs, ver el **capítulo 7** de esta tesis.

EML es un "high-level modularly composable" [Czarnecki00] que incluye cinco sub-lenguajes de dominio específico más limitados, pero más especializados:

- **Properties Definition Language (EML-PDL)**: expresa la información literaria (parte fundamental del nivel de conocimiento) de una entidad a partir de composición dinámica de propiedades atómicas (por ejemplo, la plantilla de un patrón es un conjunto de propiedades).

- **Relationships Definition Language (EML-RDL)**: expresa relaciones entre entidades. Las entidades pueden ser del mismo o diferentes tipos. Se pueden expresar relaciones incompletas, donde uno de los elementos no existe o no está registrado en el catálogo.

- **Annotation Language (EML-AL)**: anotación de la entidad mediante etiquetas.

- **Structure Definition Language (EML-SDL)**: expresa la estructura de la entidad (participantes, relaciones y responsabilidades).

- **Behavior Definition Language (EML-BDL)**: expresa el comportamiento de los participantes de la entidad (declarados usando EML-SDL) a un alto nivel de abstracción. Contiene abstracciones de construcciones comunes en lenguajes de programación.

Cada uno de estos lenguajes tiene un objetivo bien definido y responsabilidades muy claras. El resultado de la combinación e interacción entre estos lenguajes es un marco de trabajo[144] para describir patrones y conceptos de soporte. En el **apéndice B** se presenta en forma detallada la especificación de cada uno de estos lenguajes en la versión 1.0 de EML.

[144] framework

12.2.2 Dominio de Definición de Patrones

Los patrones no existen aislados de un contexto. De la misma forma en que una implementación no es suficiente para transmitir un patrón, un patrón puede no ser suficiente para transmitir el conocimiento que intenta expresar. Para subsanar esta situación hemos definido una ontología sencilla para formalizar el dominio de la definición de patrones. En este dominio, se describen patrones y conceptos de soporte que añaden información a los patrones (por ejemplo lenguajes de patrones, categorías, autores, refactorizaciones, principios de orientación a objetos, etc.). Los patrones y los conceptos se describen en forma similar, utilizando EML.

12.2.2.1 Ontología de Entidades

¿Qué es una ontología para el propósito de nuestra investigación? Hemos utilizado la definición de Tomas Gruber que dice que "*una ontología es la especificación de una conceptualización*"[145] [Gruber93] y que "*una ontología es una descripción (como la especificación forma de un programa) de los conceptos y relaciones que pueden existir para un agente o comunidad de agentes*"[146] [Gruber93]. Según estas definiciones el objetivo de la ontología es habilitar la compartición y reutilización de conocimiento [Gruber93], que es uno de los pilares sobre los que se funda nuestro modelo [Welicki06b].

En la figura 12.1 a continuación se muestra la estructura de la ontología que hemos utilizado en el prototipo construido en la tesis (detallado en los **capítulos 15** y **16**), donde el patrón es el concepto central y está relacionado con otros conceptos que a su vez están relacionados entre sí.

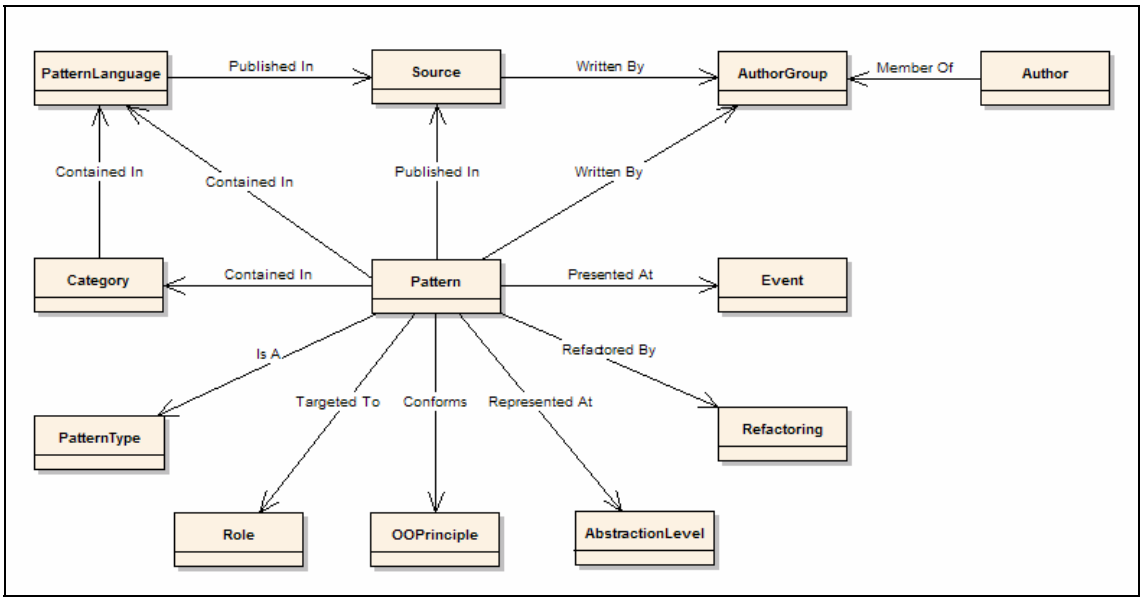


Figura 12.1 – Ontología utilizada para definir el dominio de la definición de patrones, adaptada de [Welicki06]

[145] "an ontology is a specification of a conceptualization" [Gruber93]
[146] "an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents" [Gruber93]

Es importante destacar que existen otras ontologías especializadas en la descripción de patrones como [WOP] y [W3CSWE] para conceptos de ingeniería del software. El problema que hemos encontrado con ellas es la rigidez y complejidad de producción (§ 10.1.5.1). En nuestro caso, la ontología no es determinante en el dominio y puede definirse y modificarse dinámicamente ad-hoc, dado que las relaciones se construyen a partir de uno de los sub-lenguajes de EML (que se presenta más adelante en § 12.4.1).

12.2.3 Secciones de una Entidad EML

EML permite representar una entidad en un único fichero. Una representación EML es auto-contenida: esto significa que contiene toda la información necesaria para expresar un patrón o un concepto[147], incluyendo la información literaria, la información de implementación y metadatos para clasificación y búsquedas. Esta representación puede ser utilizada con múltiples propósitos (§ 12.2.6), como por ejemplo, visualizar la información literaria, generar artefactos de software (modelos, código fuente, XMI, etc.), generar documentación, ver relaciones, etc.

EML no se limita a la descripción de patrones, permitiendo describir cualquier tipo de entidad (lenguajes de patrones, refactorizaciones, categorías, conceptos de orientación a objetos, libros, autores, etc.)

Una meta-especificación EML se compone de las siguientes partes:

- **Identificación:** información general para identificar a la entidad (nombre de la entidad, tipo de entidad).

- **Etiquetas (Tags):** etiquetas para anotar a las entidades [Technoratti]. Decora a la entidad con metadatos descriptivos que pueden ser utilizados para búsquedas o categorización.

- **Relaciones**: relaciones entre la entidad que esta siendo descripta y otras entidad (por ejemplo "ABSTRACT FACTORY está contenido en el lenguaje de patrones del GoF", "FACTORY METHOD es un patrón de diseño", "ABSTRACT FACTORY contiene a FACTORY METHOD", etc.)

- **Resumen**: un resumen de la entidad que se está describiendo.

- **Plantilla (Template):** información literaria de la entidad. En el caso de los patrones, contiene la información de la plantilla [C2PatternForm] (para más detalles sobre las plantillas existentes, ver el **capítulo 4**).

- **Estructura**: Estructura de la entidad (participantes con sus relaciones y responsabilidades).

[147] Tiene toda la información necesaria sobre la entidad que se está describiendo y relaciones con otras entidades que aumentan la información sobre ésta.

- **Implementación**: información sobre el comportamiento de la entidad (nivel de implementación). En esta sección tenemos dos partes bien diferenciadas: la implementación base, que es el modelo general de comportamiento y las implementaciones concretas, que son ejemplos específicos del comportamiento de la entidad[148].

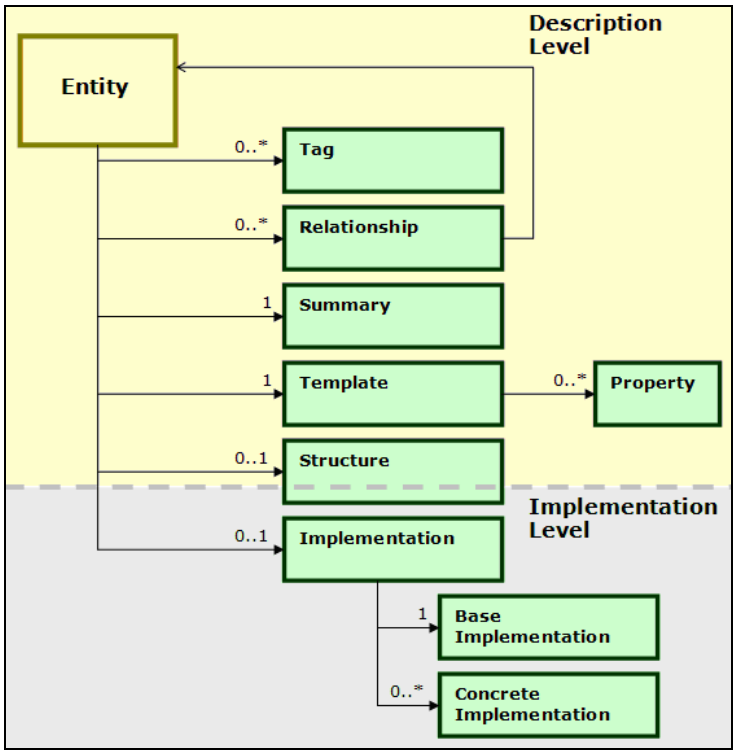


Figura 12.2 – Definiendo entidades con EML (tomado de [Welicki06]).

Como puede observarse en la figura 12.2, el nivel de conocimiento se compone de las secciones *Anotaciones*, *Relaciones*, *Resumen*, *Plantilla* y *Estructura*. El nivel de implementación consiste de las secciones *Estructura* e *Implementación*. En la misma figura puede observarse también que la sección *Estructura* es compartida por los niveles de conocimiento e implementación. Esto se debe a que contiene una enumeración formal de los participantes y sus relaciones (nivel de implementación) y la descripción textual de las responsabilidades de los participantes (nivel de conocimiento).

Las secciones *Estructura* e *Implementación* son opcionales. Algunas entidades pueden no necesitarlas. Por ejemplo, los patrones de "*Design Patterns*" [GoF95] tienen ambos niveles. Los patrones de "*Analysis Patterns*" [Fowler96] sólo tienen nivel de conocimiento. Los antipatrones [BMMM98] en algunos casos tienen ambos niveles (LAVA FLOW, THE BLOB, etc. [BMMM98]) y en otros sólo tienen el nivel de conocimiento (CORNCOB, THE FEUD, etc. [BMMM98]). Como conclusión, podemos afirmar que el nivel de conocimiento es obligatorio y que el nivel de implementación es opcional.

[148] Podría establecerse un paralelismo a modo de ejemplo entre la implementación base y una clase base abstracta y la implementación concreta y una clase concreta que implementa la clase base abstracta.

12.2.4 Descripción Patrones con EML

La descripción de los patrones no es un tema trivial y se vuelve más complicada cuando se eleva al nivel meta. En este punto es interesante recordar la frase del GoF: "*Encontrar patrones es mucho más fácil que describirlos*" [GoF95].

EML provee un conjunto completo de características que permiten describir diversos aspectos de una entidad. Estas características se dividen en dos grandes grupos: las que están orientadas a describir el nivel de conocimiento y las orientadas a describir el nivel de implementación.

EML permite representar el nivel de conocimiento en forma flexible. Las plantillas de los patrones se describen mediante la composición de elementos atómicos a los cuales llamamos ***propiedades*** (pueden ser creadas y modificadas por los usuarios). Estas propiedades se refieren a un aspecto concreto de una entidad (por ejemplo, en el caso del patrón pueden ser el problema, la solución, el contexto, la aplicabilidad, etc.). Mediante la composición dinámica de propiedades se puede crear un número ilimitado de plantillas. Los patrones pueden relacionarse con otras entidades mediante un lenguaje sencillo de definición de relaciones. Este lenguaje permite tratar con información incompleta y por ende un patrón puede relacionarse con entidades que no están registradas en el catálogo. Finalmente, las entidades pueden ser anotadas con etiquetas, que pueden ser utilizadas posteriormente para búsquedas y clasificación colaborativa.

Respecto al nivel de implementación, EML contiene un conjunto de construcciones que permiten describir el comportamiento de un conjunto de artefactos de software sin utilizar ningún lenguaje de programación en particular y sin compromiso con ninguna plataforma de ejecución. Estas construcciones se dividen en dos grandes grupos:

- **Descripción de Artefactos**: descripción de módulos de código (por ejemplo, clases) y sus elementos.

- **Descripción de Algoritmos**: descripción del comportamiento de un elemento de un módulo de código (por ejemplo, métodos).

En el **apéndice B** se presenta una descripción de EML 1.0.

12.2.4.1 EML-PDL: Descripción de Propiedades (Plantillas)

EML-PDL (*Entity Metaspecification Lenguaje – Property Definition Language*) es un sub-lenguaje de dominio específico[149] (DSL) de EML que permite describir plantillas de patrones a través de la composición dinámica de elementos atómicos sencillos a los cuales llamamos "propiedades", ofreciendo un marco de trabajo ágil, flexible y

[149] Recordar que EML es un DSL compuesto de varios sub-DSLs

extensible [Welicki04] para describir entidades (patrones y conceptos) de ingeniería del software.

12.2.4.1.1 El Problema de las Plantillas Fijas

Describir en forma general y flexible las propiedades que conforman a una entidad es una tarea muy ardua y compleja (esto ha quedado demostrado en el **capítulo 10**, § 10.1). En el caso de los patrones, por ejemplo, existe un gran número de plantillas para describirlos (como hemos podido observar en el **capítulo 4**, § 4.1) y crear una nueva que resuma a todas éstas no es una tarea trivial. Tampoco creemos que sea una tarea práctica, dado que siempre estaremos dejando algo importante de lado o imponiendo un esfuerzo importante de traducción[150].

En las primeras versiones de EML[151] (versiones que estaban totalmente enfocadas a la descripción de patrones) [Welicki04c] [Welicki05], habíamos creado una plantilla general con el objeto de describir cualquier tipo de patrón. El fundamento de este esfuerzo de normalización eran los posibles isomorfismos existentes entre las distintas plantillas, tomando como base a la plantilla canónica (§ 4.1.7). En la siguiente figura se muestra la relación entre la plantilla canónica y la del GoF (§ 4.1.1), que ha sido nuestro punto de partida.

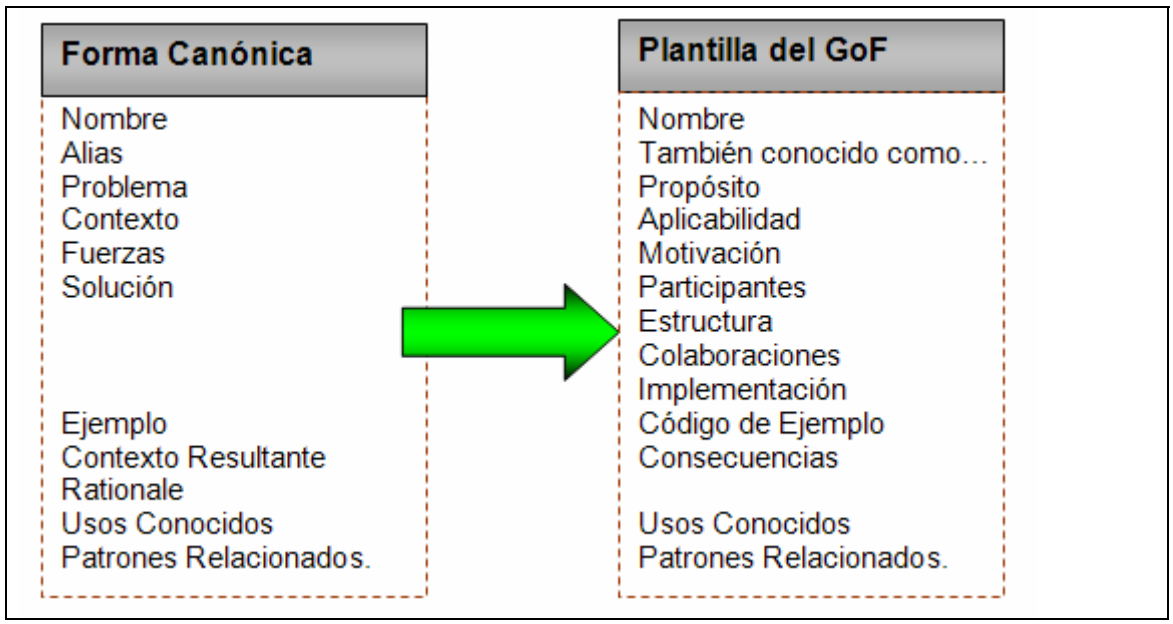


Figura 12.3 – Relación entre las plantillas Canónica y del GoF, adaptado de [PPR]

Al intentar describir patrones de varios lenguajes diferentes, este enfoque demostró ser insuficiente (la plantilla resultante no contenía todas las secciones necesarias para incluir a todas las plantillas). Esta situación se vio ampliamente agravada ante la necesidad de describir otro tipo de entidades que no son patrones (categorías, conceptos de orientación a objetos, refactorings, etc.). Por tanto, concluimos que la

[150] Los problemas mencionados son los que hemos detectado en PLML [PLML03] en capítulo 10 de esta tesis.
[151] En sus primeras versiones, EML se llamaba M4PS (Metalenguaje for Pattern Specification) [Welicki04c] [Welicki05]

descripción a partir de la búsqueda de isomorfismos o normalización no era la vía adecuada para la resolución del problema de la descripción de las plantillas de los patrones.

12.2.4.1.2 EML-PDL: de Plantillas Fijas a Plantillas Basadas en Composición Dinámica

A partir de estos antecedentes creamos un sistema flexible de definición de plantillas que permite la combinación de elementos existentes en cualquiera de ellas. Este sistema se basa en la composición dinámica de unidades atómicas sencillas de información, a las cuales llamamos "propiedades". Una plantilla es un conjunto de propiedades vinculadas. El lenguaje de definición de plantillas es capaz de tratar con cualquier agregación de elementos sencillos que siguen unas reglas sintácticas básicas[152] (en este punto, es necesario recordar que EML esta basado en XML). El lenguaje que da soporte a este sistema, permitiendo expresar las propiedades y sus relaciones es EML-PLD.

El elemento fundamental de expresión de EML-PDL es la propiedad. Una propiedad permite describir un aspecto concreto de una entidad. A partir de la composición de un conjunto discreto y finito de propiedades se consigue una plantilla (§ 4.1) que contiene la descripción de una entidad concreta. Esta composición puede modificarse dinámicamente en cualquier momento, estableciendo de esta forma un marco para la constante evolución y mejora de las entidades que describen (§ 11.6). En este contexto podemos decir que la descripción de una entidad es un fenómeno emergente (**capítulo 9**), que se produce a partir de la combinación de elementos simples especializados en la descripción de un aspecto local (las propiedades).

Además de soportar cualquier tipo de propiedades (que pueden ser definidas dinámicamente por el autor de la meta-especificación), existen algunos "nombres reservados" que se refieren a propiedades especiales que el analizador del lenguaje sabrá tratar en forma especial. Estas propiedades especiales se definen mediante metadatos en forma dinámica, dotando al lenguaje de un gran nivel de extensibilidad y flexibilidad.

12.2.4.1.3 Incluyendo Información de Fuentes Externas

Existen propiedades predefinidas para integrar una meta-especificación concreta con información de fuentes externas (a partir de relaciones sintácticas basadas en hipervínculos). De esta forma una entidad puede reaprovechar conocimiento existente a partir de vinculación mediante propiedades predefinidas[153]. Estas propiedades funcionan como elementos atómicos para la composición de plantilla en el mismo modo que cualquier otro tipo de propiedad.

[152] Estas reglas son las de "xml bien formado" (well-formedness) [W3C]
[153] Este es el uso principal de las propiedades con "nombres reservados" mencionadas al final de § 12.2.4.1.2

12.2.4.1.4 Propiedades Atómicas y Propiedades Compuestas

Existen dos tipos principales de propiedades que pueden ser utilizadas para describir a las entidades: las propiedades atómicas y las propiedades compuestas. La forma de interacción entre estos tipos de propiedades se basa en los principios expuestos en el patrón COMPOSITE [GoF95].

- Las **propiedades atómicas** se refieren a elementos sencillos que expresan un único concepto o tipo de dato. Ejemplos de este tipo de propiedades incluyen cadenas, fechas, relaciones, etc.

- Las **propiedades compuestas** permiten crear propiedades compuestas a su vez de otras. Las propiedades compuestas permiten crear propiedades complejas a partir de elementos sencillos. Por ejemplo, para describir los beneficios de aplicar un patrón podemos crear una propiedad compuesta llamada `beneficios` que se compone de propiedades de tipo `beneficio`. La propiedad `beneficio` es a su vez una propiedad compuesta que se compone de los elementos atómicos `título` y `discusión`.

A partir de la composición dinámica de propiedades se pueden crear elementos complejos sin una estructura rígida previamente definida. La composición dinámica basada en vinculación tardía provee un alto grado de flexibilidad y extensibilidad[154], permitiendo modificar la estructura de la plantilla de la entidad en tiempo de ejecución[155].

12.2.4.2 EML-RDL: Descripción de Relaciones

EML-RDL (*Entity Metaspecification Lenguaje – Relationship Definition Language*) es un sub-lenguaje de dominio específico[156] (DSL) de EML que permite describir relaciones entre entidades EML. Puede trabajar con información incompleta, permitiendo introducir relaciones entre entidades existentes y elementos no introducidos en el catálogo. A partir de estas relaciones se puede armar una "red" de conceptos que pueden ser navegados de diversas formas. Esta red puede utilizarse también para clasificación dinámica y para alimentar un motor de inferencias.

La sintaxis de EML-RDL es muy sencilla, permitiendo establecer en forma sencilla relaciones entre entidades. Dada una meta-especificación EML de una entidad, una relación de EML-RDL especifica el tipo de la entidad destino, su dirección en el sistema (§ 13.4.1.3) y el tipo de relación que vincula a la entidad destino con la entidad actual. Las relaciones se incluyen dentro de la meta-especificación de una entidad (recordar que una meta-especificación es auto-contenida § 12.2.3) y por lo tanto, el origen está implícito (es la entidad que está siendo definida).

---

[154] Esta aserción se ve reforzada por el principio de diseño sobre el que se fundan todos los patrones de [GoF95] "utilizar composición en lugar de herencia". Aunque en este caso la frase se refiere a programación orientada a objetos la idea subyacente es muy general, siendo también aplicable a nuestro problema concreto.
[155] Esta característica puede ser encontrada también en los sistemas reflectivos [Ortin03]
[156] Recordar que EML es un DSL compuesto de varios sub-DSLs

12.2.4.2.1 Por qué Hemos Creado EML-RDL en Lugar de Utilizar OWL

¿Por qué hemos elegido describir el dominio y sus relaciones con EML y no hemos utilizado OWL [W3C]? El objeto de EML-RDL es proveer una infraestructura que permita la descripción de relaciones que puedan ser creadas, establecidas y modificadas en forma flexible y sencilla en tiempo de ejecución emergentemente (recordar la discusión sobre sistemas emergentes en el **capítulo 9**), sin la necesidad de un coordinador de alto nivel o un diseño previo (las relaciones se pueden establecer ad-hoc en forma dinámica). Los conceptos, tipos de relaciones y extremos pueden ser creados y eliminados en forma ágil, sin necesidad de estar registrados previamente, facilitando a usuarios finales la creación de relaciones a través de una gramática sencilla.

De todos modos, la traducción entre EML y OWL es una tarea relativamente sencilla, dado que comparten varios conceptos fundamentales (volveremos a este tema en § 12.2.4.2.3, cuando tratemos la relación entre EML y RDF).

12.2.4.2.2 EML-RDL en la Ontología de Patrones

Si EML-RDL permite definir a las ontologías en forma dinámica y no requiere un diseño previo, la pregunta es entonces para qué sirve la ontología presentada previamente en la figura 12.1 (§ 12.2.2.1). La respuesta es muy simple: hemos utilizado a este modelo como punto de partida, para establecer las relaciones los tipos de entidades que existirán inicialmente en el catálogo. Como esperamos que el catálogo sea un elemento "vivo", se introducirán nuevos patrones y conceptos constantemente. Por lo tanto, nuestro objetivo es crear un sistema de clasificación que soporte la evolución emergente [Resnick94] basada en el conocimiento de la comunidad [Johnson01].

Al no requerir un diseño de alto nivel y tener la posibilidad de evolucionar en función de las relaciones existentes (que pueden estar incompletas) el lenguaje puede dar el soporte básico a un "sistema de feedback emergente", que permitirá a los usuarios "ajustar" las relaciones en forma colaborativa, sin un coordinador general ni un diseño de alto nivel, estableciendo redes dinámicas de conceptos relacionados[157].

Finalmente, el lector podrá pensar "el dominio que se muestra en la figura 12.1 es incompleto". Esta situación es inevitable. Según Czarnecki y Eisenecker ([Czarnecki00], pp34), "*todo dominio real es infinito en el sentido de que siempre se puede refinar con más detalles…Consecuentemente, los intentos de formalizar completamente los dominios están condenados al fracaso*". Por tanto, un enfoque "de arriba-hacia-abajo"[158] no es adecuado para resolver este problema[159]. Es buen momento para aclarar

[157] Un fenómeno similar al de la Web [W3C] [BernersLee00], aunque a una escala significativamente menor y más acotada. La Web a demostrado que este tipo de sistemas sin una regulación centralizada o ente publicador central son posibles y que pueden ser extremadamente exitosos.
[158] Top-Down (en inglés)
[159] Este es el enfoque utilizado por el proyecto Web of Patterns [WOP], donde se define una ontología para describir patrones y refactorizaciones.

nuevamente que la idea es que las relaciones surjan en forma emergente, a partir de interacciones entre los miembros de las comunidades (como por ejemplo, la comunidad de patrones, de HCI[160] o de ingeniería del software) que son quienes tienen el conocimiento.

12.2.4.2.3 EML-RDL y RDF

La forma de describir las relaciones en EML-RDL es muy similar a los triples de RDF. Por lo tanto, estas relaciones pueden ser traducidas fácilmente a RDF y almacenadas en una base de datos RDF [RDFDB] complementaria para su posterior uso. Por lo tanto, existen tres vías directas de integración entre EML y RDF / OWL.

- **Crear traductores** de EML-RDL a RDF / OWL

- **Aumentar EML utilizando RDF y OWL**, a efectos de aprovechar ontologías existentes como la que está siendo creada en el *Web of Patterns Project* [WOP].

- **Utilizar EML y RDF en forma simultánea**. De esta forma, ambos lenguajes se complementan obteniendo lo mejor de ambos (las capacidades de descripción y evolución de EML con las capacidades de anotación de meta-datos de RDF). En este caso ambas especificaciones conviven y evolucionan en forma paralela.

12.2.4.3 EML-AL: Anotación de Entidades

EML-AL (*Entity Metaspecification Lenguaje – Annotation Language*) provee una infraestructura para anotar a las entidades con etiquetas. Esta anotación puede realizarse en modo colaborativo (la forma de "implementación" de la infraestructura de edición de etiquetas es responsabilidad de la infraestructura de catalogación y visualización).

Las etiquetas permiten anotar elementos con meta-datos descriptivos de baja carga semántica y formal. Las etiquetas son el medio para que emerjan las folksonomías, que son mecanismos de categorización colaborativos [Estatella05]. Ayudan a la categorización, clasificación y búsquedas de entidades en el catálogo. Al crearse en forma colaborativa y no supervisada se convierten en un medio social de clasificación y contextualización de conocimiento.

Las etiquetas creadas con EML-AL pueden integrarse con sistemas populares de folksonomías como Del.ici.us [Delicious] o Technoratti [Technoratti] e incluso pueden crearse nubes de etiquetas[161] [TagCloud] a partir de éstas.

[160] Human Computer Interaction (Interacción Persona-Ordenador)
[161] Tag Cloud (en inglés)

EML-AL es el sub-DSL más sencillo de EML: consta sólo de una primitiva sencilla que permite añadir una etiqueta a una entidad y de un contenedor para dichas etiquetas.

12.2.4.4 EML-SDL: Descripción de la Estructura

EML-SDL (*Entity Metaspecification Lenguaje – Structure Description Language*) permite describir la estructura de la solución en un patrón en términos de grupos de entidades relacionadas. Una entidad EML contiene un conjunto de participantes que se relacionan entre si. Estos participantes junto con sus relaciones conforman la estructura del nivel de implementación.

La descripción de la estructura se realiza en tres pasos sencillos y muy esquemáticos:

1. **Definición de los participantes**: se definen los participantes en la solución al problema. Para cada participante se define el rol que ocupa, la cardinalidad y si es o no abstracto. Los participantes se refieren en forma general a entidades que pueden ser traducidas a artefactos de software (por ejemplo, clases), aunque no están limitados a esto (recordar que la semántica de utilización la pone el usuario de la meta-especificación).

2. **Definición de las relaciones entre los participantes**: una vez creados los participantes se establecen las relaciones entre ellos. Las relaciones pueden ser de creación, asociación, composición y herencia.

3. **Definición de las responsabilidades de los participantes**: descripción de las responsabilidades de los participantes. Esta descripción se realiza en prosa (es la única parte de la descripción de la estructura que no es formal).

Este modelo de descripción ha sido diseñado teniendo en cuenta el orden en que se realiza la traducción de un patrón existente a EML, a efectos de facilitar la producción de entidades EML (y codificación de patrones existentes).

12.2.4.5 EML-BDL: Descripción del Comportamiento

EML-BDL (*Entity Metaspecification Lenguaje – Behavior Description Language*) permite describir el comportamiento de los participantes en la solución de un patrón (los participantes se definen utilizando EML-SDL). El comportamiento se describe a un alto nivel de abstracción, con total independencia de los lenguajes de programación.

Para poder definir el comportamiento debe existir previamente una estructura (descrita utilizando EML-SDL), dado que EML-BDL define el comportamiento de los participantes de la solución.

EML-BDL puede ser traducido a cualquier lenguaje de programación de alto nivel. En el prototipo desarrollado para esta tesis (presentado en los **capítulos 15** y **16**) se desarrollaron tres traductores: C#, VB.NET y Java.

12.2.4.5.1 Describiendo el Comportamiento de un Participante

EML-BDL permite describir en forma abstracta el comportamiento de los participantes (que han sido declarados previamente mediante EML-SDL, § 12.2.4.4).

El comportamiento se describe mediante un conjunto de primitivas generales que se incluyen en el **apéndice B**. Estas primitivas son abstracciones básicas presentes en múltiples lenguajes de programación (manipulación de variables, control de flujo, bucles, invocación, etc.) y se pueden combinar para expresar algoritmos.

Los algoritmos se incluyen en métodos (que son conjuntos de primitivas), para los cuales se puede especificar una firma (parámetros de entrada y salida, tipo de retorno, si es abstracto, si es virtual, si es final o la cardinalidad). Esta descripción de algoritmos puede ser modificada y adaptada en tiempo de ejecución.

EML-BDL permite describir propiedades (junto con sus modificadores y tipos) y conjuntos de métodos (un método tiene una firma y agrupa un conjunto de primitivas atómicas que definen un algoritmo).

EML-BDL describe el comportamiento de una entidad (mediante algoritmos y propiedades) a un alto nivel de abstracción, pudiendo ser traducido posteriormente (junto con EML-SDL, § 12.2.4.4) a cualquier lenguaje de programación de alto nivel (§ 13.4.10).

12.2.4.5.2 Implementación Base e Implementaciones Concretas

EML-BDL permite definir una implementación base prototípica y partir de ésta diversas implementaciones concretas. De esta forma, se pueden dar distintas alternativas de implementación para una misma solución. De esta forma se puede determinar una estrategia general de solución, describirla en términos abstractos como implementación base y a partir de esta crear múltiples implementaciones concretas que redefinan aspectos particularmente relevantes.

12.2.5 Gradiente Semántico

Probablemente los autores no cuenten con toda la información al momento de crear la meta-especificación EML de una entidad o describir relaciones entre entidades. EML no obliga a los autores a tener toda esta información a priori para poder describir una entidad. En lugar de esto hemos creado el "gradiente semántico", que permite a aumentar la carga semántica asociada a un término específico en función de la información que se disponga sobre éste.

Cualquier término o concepto atraviesa un ciclo de vida en el gradiente semántico que tiene estos tres estados:

- **Palabra:** el término es una palabra en la descripción de la entidad. No hay semántica asociada a la palabra[162]. Puede ser utilizada en una búsqueda por coincidencia de texto[163].

- **Etiqueta**: el término puede ser una etiqueta vinculada a una entidad. No tiene una especificación, pero puede ser utilizado para búsqueda y clasificación basada en etiquetas.

- **Entidad**: el término es descrito como una entidad EML completa. Tiene significado completo y puede ser diseccionado como cualquier otra entidad EML.

Como hemos mencionado anteriormente, EML no obliga a especificar las entidades en forma completa a priori. Las entidades pueden ser "*descubiertas*" durante el proceso de descripción. Para explicar mejor estos conceptos, presentaremos a continuación un ejemplo concreto. Al especificar los patrones ABSTRACT FACTORY y FACTORY METHOD [GoF95] la palabra "factory" está presente en la descripción de ambos patrones. Por lo tanto, una búsqueda por coincidencia de textos de la palabra "factory" retorna como resultado un conjunto de entidades que incluye a éstas dos (y a todas las que tengan a este término en algún lugar de su meta-especificación). Podemos descubrir que el concepto "factory" es más que una palabra, aunque no tengamos una definición. En este caso, podemos crear una etiqueta "factory" y vincularla a los patrones que estamos describiendo. Al realizar una búsqueda por etiqueta obtendremos como resultado un conjunto de entidades anotadas con la etiqueta utilizada como argumento de búsqueda. Finalmente, al cabo de un tiempo, hemos encontrado información sobre el concepto "factory" y podemos tomar la decisión de crear una entidad EML completa con esta información. En este caso, primero creamos la meta-especificación EML de la entidad "Factory" y luego la vinculamos a los patrones ABSTRACT FACTORY y FACTORY METHOD. En este ejemplo hemos visto en forma práctica como el término "factory" a pasado por todos los estados del gradiente semántico (palabra, etiqueta y entidad).

[162] Nos referimos a carga semántica que permita explotar esa información. Esta palabra tiene, por supuesto, un significado en el contexto de la descripción, pero no puede ser utilizada para realizar relaciones, búsquedas o inferencias en forma automática.
[163] Full-text search (en inglés).

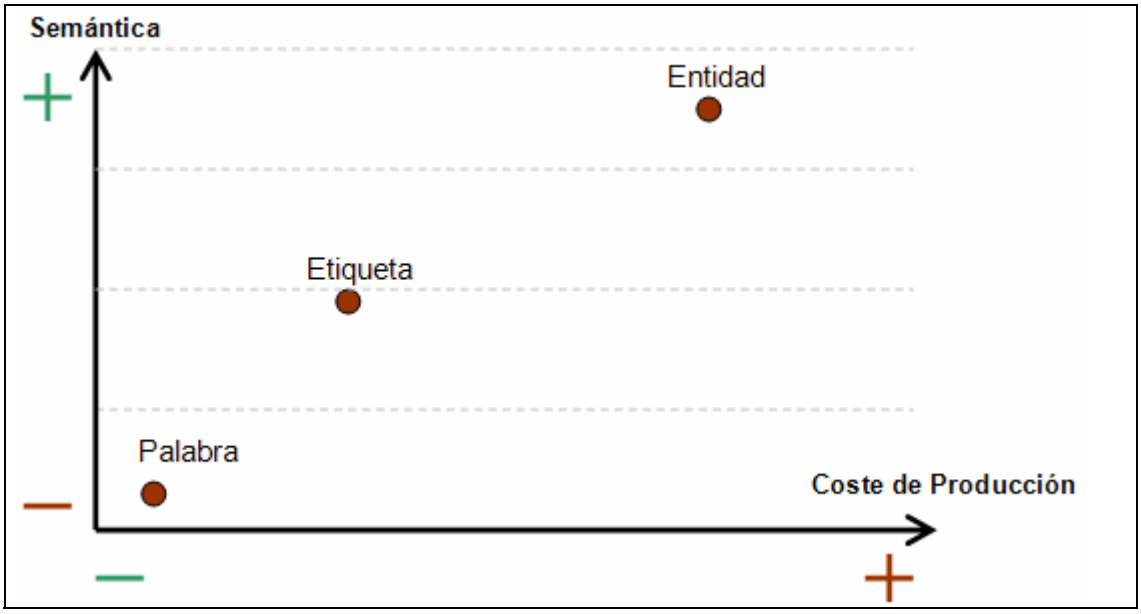


Figura 12.4 – El gradiente semántico.

12.2.6 Visualización y Utilización de EML

EML es sólo un lenguaje de meta-especificación y no provee ningún tipo de aserción o guía sobre como mostrar o consumir las entidades que se describen utilizándolo. Esto implica que la responsabilidad de explotación de la entidad recae absolutamente en el cliente de ésta.

En la figura 12.5 se muestra como una entidad descripta utilizando EML puede ser traducida a distintos artefactos (los artefactos que se muestran son sólo un subconjunto de las traducciones posibles). Nuevamente, consideramos importante repetir que la especificación de EML no hace ninguna referencia sobre cómo visualizar los datos que contiene y que ésta es responsabilidad de quien utiliza la entidad.

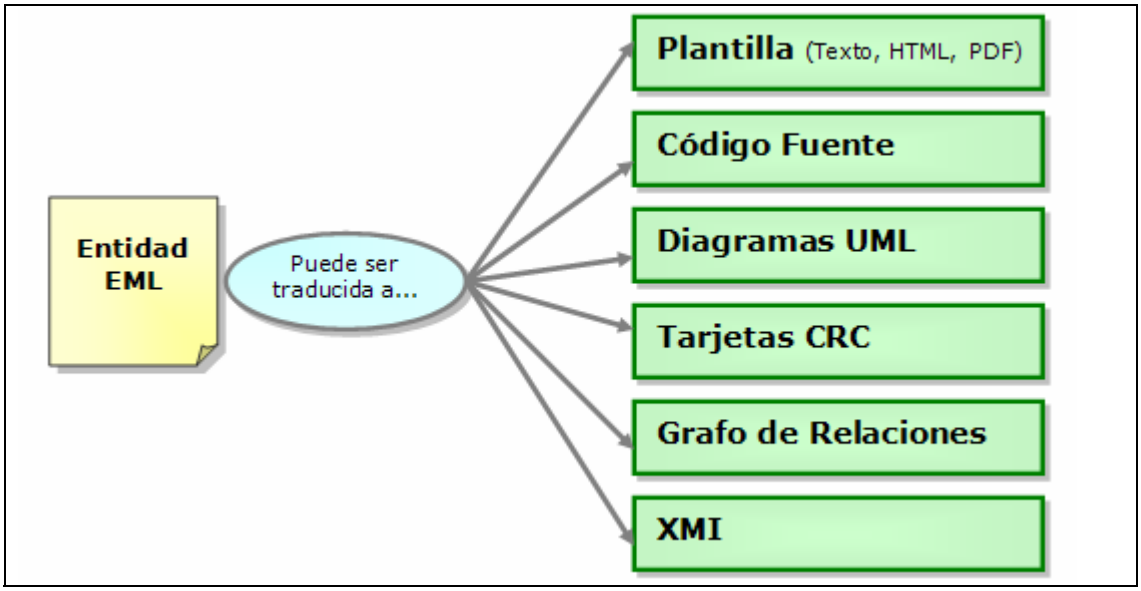


Figura 12.5 – EML no contiene construcciones que especifiquen pautas de visualización. La forma de consumir EML es responsabilidad de quien lo utiliza.

12.2.7 EML 1.0

Durante el desarrollo del prototipo hemos creado la versión 1.0 de EML[164]. La especificación completa se presenta en el **apéndice B**. En el **capítulo 16** se presenta un conjunto de ejemplos seleccionados de descripciones de fragmentos de patrones y entidades utilizando este lenguaje (§ 16.10).

12.3 Conclusión: Revisión de los Requisitos

EML cumple con todos los requisitos establecidos al inicio de este capítulo (§ 12.1). En la tabla 12.1 a continuación, se verifica la forma en que se realiza cada uno de estos requisitos, incluyendo en cada caso una breve frase que resume como se realiza y referencias a las secciones del capítulo donde puede ampliarse esta información.

[164] Esta versión 1.0 es el producto de la evolución de versiones anteriores, como las que se pueden encontrar en [Welicki04c] y [Welicki05]

Requisito	Descripción	Realización
RL-1	Alto nivel de Abstracción	EML es sólo un lenguaje de meta-especificación y no provee ningún tipo de aserción o guía sobre como mostrar o consumir las entidades que se describen utilizándolo (§ 12.2.6)
RL-2	Soportar todas las plantillas de patrones	EML permite representar entidades (patrones y conceptos de soporte) a un alto nivel de abstracción (§ 12.2.4.1).
RL-3	Descripción de relaciones	EML permite describir relaciones entre entidades (§ 12.2.4.2).
RL-4	Descripción del comportamiento independiente de plataforma	Los participantes y la implementación se describen con total independencia de la plataforma y lenguajes de programación (§ 12.2.4.5)
RL-5	Anotación mediante etiquetas	EML permite anotar a las entidades utilizando etiquetas (§ 12.2.4.3)
RL-6	Describir entidades con y sin implementación	El nivel de conocimiento es obligatorio. El nivel de implementación es opcional (§ 12.2)
RL-7	Reaprovechar el conocimiento existente	EML incluye construcciones para relacionar una entidad a otros elementos existentes, ya sea en el catálogo (**capítulo 13**) o en la Web (§ 12.2.4.1.3).
RL-8	Metadatos para búsquedas y clasificación	EML permite anotar a las entidades con información para búsquedas usando etiquetas y relaciones (§ 12.2.4.2, § 12.2.4.3, § 12.2.5)
RL-9	Facilidad de producción	Al ser un DSL textual basado en XML puede producirse utilizando cualquier editor de textos (§ 12.2). Adicionalmente, la posibilidad de describir entidades sin necesidad de contar con toda la información a priori (§ 12.2.4, § 12.2.5) facilita la producción de meta-especificaciones.
RL-10	Uniformidad	Los niveles de conocimiento e implementación se describen utilizando los DSLs de EML (§ 12.2.3). Los patrones y conceptos de soporte se describen utilizando EML.

Tabla 12.1 – Revisión del cumplimiento de los requisitos del lenguaje de meta-especificación.

Capítulo 13
El Catálogo

"Para ciertas personas, la lengua reducida
a su principio esencial es una nomenclatura,
es decir, una lista de términos que
corresponden a tantas otras cosas"
Ferdinand de Saussure

En el **capítulo 11** hemos realizado un análisis y argumentación del contexto de la solución (§ 11.1.11) al problema planteado al inicio de esta tesis (§ 1.2). En esa misma sección hemos establecido por deducción la necesidad de crear una infraestructura de catalogación y hemos determinado una serie de requisitos generales para ésta[165]. Más adelante en ese capítulo, presentamos brevemente el catálogo como solución a la gestión de las meta-especificaciones EML (**capítulo 12**) de patrones y conceptos de ingeniería del software.

El catálogo se compone de meta-especificaciones EML de patrones y conceptos de soporte y de la infraestructura necesaria para poder gestionar, exponer, compartir y utilizar esas meta-especificaciones. Es el elemento más complejo de la solución, debido principalmente a los siguientes motivos:

- las descripciones de las entidades y sus relaciones son complejas de generar (requieren mucho trabajo de estudio y análisis de la literatura existente) y

- el catálogo no se limita a ser un contenedor pasivo de información: expone su contenido mediante una serie de mecanismos y ofrece funcionalidades avanzadas para gestionar sus contenidos (búsquedas, colaboración, suscripción, seguridad, etc.).

En este capítulo presentaremos el modelo general del catálogo. Comenzaremos estableciendo los requisitos para el catálogo en forma detallada. A lo largo del capítulo presentaremos en detalle la arquitectura general y detalles particulares de la solución de catalogación propuesta en esta tesis. Finalmente, a modo de conclusión revisaremos los requisitos establecidos al inicio, contrastándolos con lo expuesto a lo largo del capítulo.

❖ ❖ ❖

[165] En ese mismo capítulo, prometimos ampliar esos requisitos generales. Los requisitos ampliados se presentan en este capítulo.

13.1 Requisitos para el Catálogo

En el **capítulo 11** presentamos en forma resumida y coloquial (§ 11.1) los requisitos para la construcción de un catálogo de meta-especificaciones de patrones y conceptos de ingeniería del software, que se establecen a partir de los problemas recurrentes detectados en los enfoques existentes en el **capítulo 10** (§ 10.4). A continuación se refinan esos requisitos y se presentan en forma detallada en la siguiente lista:

RC-1. **Soporte para almacenar entidades**: brindar la estructura necesaria para almacenar meta-especificaciones de patrones y conceptos de ingeniería del software descriptos con EML. Este requisito se refiere a la estructura física donde vivirán las especificaciones. La estructura lógica es atendida por los requisitos RC-2, RC-3, RC-4 y RC-7.

RC-2. **Soporte para almacenar relaciones:** almacenar relaciones entre las meta-especificaciones almacenadas en el catálogo[166]. Este requisito se relaciona directamente con RC-1 y RC-4, dado que las entidades a relacionar pueden estar almacenadas en el catálogo y por lo tanto deben haber sido previamente registradas. También se relaciona con RC-6, dado que una entidad puede relacionarse con otra que todavía no ha sido incluida en el catálogo[167] (el catálogo debe saber como gestionar este tipo de situaciones). Las relaciones deben poder establecerse ad-hoc en forma dinámica.

RC-3. **Analizador de entidades EML**: el analizador debe poder verificar la corrección sintáctica y semántica de las entidades EML. El analizador debe ser flexible, pudiendo adaptarse rápidamente a cambios en la especificación de EML.

RC-4. **Registro de entidades:** facilidades para registrar, modificar y desregistrar entidades en el catálogo. Antes de registrar una entidad, ésta debe ser analizada con el analizador (RC-3). Las entidades deben almacenarse en la estructura física creada a tal fin (RC-1).

RC-5. **Soporte para búsquedas:** debe proveer soporte para realizar búsquedas en los contenidos del catálogo, utilizando las definiciones de las entidades, los metadatos y las relaciones entre entidades.

RC-6. **Capacidad para manipular especificaciones incompletas:** no debe ser necesario disponer de toda la información sobre una entidad para introducirla en el catálogo (esta puede ser introducida a posteriori). Esto nos permite también crear la infraestructura para dar soporte al *gradiente semántico* (presentado en el **capítulo 12**, § 12.2.5).

[166] Las entidades pueden ser del mismo o diferente tipo.
[167] Esta característica es soportada por EML-RDL (§ 12.2.4)

RC-7. **Exponer las entidades mediante un API OO:** exponer las entidades mediante un API de objetos, para que puedan ser utilizadas por otros componentes. Estos objetos pueden ser compartidos utilizando una arquitectura de componentes "en-proceso"[168] o de componentes distribuidos (.NET Remoting [NETRemoting], CORBA [CORBA], DCOM [Microsoft], etc.).

RC-8. **Exponer las entidades mediante servicios Web:** exponer las entidades mediante una interfaz de servicios Web [W3C]. La granularidad de esta interfaz es diferente a la de RC-7[169].

RC-9. **Direccionamiento de entidades:** cada entidad debe tener una dirección que permita hacer referencia a ella en forma unívoca e inequívoca. Esta dirección será utilizada para recuperar la entidad del catálogo.

RC-10. **El catálogo debe ser un "espacio de información vivo":** el catálogo debe proveer mecanismos para mantener "vivo" ese espacio de información, a partir de fomentar y soportar la interacción entre sus usuarios y/o hacer "push" de información.

RC-11. **Generación de código:** generar código fuente a partir de las especificaciones EML[170]. El marco de generación debe ser extensible, permitiendo incluir nuevos generadores o modificar los existentes.

13.2 El Catálogo de Patrones y Entidades

El catálogo de patrones es el elemento más complejo de la solución general (§ 11.3). Es el repositorio centralizado donde se encuentran alojadas todas las meta-especificaciones EML de patrones y entidades registradas en el sistema. No se limita a ser un mero contenedor pasivo de información y además de la gestión de la información provee la infraestructura necesaria para que ésta sea explotada por diversos clientes.

A modo de resumen, podemos decir que el catálogo se compone de un conjunto de meta-especificaciones de patrones y conceptos de soporte escritos utilizando EML y de la infraestructura necesaria para poder gestionar, exponer, compartir y utilizar esas meta-especificaciones. Ofrece también servicios avanzados como gestión de la seguridad, auditorías y mecanismos de suscripción para realizar "push" de información.

[168] In-process (en inglés)
[169] En este caso, estamos ante una interfaz de grano grueso ("coarse-grained interface") mientras que en el otro caso, ante una interfaz de grano fino ("fine-grained interface").
[170] Esto es aplicable a entidades EML que incluyen EML-SDL (§ 12.2.4.4) y EML-BDL (§ 12.2.4.5)

13.2.1 ¿Qué hay en el Catálogo?

Los elementos del catálogo se dividen en dos grandes grupos:

- **Componentes Pasivos**: conjunto de elementos persistentes donde cada uno contiene la meta-especificación EML de una entidad[171]. Incluye también la definición de los iteradores virtuales e imágenes asociadas a las entidades. Incluye los siguientes elementos:

    - *Entidades EML (Patrones y Conceptos de Soporte)*
    - *Relaciones entre entidades*
    - *Imágenes*
    - *Definición de Iteradores Virtuales*

- **Componentes Activos**: infraestructura que permite manipular, exponer y compartir los componentes pasivos del catálogo. Incluye los siguientes elementos:

    - *Interfaz de Registro de Entidades*
    - *FREP (Flexible Runtime Execution Platform)*
    - *Motor de Búsquedas*
    - *Iteradores Virtuales*
    - *Seguridad y Auditoria*
    - *Colaboración*
    - *Suscripción a Notificaciones*
    - *Fachada para exponer las entidades*
    - *Generación de código*
    - *Versionado*
    - *Direccionamiento*[172]

En la secciones a continuación se explican en forma detallada todos los elementos enumerados arriba.

13.2.2 Arquitectura Lógica General

El catálogo esta compuesto de una combinación de elementos activos y pasivos, los cuales dan soporte al almacenamiento y utilización de la información. Es a la vez un contenedor y proveedor de servicios de información. Podemos decir que el catálogo es una sumatoria de especificaciones EML de patrones y entidades, un conjunto de servicios dedicados a gestionar esa información y unas APIs (orientadas a objetos y orientadas a servicios) que permiten exponer los contenidos del catálogo[173].

[171] Cada meta-especificación EML es totalmente auto-contenida (§ 12.2.3)
[172] Addressing (en inglés)
[173] En el caso del catálogo podemos decir que se produce una sinergia interesante entre sus elementos. En este caso estamos en condiciones de afirmar que el todo es más que la suma de las partes.

En la figura 13.1 a continuación se muestra la arquitectura de alto nivel del catálogo. Esta visión de arquitectura incluye los componentes activos y pasivos.

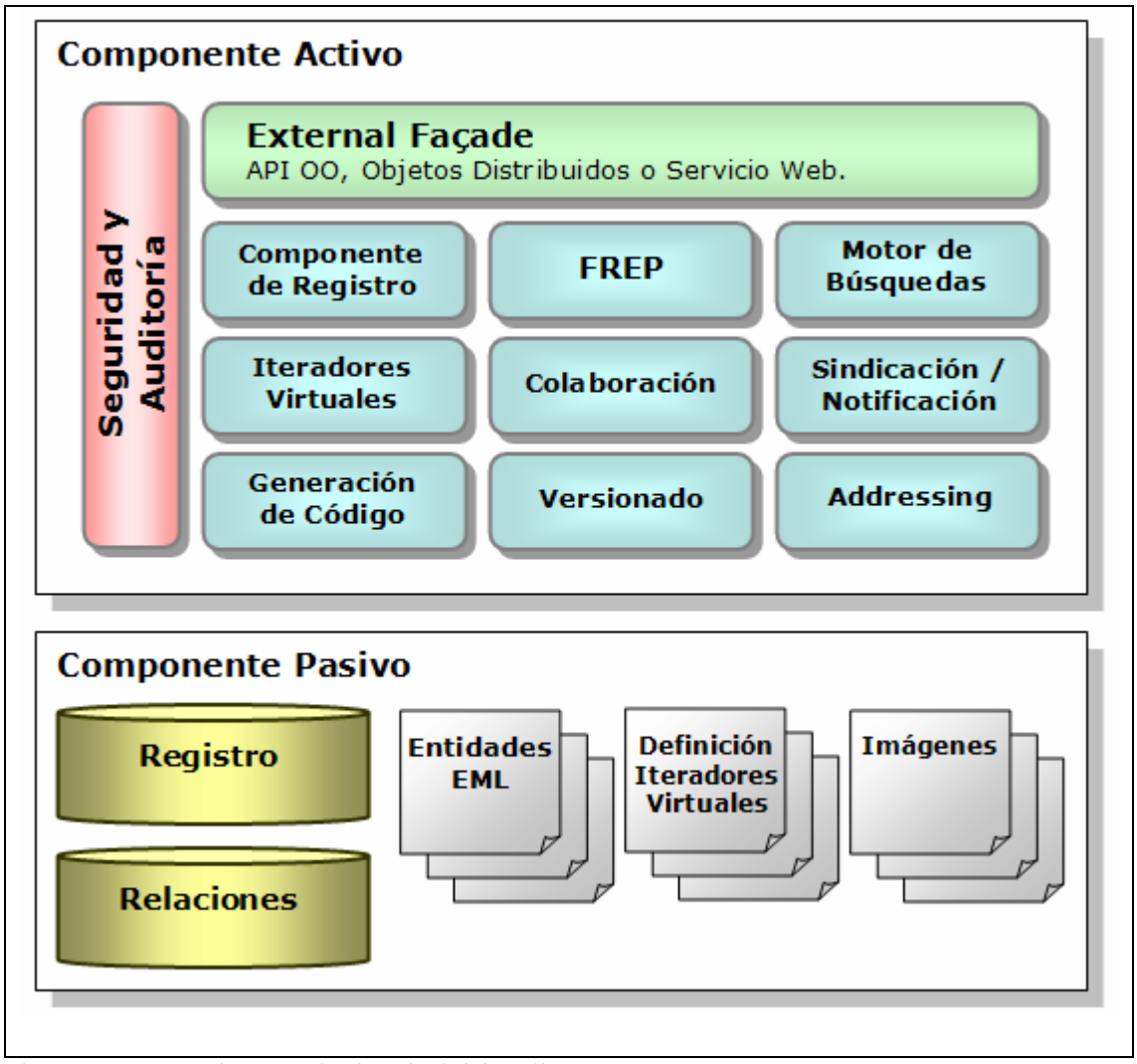


Figura 13.1 – Arquitectura de alto nivel del catálogo.

13.2.3 Premisas de Diseño

Para la arquitectura y diseño del catálogo hemos tomado como premisas y visión los siguientes conceptos de la magnifica obra de referencia [HT00][174]:

- *Poner la abstracciones en el código y los detalles en metadatos* ([HT00], pp. 145)
- *Separar las vistas de los modelos* ([HT00], pp. 161)
- *Configurar en lugar de integrar* ([HT00], pp. 144)
- *Las abstracciones viven más que los detalles* ([HT00], pp. 209)
- *Mantener el conocimiento en texto plano* ([HT00], pp. 74)

[174] "The Pragmatic Programmer: From Journeyman to Master", de Hunt y Thomas.

13.3 Componentes Pasivos

Los componentes pasivos son datos o información persistente que se encuentran almacenados en el catálogo y son consumidos o manipulados por los usuarios externos[175] a través de los componentes activos[176]. En esta sección presentaremos los distintos tipos de elementos pasivos que se almacenan en el catálogo.

13.3.1 Registro de Entidades EML

El registro de entidades es un repositorio donde se almacenan las meta-especificaciones EML de patrones y conceptos de soporte. Esta estructura brinda el soporte necesario para registrar las entidades para su posterior utilización.

13.3.1.1 Patrones

Meta-especificaciones de patrones descriptos utilizando EML (presentado anteriormente en los **capítulos 12** y **11** de esta tesis). Contienen toda la información sobre un patrón y las relaciones entre éste y otras entidades[177]. Las meta-especificaciones de patrones pueden estar incompletas. El catálogo tiene la habilidad de poder gestionar esta información incompleta y provee los mecanismos para completarla a posteriori[178].

El catálogo puede almacenar patrones pertenecientes a cualquier lenguaje o sistema de patrones[179] [Welicki05d]. Como ejemplos de patrones podemos mencionar los que existen en las siguientes obras: "*Design Patterns: Elements of Reusable Object Oriented Software*" [GoF95], "*Patterns of Enterprise Application Architecture*" [Fowler02], "*Evolving Frameworks*" [RJ96], "*Pattern Oriented Software Architecture, Volume 1*" [POSA96], entre otros.

13.3.1.2 Conceptos de Soporte

Descripción de conceptos de soporte[180] a los patrones utilizando EML (presentado en el **capítulo 12** esta tesis). Las meta-especificaciones de los conceptos de soporte pueden estar incompletas. El catálogo tiene la habilidad de poder gestionar esta información incompleta y provee los mecanismos para completarla a posteriori.

---

[175] Estos usuarios externos pueden ser personas u aplicaciones informáticas

[176] Los usuarios nunca acceden directamente a los elementos pasivos, sino que lo hace a través de los elementos activos.

[177] Estas entidades pueden ser otros patrones o cualquier tipo de concepto de soporte (refactorización, lenguaje de patrones, categorías, etc.)

[178] El análisis y manipulación de entidades incompletas y los mecanismos para manipularlas a priori son parte del componente activo del catálogo. La habilidad de representar una entidad incompleta es parte de la especificación de EML (capítulo 12).

[179] En el capítulo anterior hemos mencionado que EML puede representar patrones de cualquier lenguaje o sistema de patrones, debido al enfoque de composición dinámica de propiedades utilizado para definir la plantilla (template) del patrón.

[180] Los conceptos de soporte añaden información sobre el patrón, permitiendo comprenderlo realmente. Incluyen lenguajes de patrones, categorías, refactorizaciones, principios de orientación a objetos, etc.

El catálogo puede almacenar conceptos de cualquier tipo. Como ejemplos de conceptos de soporte podemos mencionar los lenguajes y sistemas de patrones, categorías, refactorizaciones, autores, libros, conceptos de orientación a objetos en que se basa un patrón, tipos de patrones, etc.

13.3.2 Imágenes

Imágenes[181] que pueden ser asociadas a los patrones o a los conceptos de soporte. Una única entidad (ya sea un patrón o concepto) puede tener asociada más de una imagen (o ninguna). A modo de ejemplo, una imagen puede ser el diagrama de clases UML[182] [UML] u OMT[183] [RBP91] de la estructura de un patrón.

Las imágenes se almacenan en los formatos estándar de representación de imágenes (gif [GIF], jpg [JPG], etc.), de forma tal que pueden ser consumidas en forma sencilla por un amplio espectro de clientes.

13.3.3 Definición de los Iteradores Virtuales

Los iteradores virtuales permiten realizar recorridos sobre las entidades del catálogo a partir de unos metadatos que indican el camino a seguir. Estos metadatos son elementos pasivos que se almacenan en el catálogo.

Se codifican utilizando un DSL (**capítulo 7**) textual sencillo basado en XML. Este lenguaje permite especificar los distintos niveles y las relaciones que utilizará el iterador para construir una jerarquía de entidades. La descripción completa de los iteradores virtuales se encuentra más adelante en este capítulo (§ 13.4.4).

13.3.4 Relaciones entre Entidades

Las relaciones existentes entre las entidades almacenadas en el catálogo son elementos pasivos que se almacenan en forma persistente. A partir de estas relaciones es posible moverse de una entidad a otra (o realizar inferencias y búsquedas contextuales). La capacidad relacional del catálogo es uno de los aspectos más importantes y con mayor peso. Es un gran diferenciador respecto a otros enfoques existentes[184]. Los conceptos en el catálogo forman una red semántica[185].

Las relaciones son la base para las siguientes funcionalidades avanzadas:

[181] Ficheros de gráficos, por ejemplo gif, jpg, etc.
[182] Unified Modelling Language
[183] Object Modelling Technique
[184] Por ejemplo, en el caso de los wikis, las relaciones son relaciones sintácticas, a través de URL. En este caso, estamos ante relaciones semánticas.
[185] Red Semántica: Grafos dirigidos donde los vértices son conceptos y los enlaces son relaciones entre conceptos [Labra04]

- **Red de conceptos**. Poder moverse de una entidad a otra fácilmente, brindando la infraestructura para crear una red de conceptos donde cada concepto añada significado a los conceptos con los que se relaciona. Por ejemplo, cuando estamos ante el patrón ABSTRACT FACTORY [GoF95], podemos acceder fácilmente al libro "*Design Patterns*" [GoF95], a la refactorización *Move Creation Knowledge to Factory* [Kerievsky04], a los *patrones creacionales* [GoF95], etc.

- **Iteradores virtuales**. Los árboles de navegación de entidades se construyen a partir de las relaciones almacenadas en el catálogo y de metadatos que indican como moverse a través de esas relaciones.

- **Búsquedas**. El buscador puede explotar estas relaciones para realizar búsquedas contextuales (por ejemplo: "*todos los patrones que conformen el principio de inversión de dependencias*", "*todos los patrones de la categoría creacional del libro Design Patterns*", etc.).

- **Inferencia**. Se pueden realizar inferencias a partir de las relaciones directas o transitivas existentes entre las entidades almacenadas en el catálogo.

Las relaciones se almacenan a partir de la descripción de la relaciones de una entidad en una meta-especificación EML (utilizando EML-RDL, § 12.2.4.2).

13.4 Componentes Activos

Los componentes activos contienen un amplio y comprehensivo conjunto de funcionalidades que permiten manipular y exponer los componentes pasivos del catálogo. Adicionalmente, incluyen funcionalidades avanzadas como gestión de la seguridad, auditorías y colaboración. En esta sección estudiaremos las funcionalidades ofrecidas por los componentes activos.

13.4.1 Interfaz de Registro de Entidades

El registro de una entidad se refiere a su inclusión o exclusión en el catálogo de entidades. Una entidad puede ser registrada (incluida[186]) o desregistrada (excluida). Una vez que una entidad ha sido registrada está lista para ser utilizada por los clientes del catálogo.

La interfaz de registro de entidades permite introducir, modificar o quitar alguna entidad (patrón o concepto) al catálogo. Esta interfaz manipula los elementos pasivos, añadiendo nueva información o modificando la existente a través de un componente creado a tales efectos. Este componente tiene un acoplamiento significativo con la estructura del catálogo, dado que es el único punto a través del cual pueden modificarse sus contenidos.

[186] Si no existe se incluye, si existe se actualiza

El registro de las entidades puede realizarse en tres modalidades:

- **Cliente Pesado**[187]: cliente pesado (puede ser WinForms [NET], Swing [SJFC], etc.) que permite interactuar con los contenidos del catálogo. Este cliente tiene un GUI[188] con funcionalidades de interacción avanzadas, como exponer en forma esquemática los contenidos del catálogo, verificar la entidad a registrar antes de introducirla en el catálogo, registrar y desregistrar entidades, etc.

- **Línea de Comandos**: interfaz de línea de comandos que permite registrar o desregistrar entidades mediante comandos sencillos. A diferencia del cliente anterior, no tiene capacidades avanzadas de interacción y visualización de información.

- **Servicio Web**: interfaz de servicios Web que permite registrar entidades en forma remota. Proporciona todos los beneficios de los servicios Web, entre los cabe destacar bajo acoplamiento, independencia de la plataforma, distribución, etc. [W3C]. Esta interfaz permite integrar la provisión de entidades en el catálogo con cualquier herramienta existente que tenga capacidades de utilizar servicios Web. Un caso de uso interesante es la integración con entornos de desarrollo como Eclipse [Eclipse] o Visual Studio .NET [VSNET].

Las tres modalidades de registro utilizan un mismo API para interactuar con el catálogo (expuesta por el componente de registro). Los detalles de presentación de cada interfaz de registro son gestionados mediante las distintas fachadas (siguiendo una variante del patrón FAÇADE [GoF95]).

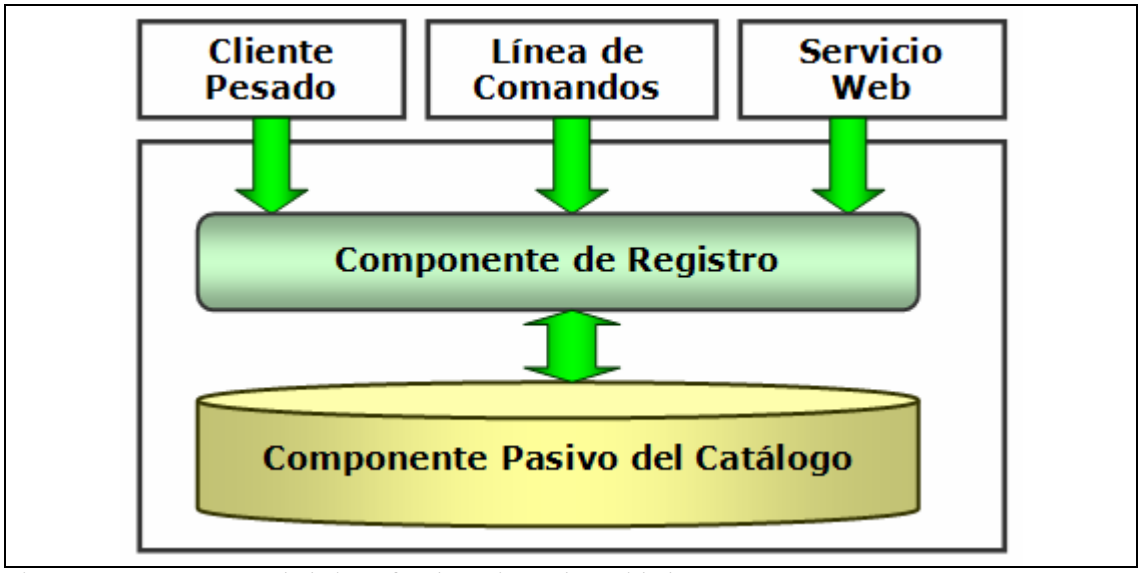


Figura 13.2 – Esquema de la interfaz de registro de entidades.

[187] Fat Client en inglés
[188] Graphic User Interface (Interfaz Gráfica de Usuario)

13.4.1.1 Soporte para el Gradiente Semántico

La interfaz de registro es uno de los componentes fundamentales para dar soporte al gradiente semántico, dado que es la que permite evolucionar las meta-especificaciones almacenadas en el catálogo.

13.4.1.2 Versionado de Entidades

La interfaz de registro permite interactuar con el sistema de versiones del catálogo, permitiendo versionar, modificar o eliminar versiones existentes de una entidad. La funcionalidad de versionado se presenta más adelante en este capítulo (§ 13.4.9).

13.4.1.3 Direccionamiento de Entidades en el Catálogo

Las entidades tienen un identificador único, que es el que permite luego hacer referencia a ellas para su posterior recuperación. El enfoque más sencillo para el direccionamiento es que el identificador puede coincidir con su nombre y puede ser determinado por el autor de la entidad. El problema con este enfoque es que tiene muchas posibilidades de producir colisiones en las direcciones. Por ejemplo, en el caso del lenguaje de patrones de "*Design Patterns: Elements of Reusable Object Oriented Software*" [GoF95] (conocido como "GoF") coincide con el grupo de autores de esta obra (conocido también como "GoF").

Por lo tanto, para el direccionamiento de las entidades proponemos las siguientes opciones:

- **Utilizar el identificador de la entidad como clave principal**. Al momento del registro (§ 13.4.1) el sistema informa de las colisiones y el autor de la entidad debe modificar el identificador hasta que éste sea único en el sistema[189]. A continuación se presentan ejemplos de direccionamiento para el patrón ABSTRACT FACTORY y varios elementos relacionados:
    - GoF.AbstractFactory (patrón)
    - GoF.Creational (categoría)
    - GoF (lenguaje de patrones)
    - DesignPatternsBook (libro)
    - GangOfFour (autores)
- **Utilizar el identificador y el tipo de la entidad como clave principal**. De este modo, se evitan las colisiones (si dos entidades del mismo tipo se llaman igual, es muy probable que sean la misma cosa). A continuación se presenta ejemplos de direccionamiento para el patrón ABSTRACT FACTORY y varios elementos relacionados:
    - pattern::AbstractFactory

[189] Este sistema es utilizado por populares sistemas de correo Web como Hotmail [Hotmail] y Yahoo [Yahoo]

- category::Creational
- patternLanguage::GoF
- book::DesignPatterns
- author::GoF

En caso de utilizar versionado (§ 13.4.9), el sistema de direccionamiento es el mismo, adjuntando como postfijo al identificador de cada entidad el número de versión, como se muestra en los ejemplos a continuación[190]:

- pattern::AbstractFactory,2
- category::Creational,1
- DesignPatternsBook,12 (libro)

El número que se encuentra luego de la coma es el número de versión. Al utilizar las direcciones especificadas arriba, el catálogo devolverá la versión adecuada de la entidad solicitada (los ejemplos utilizan los dos sistemas de direccionamiento propuestos previamente).

13.4.2 Flexible Runtime Execution Platform (FREP)

FREP es el acrónimo de *Flexible Runtime Execution Platform* y es la plataforma mediante la cual se analizan y comparten las entidades almacenadas en el catálogo. Contiene el analizador de EML[191] y la plataforma basada en un modelo de objetos adaptativo (**capítulo 8**) que aloja a las especificaciones EML en tiempo de ejecución. Estas especificaciones pueden ser compartidas en varias formas (objetos, objetos distribuidos o servicios) a través de las fachadas del catálogo.

FREP combina DSLs[192] y AOMs[193] para crear una solución flexible, extensible y ágil, que puede ser evolucionada en forma sencilla. Cada componente de la solución puede evolucionar en forma independiente[194], dado que el acoplamiento es mínimo y las reglas de composición de las instancias residen en metadatos. Los DSLs que conforman EML son analizados por un analizador que da como resultado una instancia de una entidad (en forma de grafo de objetos) basada en un AOM, que puede ser utilizada por los clientes del catálogo o por otros componentes activos.

La combinación de DSL y AOM permite crear una plataforma ágil, flexible y robusta [Welicki04] muy fácil de evolucionar [Welicki06]. Cada una de estas tecnologías permite gestionar en forma óptima su ámbito de incumbencia:

[190] Los dos primeros ejemplos utilizan el segundo sistema de direccionamiento. El último ejemplo utiliza el primer sistema de direccionamiento.
[191] El analizador obtiene como entrada una especificación EML y da como salida una instancia de un AOM.
[192] Domain Specific Languages, estudiados en detalle en el capítulo 7 de esta tesis
[193] Adaptive Object Model, estudiado en detalle en el capítulo 8 de esta tesis.
[194] Esta independencia tiene sus límites. Por ejemplo, si se modifica un DSL y esta modificación afecta al analizador, se deberá modificar la pieza de código que realice el análisis de esa sección del DSL para poder introducir los cambios en el sistema.

- Los DSLs nos permiten describir en forma precisa los elementos, utilizando lenguajes especializados en la descripción de un dominio específico y acotado.

- Los AOM nos permiten crear un modelo de objetos dinámico que varía en función de metadatos, permitiendo que la plataforma evolucione sin modificar su código fuente y evolucionar la plataforma añadiendo nuevos metadatos o nuevas piezas de código con responsabilidades bien definidas.

Por lo tanto, consideramos que hemos hecho una buena elección porque…

- Utilizamos la expresividad de los DSLs para describir a las entidades (a partir de descripciones puntuales de las partes que las componen)

- Utilizamos la flexibilidad, agilidad y potencia de AOM para ofrecer una plataforma de ejecución para las entidades

Esta combinación permite resolver un problema complejo que no ha sido resuelto previamente por ningún otro enfoque existente (ver **capítulo 10**): dar soporte a todas las plantillas existentes para describir patrones y permitir crear nuevas plantillas dinámicamente[195] manteniendo una estructura formal.

13.4.2.1 Implementación de AOM en FREP

El modelo de objetos adaptativo (AOM) diseñado es una adaptación con varias mejoras respecto al modelo original propuesto en [FY98] [YBJ01] [YJ02]. Las mejoras más significativas son la inclusión de un nivel abstracto de visualización y la utilización de inyección de dependencias [Fowler04] para poder "inyectar" módulos de código en tiempo de ejecución (estas inyecciones se realizan a partir de metadatos y reflectividad [POSA96] [Ortin02]).

La implementación propuesta se basa principalmente en la forma canónica de AOM [AOM] [YBJ01], principalmente en el patrón TYPE OBJECT PATTERN [Woolf97] y en una adaptación del patrón TYPE SQUARE [YBJ01] a nuestro dominio de problema. Contiene varios tipos de objetos "propiedad" orientados a representar los diferentes elementos que pueden conformar la plantilla de un patrón. La implementación de TYPE SQUARE esta combinada con el COMPOSITE [GoF95] para representar jerarquías de composiciones que pueden variar dinámicamente. Para gestionar las relaciones entre entidades hemos utilizado una variante del patrón ACCOUNTABILITY [Fowler96]. El modelo de vistas dinámicas (representado en la capa de visualización) utiliza una combinación de STRATEGY [GoF95] y SEPARATED INTERFACE [Fowler02].

[195] Estas nuevas plantillas se pueden crear en forma arbitraria. Por ejemplo, en el prototipo hemos creado una plantilla especial para describir patrones del GoF que incluye contenidos de otras fuentes y elementos de otras plantillas, con el objeto de aumentar la información sobre el patrón y hacer que sea más fácil de comprender.

En la figura 13.3 a continuación se presenta el AOM diseñado. En este modelo, se especifican tres niveles:

- **Nivel de Conocimiento**: registra las reglas generales que gobiernan la estructura [Fowler96]

- **Nivel Operacional**: registra la estructura del dominio [Fowler96]. En nuestro caso, la estructura se compone de entidades (patrones, fuentes de información y conceptos), propiedades (cadenas, imágenes, propiedades compuestas, etc.) y de relaciones entre entidades.

- **Nivel de Visualización**: contiene instrucciones primitivas sobre como mostrar cada elemento del modelo. Estas instrucciones pueden ser adjuntadas y compuestas en tiempo de ejecución (a partir de información de configuración). La implementación de los objetos de estos es responsabilidad de la capa de presentación (en este caso, del visor del catálogo que se presenta en el **capítulo 14**).

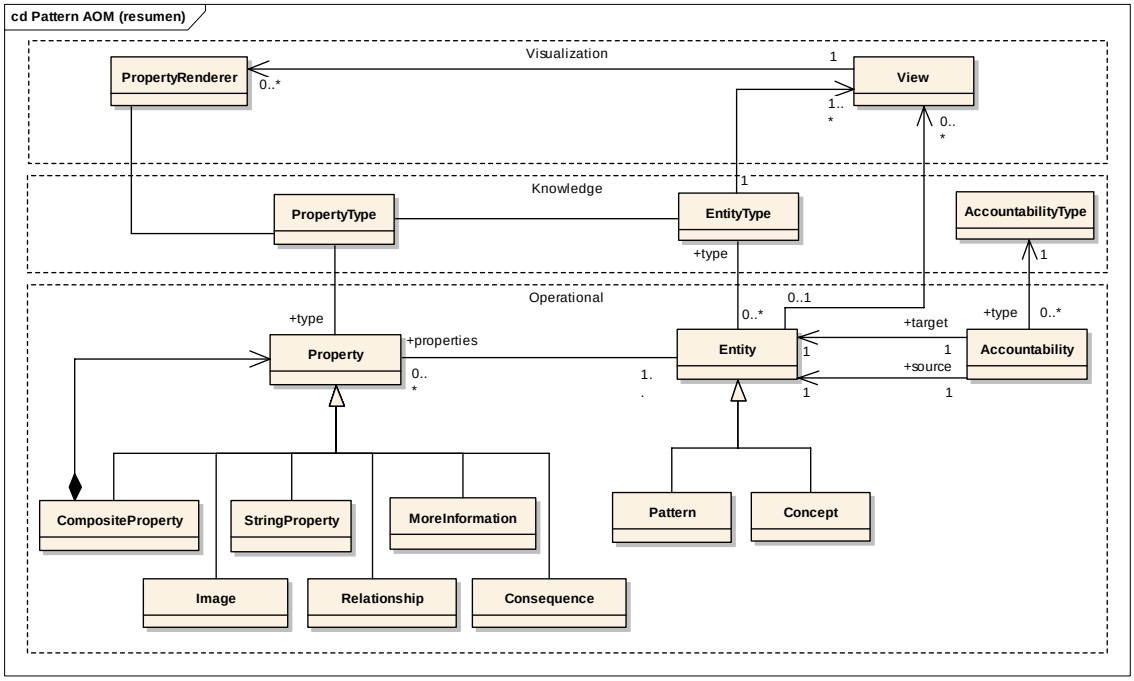


Figura 13.3 – Diseño general del AOM. Como puede observarse, no hemos incluido las estrategias (comportamiento dinámico), dado que hemos establecido el foco principal en el modelado de la estructura. De todos modos, existe una estrategia adaptable para la generación de código.

13.4.2.2 Analizador de EML

Para crear el analizador de EML y construir las instancias en tiempo de ejecución hemos utilizado una combinación de los patrones BUILDER, INTEPRETER [GoF95] y REGISTRY [Fowler02] sobre una arquitectura de plug-ins [Fowler02].

El analizador EML toma como entrada una meta-especificación EML de una entidad (elemento pasivo y rígido) y da como resultado una instancia de un AOM

(elemento activo y dinámico), como se muestra en la figura 13.4 a continuación. Para la validación de la corrección sintáctica de la entidad EML se utilizan XML Schemas [W3CXS]. Existe un esquema para cada uno de los DSLs que conforman EML.

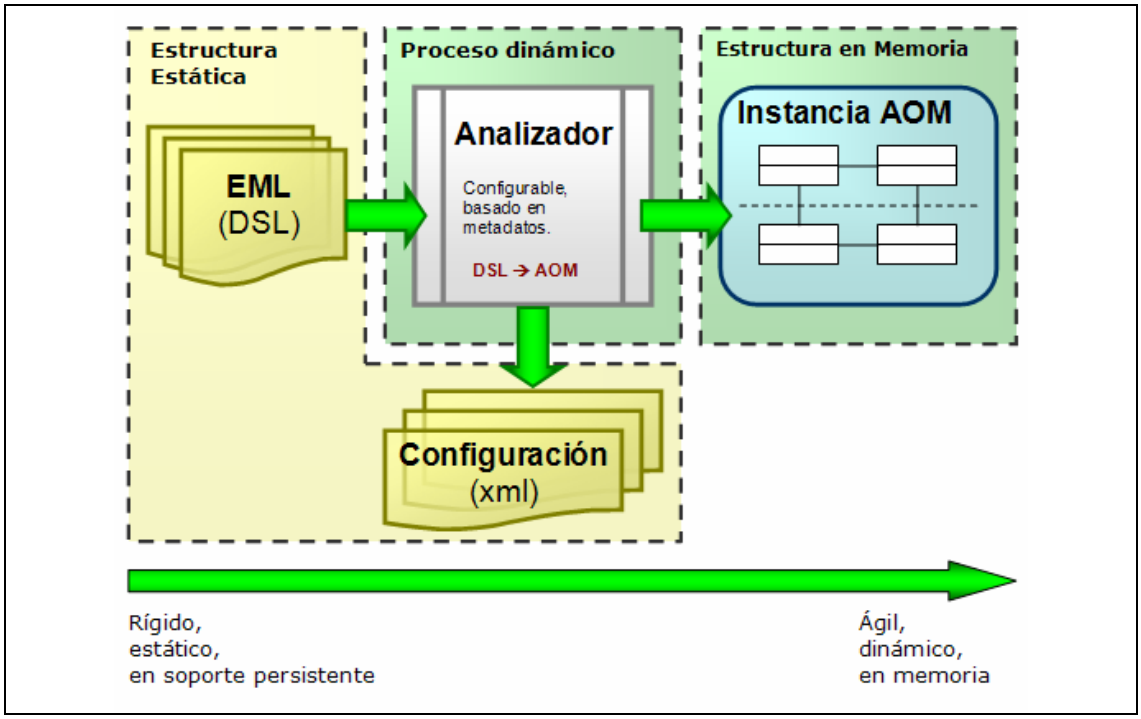


Figura 13.4 – Arquitectura general de alto nivel del analizador de EML

El análisis se realiza en varios pasos. En cada paso, se analiza un aspecto concreto de la entidad. El análisis se ejecuta mediante la implementación de un pipeline (similar al propuesto en el patrón PIPES AND FILTERS [POSA96]) donde en cada paso se realiza una porción del análisis bien definida. Una vez terminado el análisis del DSL y obtenido el AOM, éste puede ser utilizado en la forma que el usuario considere apropiada.

13.4.2.2.1 Configuración del Analizador

Los pasos se configuran en un fichero de metadatos y el pipeline completo se crea en tiempo de ejecución, utilizando reflectividad. En el fichero de configuración se establece para cada tipo de entidad los pasos que deben seguirse. El motor de análisis interpreta ese fichero y crea todos los objetos necesarios para realizar el análisis completo de la entidad.

En el siguiente bloque de código, se muestra un ejemplo de cómo configurar el análisis de patrones en el fichero de configuración del analizador:

```
<entityType id="Pattern">
  <parseStep type="LW.PGen.Parser.BasicDataParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.PropertiesParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.TagsParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.ContextParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.KnowledgeLevelParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.BaseImplementationParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.ConcreteImplementationParseStep, LW.PGen"/>
  <parseStep type="LW.PGen.Parser.DumpImplementations, LW.PGen"/>
</entityType>
```

Código 13.1 – Configuración del analizador de patrones EML.

En cada entrada del fichero de configuración se especifica un paso de análisis. Para cada paso, se especifica el tipo que lo implementa y el contenedor donde está alojado ese tipo. Con esta información el analizador puede crear el implementador del paso en tiempo de ejecución mediante reflectividad y añadirlo al pipeline de análisis.

13.4.2.3 Metadatos

FREP esta basado totalmente en metadatos, permitiendo modificar la mayor parte de sus componentes sin necesidad de modificar su código fuente. Todos los componentes de FREP se configuran a partir de metadatos, siguiendo la premisa "*Poner la abstracciones en el código y los detalles en metadatos*" ([HT00], pp. 145).

Tanto el analizador como los pasos de análisis están basados en metadatos. El mecanismo de inyección de dependencias y la composición del AOM también lo están. De esta forma, podemos modificar el comportamiento o la estructura del sistema en tiempo de ejecución sin la necesidad de programar [YJ02] o de realizar un nuevo despliegue.

13.4.2.4 Extensibilidad

El mecanismo de extensibilidad de FREP está basado en una arquitectura de plug-ins [Fowler02] extendida, fundada en el patrón de DEPENDENCY INJECTION[196] [Fowler04] y en los principios "*Dependency Inversion Principle*" [Martin02], "*Open Closed Principle*" [Martin02] [JCJO92] y "*Liskov Substitution Principle*" [Martin02]. La estructura de los componentes reside en metadatos que son interpretados en tiempo de ejecución.

Esta combinación da como un resultado una plataforma débilmente acoplada, ágil y flexible donde podemos:

- añadir elementos al sistema implementando interfaces sencillas y registrando componentes

[196] Inyección de Dependencias

- cambiar algún aspecto concreto del sistema en tiempo de ejecución modificando metadatos de configuración.

Para cada aspecto general existen interfaces abstractas que definen el contrato del aspecto y factorías dinámicas que pueden crear instancias de objetos en tiempo de ejecución que implementan estas interfaces (no es necesario enlazar en tiempo de compilación, dado que la creación se realiza mediante el uso de reflectividad). El acoplamiento entre las entidades en el sistema es abstracto, a través de interfaces. Los ficheros de configuración contienen la información necesaria para crear las instancias y estructuras de objetos mediante el uso del patrón DEPENDENCY INJECTION [Fowler04] para poder inyectar estos "plug-ins" al sistema en tiempo de ejecución.

Por lo tanto, para extender el sistema sólo es necesario implementar la interfaz adecuada y registrar la nueva clase en el fichero de metadatos adecuado.

13.4.3 Motor de Búsquedas

El motor de búsquedas permite realizar distintos tipos de búsquedas sobre los elementos pasivos almacenados en el catálogo (§ 13.3). Estas búsquedas tienen como entrada un conjunto de parámetros y retornan un conjunto de entidades como resultado.

Pueden realizarse varios tipos de búsqueda diferentes, los cuales se detallan a continuación:

- **Coincidencia de Sartas**: dada una cadena, buscar las coincidencias totales o parciales de ésta en todos los bloques de texto que componen las especificaciones de los patrones y demás entidades almacenadas en el catálogo. El resultado de esta búsqueda son todas las entidades que tengan alguna sarta que coincida total o parcialmente con la sarta enviada como parámetro. Este tipo de búsqueda es muy popular especialmente en los motores de indexación elementales (como *Index Server* [Microsoft] o Google Search Appliance [GSA]) o en las aplicaciones empresariales [Fowler02].

- **Búsqueda por Etiquetas**: buscar las entidades a partir de etiquetas[197]. Las entidades están decoradas con etiquetas y este tipo de búsqueda utiliza el contenido de estas etiquetas para obtener el conjunto de resultados. Este tipo de búsqueda es muy popular actualmente en la Web, siendo sus exponentes principales Flickr [Flickr], Technoratii [Technoratti] y Del.icio.us [Delicious]

- **Búsqueda Contextual**: utilizar las relaciones existentes entre entidades para armar el conjunto de resultados. Con esta búsqueda, podemos explotar las relaciones almacenadas en el catálogo. Este tipo de búsqueda es muy popular en las aplicaciones empresariales (mediante la explotación de bases de datos

[197] Tags (en inglés)

relacionales normalizadas) y en sistemas expertos, entre otros. Anteriormente hemos mencionado la habilidad del catálogo para tratar con especificaciones incompletas. La búsqueda contextual no es una excepción a esta regla y puede gestionar relaciones "rotas" (es decir, relaciones donde uno de los extremos no está registrado en el catálogo, es incorrecto o está incompleto)

- **Inferencia**: realizar inferencias semánticas a partir de la ontología de entidades de ingeniería del software en que se funda el catálogo. Este tipo de búsquedas es popular en proyectos de Web semántica [DOS03] [BernersLee00], en sistemas expertos y en varios dominios de la inteligencia artificial.

Como puede fácilmente observarse, los tipos de búsqueda están directamente relacionados con el gradiente semántico (§ 12.2.5).

El usuario puede elegir que método de búsqueda desea emplear. Según el caso seleccionado, varía la forma en que se especifican los parámetros de entrada. En todos los casos el resultado es el mismo: un conjunto de entidades que coinciden con los criterios de búsqueda introducidos por el usuario.

13.4.4 Iteradores Virtuales

El patrón ITERATOR[198] [GoF95] "proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna". Los iteradores virtuales toman como punto de partida este patrón y lo extienden, añadiendo las siguientes características:

- La estrategia (algoritmo) de recorrido se encuentra definido en metadatos. Para crear un nuevo iterador sólo es necesario añadir información en el fichero de definición de los iteradores virtuales. El algoritmo no está en el código fuente, sino en reglas codificadas en metadatos.

- Posibilidad de recorrer estructuras jerárquicas con "n" niveles de anidamiento (similar a un COMPOSITE [GoF95])

[198] "Iterator" en el texto original

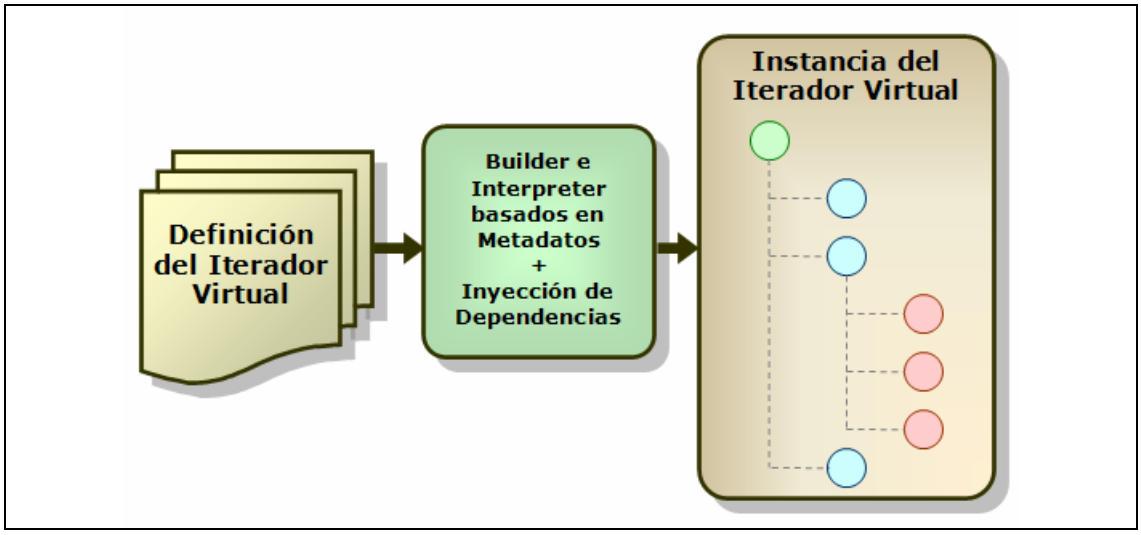


Figura 13.5 – Arquitectura de alto nivel de los Iteradores Virtuales

Los iteradores virtuales utilizan metadatos para construir la estrategia de iteración. A partir de estos metadatos construyen un árbol de navegación que puede tener una cantidad arbitraria de niveles. Esta cantidad esta determinada por los metadatos que definen al iterador.

13.4.4.1 DSL de Definición de Iteradores Virtuales

Los iteradores virtuales se definen en un fichero (que se aloja en la sección pasiva del catálogo). En este fichero, hay una entrada por cada iterador, donde se definen las reglas que configuran el algoritmo de navegación.

Para definir los iteradores virtuales hemos creado un DSL que permite especificar en forma declarativa la estrategia de composición de datos que utilizará el iterador. Para definir un nuevo iterador es necesario crear una entrada en el documento de definición y dentro de ésta un elemento para definir el comportamiento de cada nivel (qué tipo de entidades gestionará y qué tipo de relaciones entre entidades utilizará para realizar la iteración). En el bloque de código a continuación se muestra un ejemplo de cómo definir un iterador para recorrer lenguajes de patrones, categorías y patrones.

```
<iterator name="Pattern Language">
  <level
    id="1"
    entityTypeId="PatternLanguage"
    patternRelationshipTypeId="isContainedIn"/>
  <level
    id="2"
    entityTypeId="Category"
    relationshipTypeId="isContainedIn"
    patternRelationshipTypeId="isContainedIn"/>
</iterator>
```

Código 13.2 – Configuración de un iterador virtual.

El resultado de esta definición es un iterador que presenta en el primer nivel todos los lenguajes de patrones y en el segundo nivel todas las categorías de estos lenguajes con los patrones contenidos en éstas. El resultado de la ejecución de este iterador es una estructura como la que se muestra en la figura 13.6 a continuación[199].

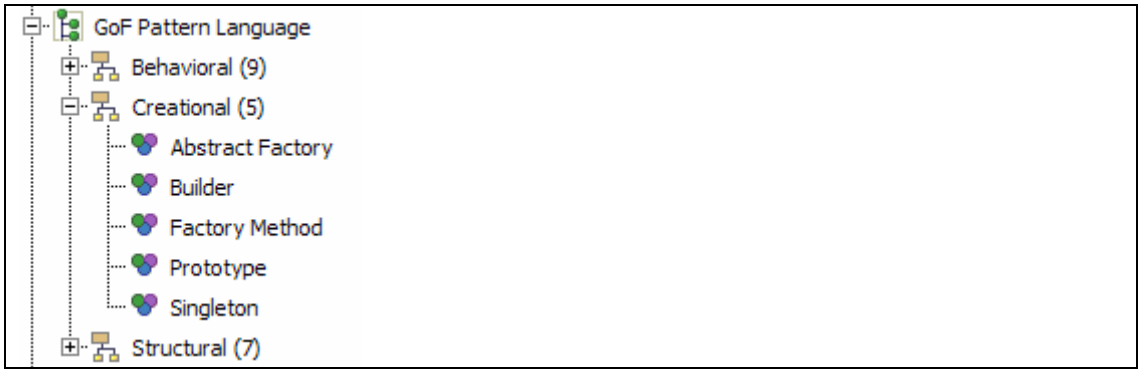


Figura 13.6 – Fragmento de UI ejemplo de la ejecución de una definición de un iterador virtual.

13.4.5 Seguridad y Auditoría

La seguridad y auditoría son aspectos transversales presentes en toda la capa[200] (patrón LAYERS [POSA96]) de comportamiento activo. El sistema de seguridad se encarga de la autenticación y autorización de las solicitudes realizadas al catálogo.

- **Autenticación**: verificar que un usuario es quién dice ser. Una vez autenticado en el sistema, podrá realizar todas las actividades que sus privilegios le permitan (autorización)

- **Autorización**: verificar que un usuario puede realizar una acción sobre una entidad almacenada en el catálogo.

Para la gestión de la seguridad se utilizan usuarios y grupos de usuarios. Pueden asignarse permiso a ambos elementos, aunque siempre es más recomendable asignar los permisos a nivel de grupo para facilitar la gestión de la seguridad[201].

Si bien gran parte de los contenidos del catálogo son públicos, el sistema de verifica la información del usuario en los siguientes casos:

- **Modificación de elementos del catálogo**: cualquier modificación sobre la información del catálogo requiere que el usuario este autenticado y tenga la autorización necesaria para realizar dicha modificación. Al momento de modificar un elemento del catálogo (por ejemplo, los elementos pasivos) el sistema verifica que el usuario que intenta realizar la modificación tenga los

[199] En la figura se muestra sólo un fragmento del árbol resultante de la ejecución del iterador virtual.
[200] Layer en inglés (del patrón Layers [POSA96])
[201] Al trabajar con grupos, sólo es necesario añadir un usuario a un grupo para que este tenga una serie de privilegios asociados al grupo y no es necesario dar permisos puntuales sobre cada recurso a cada usuario.

privilegios necesarios para hacerlo. Al realizar la modificación el sistema almacena los datos del usuario que ha realizado la modificación (auditoría).

- **Colaboración**: las funcionalidades de colaboración requieren que el usuario esté autenticado y que tenga los privilegios adecuados para participar en dicha colaboración. En cada interacción con el módulo de colaboración el sistema registra la información del usuario.

- **Auditoría**: ante cada acceso o interacción con el catálogo el sistema registra la información del agente que ha realizado dicha interacción. El sistema recauda constantemente información sobre los usuarios que lo utilizan. En el caso de los contenidos públicos (a los que el usuario accede sin autenticarse) se almacena información general sobre la petición, a efectos de poder realizar estadísticas. Esto permite no sólo detectar posibles anomalías sino también detectar pautas de uso, disparar notificaciones, habilitar el proceso de colaboración, etc.

Como puede observarse, la auditoría y la seguridad tienen una relación muy estrecha. El sistema recolecta constantemente información sobre los agentes que realizan peticiones. En caso de que dichos agentes estén autenticados, el sistema registra información sobre la identidad del agente, a partir de los datos almacenados en el sistema de seguridad.

13.4.6 Comunidad y Colaboración

El objetivo de estos componentes es habilitar espacios donde los miembros de la comunidad puedan interactuar y evolucionar el conocimiento almacenado en el catálogo. Este conocimiento puede aumentarse utilizando los siguientes mecanismos:

- **Foros de Discusión**: crear espacios de discusión donde los usuarios puedan intercambiar opiniones e información sobre aspectos concretos de entidades del catálogo. Para cada entidad se crea automáticamente un foro multi-hilo.

- **Directorio de Miembros**: lista con todos los miembros de la comunidad y su información de contacto. Este directorio está integrado con los módulos de seguridad y auditoría.

- **Mapa de Conocimiento**: El mapa de conocimiento permite aumentar la información del directorio, indicando cuales son las áreas de especialización y experiencia de cada uno de los miembros del directorio. De esta forma, ante una necesidad concreta es posible identificar a la persona indicada para resolverla. Los mapas de conocimiento se presentan en detalle en [DP99]. Una aplicación de estas herramientas a entornos de software se presenta en [Welicki03]

En la actualidad existe un espacio colaborativo que goza de gran popularidad (un wiki, § 10.1.1) donde se puede trabajar sobre los patrones [PPR]. Por lo tanto, consideramos que no es conveniente replicar este esfuerzo (que ha sido prolongado y constante a través de una década) y que vale la pena reaprovecharlo: en consecuencia, las entidades del catálogo están vinculadas a este entorno siempre que sea posible (esto es, que exista un espacio creado para esa entidad).

Los mecanismos de comunidad y colaboración contribuyen a transmitir al usuario la percepción de que el catálogo es un "espacio vivo de información".

13.4.7 Sindicación y Suscripción

Los clientes del catálogo deben poder suscribirse para ser notificados de los cambios y novedades sobre las entidades en que estén interesados. De esta forma, el catálogo puede informar a los usuarios sobre estos cambios, evitando que éstos deban acceder constantemente a verificar si hay nuevos contenidos. Este paradigma de interacción que siempre fue popular se ha hecho más popular aún con el advenimiento de los *blogs*. En la actualidad, la gran mayoría de los blogs tienen mecanismos de sindicación de contenidos, siendo el más popular RSS [RSS]. Este mecanismo se ha extendido a otro tipo de sitios y es común encontrarlo en cualquier portal de contenidos como por ejemplo los periódicos on-line ([ElMundo] [ElPais] [Clarin]), portales de tecnología vertical [TSN] [TSS] o sitios de noticias de tecnología [Slashdot], entre otros.

El catálogo ofrece las siguientes opciones de notificación y suscripción a los usuarios:

- **Sindicación vía RSS**: sindicación de contenidos a partir de feeds RSS. Los usuarios pueden enterarse de los cambios a partir de agregadores RSS[202] [RssReader], [SharpReader]

- **Notificaciones vía mail**: posibilidad de suscribirse a elementos lógicos concretos. Ante cambios en esos elementos, el catálogo informa a los usuarios con un correo electrónico.

La suscripción y notificación a contenidos es una forma de hacer "push" de información y contribuye a transmitir al usuario la percepción de que el catálogo es un "espacio vivo de información".

13.4.8 Fachada para Exponer las Entidades

Al inicio de este capítulo hemos mencionado que el catálogo dispone de funcionalidades para exponer sus contenidos. Las instancias del AOM, los

[202] Existe una amplia variedad de agregadores RSS gratuitos

elementos de comunidad y los iteradores virtuales son expuestos a través de una fachada (patrón Façade [GoF95]) externa.

Dentro de la fachada hay tres variantes, que se detallan a continuación:

- **API OO**: compartir los objetos "in-process", a partir de incluir en el una referencia a las librerías de componentes catálogo. Para esto es necesario tener conectividad directa (física) con los componentes activos del catálogo.

- **Objetos Distribuidos**: compartir los objetos a través de un mecanismo de distribución "out-of process", como por ejemplo CORBA [CORBA], .NET Remoting [NETRemoting], DCOM [Microsoft], etc.

- **Servicios Web**: compartir los objetos a través de una interfaz de servicios Web. En este caso, la complejidad está en la serialización del AOM. De esta forma, los contenidos del catálogo pueden ser utilizados desde entornos de desarrollo como Eclipse o Visual Studio .NET (entre otros), herramientas de modelado o suites de ofimática (para documentación de procesos). Esta interfaz puede contribuir también a la creación de "aplicaciones virtuales", como se propone en [Welicki05c]

13.4.9 Versionado de Entidades

El catálogo permite almacenar distintas versiones de una misma entidad EML. Para identificar una versión, se adjunta un número entero al identificador de la entidad. Dado que las meta-especificaciones EML se almacenan en texto plano (recordar que EML es un DSL textual basado en XML, § 12.2), la estrategia de versionado de versiones utilizada es distinta a la clásica (basada en deltas que se recomponen cuando se solicita una versión concreta): cada versión se almacena entera, pero comprimida[203] utilizando algoritmos estándar, como por ejemplo zip [ZIP] y se descomprime bajo demanda al momento de utilizarla. De esta forma, tenemos como resultado un sistema sencillo de implementar y que hace una buena utilización de los recursos de almacenamiento (y que está alineado con la premisa de "mantener el conocimiento en texto plano", presentada al inicio de este capítulo).

13.4.10 Generación de Código

EML incluye un DSL que permite describir el comportamiento de una entidad a un alto nivel de abstracción (EML-BDL, § 12.2.4.5) y otro que permite describir la estructura de una entidad (EML-SDL, § 12.2.4.4). La generación de código se realiza a partir de la traducción de EML-BDL a un lenguaje de programación de alto nivel. Los traductores deben estar registrados en el sistema. El usuario puede seleccionar en tiempo de ejecución qué traductor desea utilizar. En la figura 13.7 a continuación se muestra la arquitectura general del motor de generación de código.

[203] Los algoritmos de compresión brindan un rendimiento excelente cuando se trabaja con texto plano.

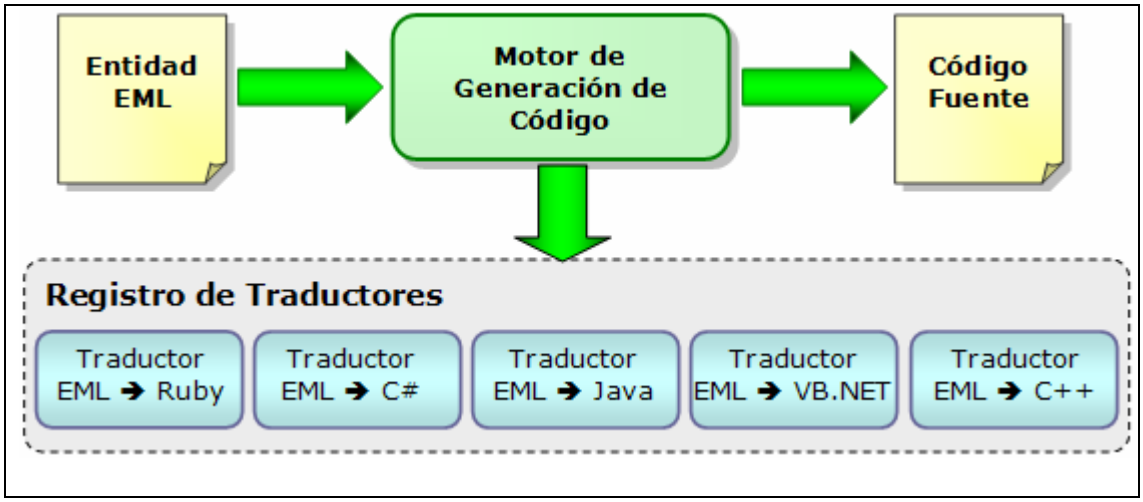


Figura 13.7 – Arquitectura del módulo de generación de código (traducción de EML).

El sistema de generación puede ser extendido añadiendo nuevos traductores de EML a lenguajes de alto nivel. De esta forma el motor de generación puede ser extendido sin comprometer el comportamiento existente. Esta arquitectura cumple con las premisas establecidas en § 13.2.3 y está basada en los principios de orientación a objetos "*Dependency Inversion Principle*", "*Open Closed Principle*", "*Liskov Substitution Principle*" [Martin02]. De esta forma, los usuarios del catálogo pueden obtener directamente código fuente a partir de la definición de un patrón.

13.4.10.1 Ajuste de los Resultados

Las especificaciones descritas utilizando los DSLs EML-BDL (§ 12.2.4.5) y EML-SDL (§ 12.2.4.4) pueden ser modificadas en tiempo de ejecución (en forma no persistente) por los usuarios, a efectos de ajustar los resultados a obtener al momento de realizar la traducción. De esta forma, se puede adaptar el resultado a obtener en la generación en función de los trade-offs y contexto de diseño para el que se desee generar dicho código (se modifican aspectos de la entidad antes de enviarla al motor de generación).

Esto también brinda soporte para comprender mejor al patrón, pudiendo estudiar distintas implementaciones del mismo concepto a partir de ajustes en la estructura original (recordar que "*un patrón puede aplicarse un millón de veces y obtener distintas soluciones*" [Alexander79]).

13.5 Arquitectura de Despliegue del Registro

Debido a su naturaleza, el catálogo puede implementarse sobre una arquitectura distribuida. Una de las ventajas de este tipo de arquitectura es que se puede ofrecer alta disponibilidad en cada elemento que la conforma.

A continuación se enumeran las principales opciones de implantación del componente pasivo del catálogo:

- **Un conjunto de ficheros**: almacenar las especificaciones, imágenes, el registro y todas las estructuras necesarias para cumplir los requisitos de catalogación en ficheros[204].

- **Una base de datos**: almacenar las especificaciones, el registro y todas las estructuras necesarias para cumplir los requisitos de catalogación en una base de datos.

- **Base de Datos y Ficheros**: almacenar las especificaciones en ficheros planos y el registro y todas las estructuras necesarias para cumplir los requisitos de catalogación en base de datos[205].

- **Base de Datos, Base de Datos RDF y Ficheros**: almacenar las especificaciones en ficheros planos y el registro y todas las estructuras necesarias para cumplir los requisitos de catalogación en base de datos y las relaciones y metadatos de búsquedas de las entidades en una base de datos RDF.

Consideramos que la tercera y cuarta opción son las más adecuadas para implementar una solución flexible y robusta, dado que ofrecen mejor rendimiento, flexibilidad y escalabilidad.

Para el componente activo, se puede utilizar cualquier servidor de aplicaciones comercial[206] de los que existen en la actualidad (como por ejemplo J2EE [J2EE] o .NET [NET]).

Las arquitecturas distribuidas (como las propuestas en los ítems 3 y 4 de la lista anterior) pueden ser monitorizadas utilizando una combinación de agentes y servicios Web como se propone en [Welicki05b]. De esta forma podemos obtener máxima fiabilidad e incluso anticiparnos a posibles situaciones anómalas y no deseadas [Welicki05b].

13.6 Conclusión: Revisión de los Requisitos

El catálogo que hemos propuesto en este capítulo cumple con todos los requisitos que hemos establecido al inicio de este capítulo. En la tabla 13.1 a continuación, verificaremos la forma en que se realiza cada uno de los requisitos planteados al inicio de este capítulo. Para cada requisito se incluye una frase breve que indica como se realiza en el sistema, con una referencia a la sección donde es presentado en este capítulo

[204] Esta fue la primera opción utilizada durante los prototipos iniciales de nuestra investigación [Welicki04c] [Welicki05].
[205] Este es el enfoque utilizado en el prototipo creado durante la investigación y presentado en los capítulos 15 y 16 de esta tesis.
[206] En el caso del prototipo, hemos utilizado los servicios proporcionados por Microsoft .NET [NET]

Requisito	Descripción	Realización
RC-1	Soporte para almacenar entidades	El componente pasivo del catálogo puede almacenar entidades EML (§ 13.3.1). Estas entidades incluyen patrones (§ 13.3.1.1) y conceptos (§ 13.3.1.2)
RC-2	Soporte para almacenar relaciones	El catálogo puede almacenar relaciones entre entidades (§ 13.3.4).
RC-3	Analizador de entidades EML	El componente activo del catálogo (en concreto FREP, § 13.4.2) incluye un analizador de entidades EML (§ 13.4.2.2).
RC-4	Registro de entidades	La interfaz de registro (§ 13.4.1) permite añadir, modificar o quitar entidades del catálogo.
RC-5	Soporte para búsquedas	El componente activo del catálogo tiene un motor de búsquedas (§ 13.4.3). Existen distintas modalidades de búsquedas (coincidencia de sartas, etiquetas, contextual e inferencia) en función de la precisión que se busque en los resultados.
RC-6	Capacidad para manipular especificaciones incompletas	El catálogo puede manipular especificaciones incompletas. La interfaz de registro permite registrar entidades incompletas y completarlas a posteriori (§ 13.4.1). El motor de búsqueda puede tratar relaciones incompletas (§ 13.4.3).
RC-7	Exponer las entidades mediante un API OO	El catálogo expone sus contenidos mediante fachadas (§ 13.4.8). Estas fachadas incluyen la posibilidad de compartir objetos, ya sean locales o distribuidos.
RC-8	Exponer las entidades mediante servicios Web	El catálogo expone sus contenidos mediante fachadas (§ 13.4.8). Estas fachadas incluyen la posibilidad de compartir los contenidos del catálogo a través de servicios Web.
RC-9	Direccionamiento de entidades	Las entidades en el catálogo tienen un identificador único (§ 13.3.1.3)
RC-10	El catálogo debe ser un "espacio de información vivo"	El catálogo incluye mecanismos para la gestión de comunidad y colaboración (§ 13.4.6) y posibilidad de suscribirse o sindicar sus contenidos (§ 13.4.7)
RC-11	Generación de código	El catálogo incluye un motor extensible de generación de código a partir de meta-especificaciones EML (§ 13.4.10)

Tabla 13.1 – Revisión del cumplimiento de los requisitos del catálogo.

Capítulo 14
El Visor del Catálogo

"Una red de notas con vínculos (como referencias)
entre ellas es mucho más útil que
un sistema jerárquico fijo"
Tim Berners-Lee

En el **capítulo 11** hemos realizado un análisis y argumentación del contexto de la solución (§ 11.1) al problema planteado al inicio de esta tesis (§ 1.2). En esa misma sección hemos establecido por deducción la necesidad de crear una infraestructura de catalogación y hemos determinado una serie de requisitos generales para ésta[207]. Más adelante en ese capítulo, presentamos brevemente el visor del catálogo como solución de visualización de los contenidos y funcionalidades del catálogo (**capítulo 13**)

El visor del catálogo (*PatternsBrowser*) es una aplicación Web [Cueva03] [Pressman02] que permite navegar por los contenidos del catálogo (presentado en el **capítulo 13**). Es la capa de presentación de la arquitectura general (presentada en el **capítulo 11**) y permite interactuar con los elementos del catálogo.

Permite a los usuarios interactuar con las entidades y funcionalidades que ofrece el catálogo en forma sencilla, siguiendo estándares de usabilidad e interacción. Al ser una aplicación Web puede ser utilizada desde cualquier ordenador a través de un navegador de Internet

En la primera sección del capítulo se establecen los requisitos detallados para el visor del catálogo. A continuación se describe *Patterns Browser*, que es el catálogo que hemos diseñado en esta tesis y que satisface esos requisitos. Finalmente, a modo de conclusión revisaremos los requisitos establecidos al inicio, contrastándolos con todo lo expuesto a lo largo del capítulo.

❖ ❖ ❖

[207] En ese mismo capítulo, prometimos ampliar esos requisitos generales. Los requisitos ampliados se presentan en este capítulo.

14.1 Requisitos para el Visor del Catálogo

En el **capítulo 11** presentamos en forma resumida y coloquial los requisitos para la construcción de un visor para el catálogo de meta-especificaciones de patrones y conceptos de ingeniería del software (§ 11.1.13), que se establecen a partir de los problemas recurrentes detectados en los enfoques existentes en el **capítulo 10** (§ 10.4). A continuación se refinan esos requisitos y se presentan en forma detallada en la siguiente lista

RV-1. **Multiplataforma:** poder ser utilizado en plataformas heterogéneas.

RV-2. **Accesible a través de la Web:** estar disponible a través de la Web. De esta manera, no existe la limitación geográfica y de distribuciones.

RV-3. **Usabilidad:** el catálogo debe ser fácil de utilizar.

RV-4. **Múltiples vistas de la información:** exponer múltiples vistas de los patrones del catálogo. Para cada patrón, por ejemplo, mostrar las vistas de conocimiento, de código fuente, etc.

RV-5. **Navegabilidad:** permitir navegar en forma sencilla y eficiente por los patrones del catálogo. Poder explotar las relaciones entre los patrones.

RV-6. **Facilidades de búsqueda:** poder buscar patrones a través de distintas técnicas de búsqueda, como por ejemplo pattern-matching, inferencia a través de los meta-datos, etc.

RV-7. **Comunidad:** ofrecer infraestructuras a los usuarios para generar comunidad para promover la evolución del conocimiento almacenado y generación de nuevo conocimiento a partir del existente.

RV-8. **Personalización de resultados:** los niveles de conocimiento e implementación deben poder ser ajustados por el usuario: los patrones son un punto de partida. Por ejemplo, ajustando detalles de implementación se puede llegar a soluciones diferentes a partir de un mismo patrón. Ajustando el nivel de conocimiento se puede expresar mejor una idea en un contexto determinado.

14.2 El Visor del Catálogo

El visor del catálogo es una aplicación Web [Cueva03] que expone los contenidos del catálogo. Permite a los usuarios interactuar con las entidades y funcionalidades que ofrece el catálogo en forma sencilla, siguiendo estándares de usabilidad e

interacción. Al ser una aplicación Web puede ser utilizada desde cualquier ordenador a través de un navegador de Internet[208].

14.2.1 ¿Qué Ofrece el Visor del Catálogo?

El visor del catálogo expone los contenidos del catálogo al mundo a través de una interfaz gráfica Web, ofreciendo los siguientes servicios:

- Interfaz gráfica para navegar por el catálogo.
- Múltiples vistas sobre una entidad.
- Múltiples posibilidades de navegación de los contenidos (jerárquica, árboles hiperbólicos y listas de elementos).
- Buscador.
- Ambientes de colaboración (foros, debates, edición colaborativa, etc.).

Las funcionalidades que ofrece el visor se construyen a partir de composiciones de los servicios básicos que ofrece el catálogo a través de sus componentes activos (§ 13.4).

14.2.2 Arquitectura General

En la figura 14.1 a continuación se presenta un gráfico de arquitectura lógica general de alto nivel del visor del catálogo.

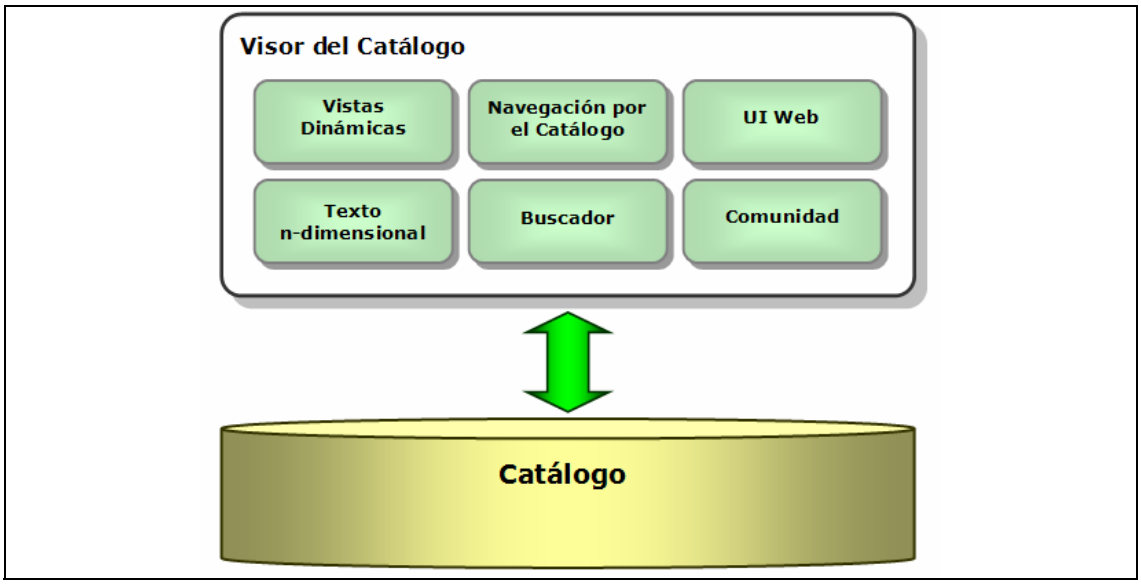


Figura 14.1 – Arquitectura funcional de alto nivel del visor del catálogo.

Como puede fácilmente observarse, el visor es una capa (patrón LAYERS [POSA96]) de presentación que expone los contenidos y funcionalidades del catálogo a través de la Web.

[208] Por ejemplo Internet Explorer, Firefox, Opera, etc.

14.2.3 Premisas de Diseño

Para la arquitectura y diseño del catálogo hemos tomado como premisas y visión general los siguientes conceptos de la magnifica obra de referencia[209] [HT00]:

- *DRY - Don't repeat yourself* ([HT00], pp. 27)
- *Separar las vistas de los modelos* ([HT00], pp. 161)
- *Configurar en lugar de integrar* ([HT00], pp. 144)

14.3 La Interfaz de usuario Web

Utilizamos como premisa básica de interacción la que postula Steve Krug en su libro "No me hagas pensar" [Krug99]: intentar que el usuario pueda manejarse fácilmente por la aplicación Web, sin necesidad de aprender pautas complejas de interacción y dejándose llevar por el sentido común[210].

Para el diseño del sitio hemos seguido las recomendaciones de Steve Krug sobre navegación [Krug99], de Jakob Nielsen sobre usabilidad [Nielsen99], de Alan Cooper sobre interacción [Cooper99] y de 37Signals sobre gestión de situaciones anómalas (errores y excepciones) [37Signals04]. El resultado es una aplicación sencilla y fácil de navegar, que ofrece funcionalidades avanzadas a los usuarios a través de un conjunto reducido de opciones[211].

14.3.1 Diseño Basado en Heurísticas

La evaluación heurística [Nielsen90], [Nielsen94], [Nielsen94b] es una variante de la inspección de usabilidad donde los especialistas en usabilidad juzgan si cada elemento de la interfaz de usuario sigue los principios de usabilidad establecidos. Este método forma parte del que se conoce como "ingeniería de la usabilidad rebajada"[212] [Nielsen94c]. Esta evaluación detecta aproximadamente el 42% de los problemas graves de diseño y el 32% de los problemas menores, dependiendo del número de evaluadores que revisen el sitio [Nielsen90].

La evaluación heurística permite evaluar sitios que ya han sido construidos y por lo tanto, funciona en forma reactiva. En este trabajo proponemos la utilización de las heurísticas como guía de diseño, utilizándolas para establecer una visión y verificando su cumplimiento constantemente[213].

---

[209] "The Pragmatic Programmer: From Journeyman to Master", de Hunt y Thomas.
[210] Las páginas deben ser obvias, evidentes, claras y fáciles de entender [Krug99]
[211] En el capítulo 17 se evalúa esta herramienta, comparándola con más de una decena de herramientas existentes.
[212] "discount usability engineering"en el texto original
[213] Podríamos establecer un paralelismo artificial entre este concepto y el Test Driven Development (TDD) utilizado para escribir software.

Para la implementación del visor, proponemos utilizar a las heurísticas como principios básicos para guiar la construcción de la interfaz gráfica de usuario. En base a las heurísticas presentadas por Jakob Nielsen en [Nielsen94] [Nielsen94c], hemos creado una lista extendida y depurada y la hemos utilizado como guía para el diseño de nuestra aplicación. Dicha lista se enumera a continuación:

1. **Usabilidad del estatus del sistema**: El sistema siempre debe mantener informado al usuario respecto de dónde está ubicado. Cada sección debe estar categorizada (titulada) e indicar a qué operación o sector de información pertenece.

2. **Coincidencias en la comunicación**: El sistema debe comunicarse en el lenguaje del usuario, con palabras, frases y conceptos familiares a éste. La información debe aparecer en un orden lógico y natural.

3. **Libertad y control del usuario**: En los casos en que la elección de una función sea errónea, el sistema debe proporcionar una salida rápida y sin consecuencias no deseadas. Esta salida debe ir más allá de la que provee el navegador. El usuario debe tener el control sobre el sistema.

4. **Prevención de errores**: Es mejor tener un diseño que prevenga los problemas que buenos mensajes de error. De todas formas, cuando se produce un error, es necesario tener una gestión apropiada de este, para no degradar la experiencia del usuario al utilizar nuestra aplicación.

5. **Reconocimiento en lugar de rememoración**: Las instrucciones acerca del uso del sistema deben estar visibles o fácilmente accesibles cuando sea apropiado.

6. **Flexibilidad y eficiencia en el uso**: Permitir que el usuario pueda personalizar las acciones más frecuentes para acelerar sus consultas.

7. **Diseño estético o minimalista**: El sistema no debe tener información irrelevante o innecesaria.

8. **Ayuda**: La información debe ser fácil de encontrar y basarse en los objetivos del usuario. Este no debe sentir que la asistencia está muy lejos, o inaccesible para la ocasión.

9. **Ayudar al usuario a reconocer, diagnosticar y recuperar los errores del sistema**: Los avisos de error del sistema deben estar expresados en un lenguaje simple y familiar para el usuario, indicando el problema preciso y sugerir una posible solución.

10. **Destino**: El usuario siempre debe estar informado hacia donde se dirige al activar los links, sean gráficos o de texto.

11. **Información sobre fragmentaciones**: La información sobre qué tamaño tiene un documento o si tiene más secciones, debe estar visible y ser clara.

12. **Proveer niveles de detalles avanzados**: La información debe estar organizada jerárquicamente. El sistema debería dar seguridad para explorarlo sin miedo a las posibles equivocaciones. También debería permitir que el usuario pueda detenerse cuando logre obtener la información requerida.

13. **No mentir al usuario**: No debe haber referencias sobre información inexistente.

La implementación de esta lista da como resultado un marco de trabajo[214] similar (en cuanto a prestaciones) pero más sencillo al propuesto por Bruce Tognazini[215] en [Tognazini00], gracias al poder de la emergencia (aplicamos reglas locales sencillas que producen un diseño global complejo, ver **capítulo 9**).

14.3.2 Escritura para la Web

Los usuarios no leen en la Web, sino que escanean sus páginas. Los usuarios en la Web no leen, o por lo menos no lo hacen en la misma forma secuencial que cuando tienen entre manos un texto impreso. Los usuarios tienen necesidades y objetivos, metas que alcanzar y saben que la forma de conseguir dichas metas no suele ser dedicando largos ratos a cada página Web que visitan. No quieren perder tiempo leyendo "textos vacíos" en la Web. [Welicki04b]

La gente raramente lee las páginas Web palabra por palabra [Nielsen97b]. En cambio, escanean la página, tomando palabras individuales u oraciones. Ante una página, el usuario ojea a saltos la información contenida. Si bien no se puede predecir cuál será el camino que seguirá, si se puede saber a qué zonas presta mayor atención en su búsqueda. [Welicki04b]

Una conclusión interesante es que los usuarios generalmente no leen las páginas: *las usan* [Nielsen97b]. Antes de saber si un texto es de su interés, lo hojean, buscan algunas palabras claves y luego, quizás lo impriman para su posterior lectura. Esto es debido a la conjunción de la infinidad de contenidos existentes en el Web y a la incomodidad de la lectura en pantalla. Debido a esto, los usuarios intentaran obtener la mayor información realizando el menor esfuerzo posible.

Para escribir un bloque de texto escaneable, recomendamos utilizar las siguientes técnicas:

- Palabras claves resaltadas (negritas, vínculos de hipertexto, variaciones de fuentes, etc.).

[214] Este marco de trabajo ha sido probado y utilizado por el autor de esta tesis para el desarrollo de aplicaciones empresariales. El resultado ha sido positivo en todos los casos.
[215] Reconocido experto en usabilidad miembro del *Nielsen Norman Group*, conformado también por Jakob Nielsen y Donald Norman.

- Subtítulos significativos.
- Listas con viñetas.
- Expresar una idea por párrafo (los usuarios saltaran a ideas adicionales si no se sienten atraídos por las palabras iniciales del párrafo).
- Utilizar el estilo de la pirámide invertida.
- Utilizar la mitad de las palabras que en el texto impreso.

Por otro lado, los usuarios detestan el texto "marketinero"[216], vacío de contenidos pero lleno de palabras [MN97] [MN98] [Nielsen97b]. Este tipo de lenguaje, tiene una alta carga de fricción cognitiva [Cooper99], haciendo borrosa a la idea principal del texto. Según Jakob Nielsen, el escribir correctamente aumenta la usabilidad en un espectacular 124% [MN97].

En consecuencia, para escribir textos efectivos para ser leídos en la Web proponemos seguir las siguientes reglas [Welicki04b] [MN97] [MN98] [Nielsen97b]

- Ser sucinto
- Escribir para usuarios que escanean
- Utilizar el estilo de la pirámide invertida
- Ser consistente
- Escribir con lenguaje claro y objetivo
- Evitar ambigüedades
- Credibilidad

En [Welicki03e] se presenta una discusión en profundidad sobre este tema, así como una guía de estilo completa para escribir textos para la Web. En la herramienta de visualización deben seguirse estas guías, a efectos de hacer fácilmente legibles los contenidos del catálogo para los usuarios y maximizar las capacidades de comunicación.

14.3.3 Áreas de Navegación

Hemos seguido la premisa de "dividir la página en zonas claramente definidas" [Krug99] [Nielsen99]. La estructura general de navegación utilizada es la tradicional, como se muestra en la siguiente figura:

[216] "Marketese" (en el texto original)

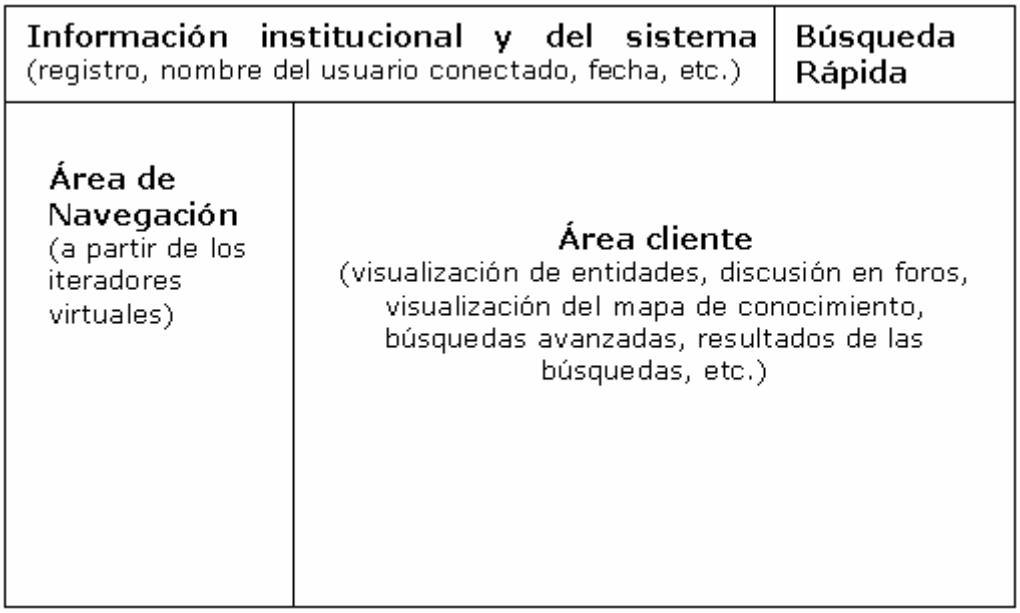


Figura 14.2 – Estructura general de la interfaz de usuario del visor del catálogo.

El gráfico presentado arriba es el esquema general de interacción de la aplicación. Este esquema se mantiene inmutable en toda la aplicación, facilitando al usuario la interacción con el sistema. Una vez aprendidas las reglas generales de uso se puede utilizar fácilmente cualquier área de la aplicación (recordar la heurística 5, "reconocimiento en lugar de rememoración")

14.4 Vistas Dinámicas

Un problema común en todos los visores de catálogos públicos de patrones es que la interfaz de usuario fuerza al usuario a concentrarse en una única vista del patrón, que ha sido definida en función de los intereses u objetivos del publicador de la información. Para evitar esta situación, proponemos realizar una fuerte separación entre las entidades y su visualización. Para lograr este objetivo hemos creado un motor de vistas dinámicas, que permiten asociar vistas dinámicamente a las entidades en tiempo de ejecución.

La premisa fundamental es separar el modelo de la vista (tomando como base una arquitectura MODEL-VIEW-CONTROLLER (MVC) [POSA96]). En este contexto, tomando al patrón (o a cualquier entidad EML) como el modelo, se le pueden asociar múltiples vistas. Más aún, las vistas se pueden vincular dinámicamente al modelo, permitiendo cambiar el estilo de visualización en tiempo de ejecución a partir de criterios arbitrarios (basado en una arquitectura de plug-ins e inyección de dependencias). Las vistas se registran en un repositorio de vistas y se vinculan a las entidades en base a reglas determinadas por el usuario o el administrador del sistema. En todo momento pueden crearse vistas nuevas y vincularlas a entidades. Es posible también modificar o eliminar las vistas existentes.

A modo de ejemplo, enumeramos algunas posibles vistas que pueden ser vinculadas a los patrones:

- **Vista Completa**: muestra la información completa sobre el patrón.
- **Vista Resumida**: resumen de la información del patrón
- **Vista de Tarjetas CRC**: tarjetas CRC[217] para los participantes del patrón
- **Vista de Código Fuente**: código fuente del patrón. Permite seleccionar el lenguaje destino, la implementación a utilizar y modificar opciones de implementación para ajustar los resultados.
- **Vista EML**: código EML del patrón
- **Vista Personalizada**: el usuario puede seleccionar en tiempo de ejecución qué elementos desea ver de todos los que están disponibles en la meta-especificación de una entidad. A partir de esta selección el sistema genera la salida.

Cada vista se especializa en mostrar un aspecto concreto de una entidad. En la figura 14.3 se muestra como pueden aplicarse diferentes vistas a una misma entidad y obtener resultados muy diferentes.

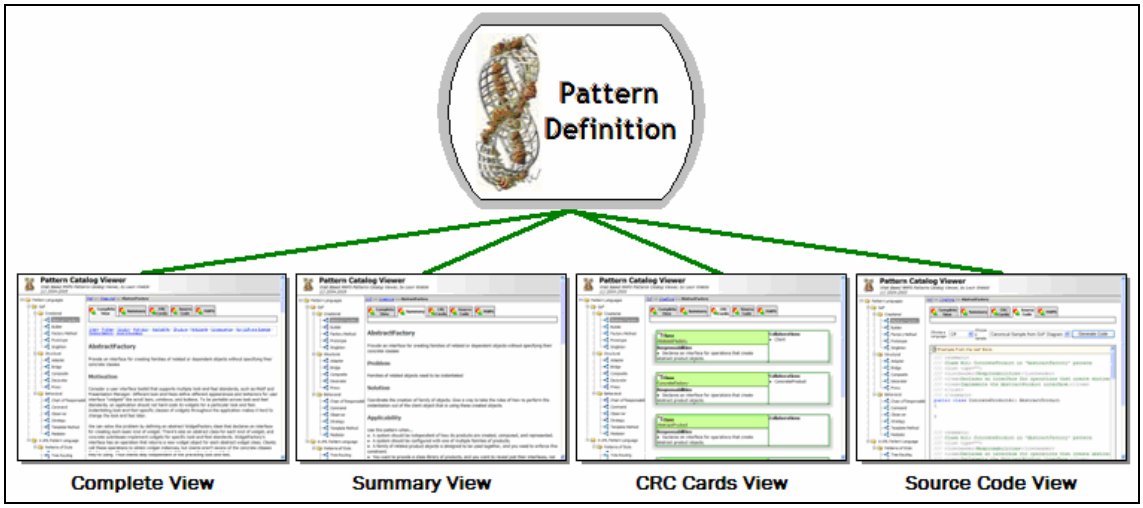


Figure 14.3 – Múltiples vistas sobre una entidad. En este caso, se muestran varias vistas asociadas a un patrón. Cada vista permite al usuario concentrarse en un aspecto específico de la entidad que esta visualizando.

14.5 Escritura N-Dimensional

En el trabajo de investigación "*XText: Un Modelo para Creación y Visualización de Textos en Múltiples Dispositivos*" [Welicki04b], se presenta un modelo n-dimensional para producción de textos. Este modelo permite "navegar" un texto en los ejes horizontales y verticales. La intención original del modelo era establecer un marco para la producción y visualización de textos en múltiples dispositivos. Este modelo es totalmente aplicable al modelo de interacción de textos del visor catálogo.

[217] Class-Responsibility-Collaboration

Para la representación del texto, proponemos un modelo multi-nivel, donde a medida que se aumenta de nivel, aumenta la cantidad de información. En consecuencia, *la cantidad de información es función del nivel* en que se encuentra.

- Cada nivel puede ser visto como una dimensión en un espacio tridimensional. Podríamos decir que el nivel es la altura del texto, tomando como punto de vista una visión de los ejes X e Y del plano del espacio asociado.

- A mayor nivel, mayor cantidad de información. Por lo tanto, los niveles inferiores son más aptos para ser visualizados por dispositivos con capacidades limitadas o tamaños de pantalla pequeña (o por usuarios que desean menos información respecto un asunto particular).

En la figura a continuación se muestra un ejemplo de diferentes niveles de representación del texto. Nótese como la cantidad de información crece en función del nivel en que se encuentra.

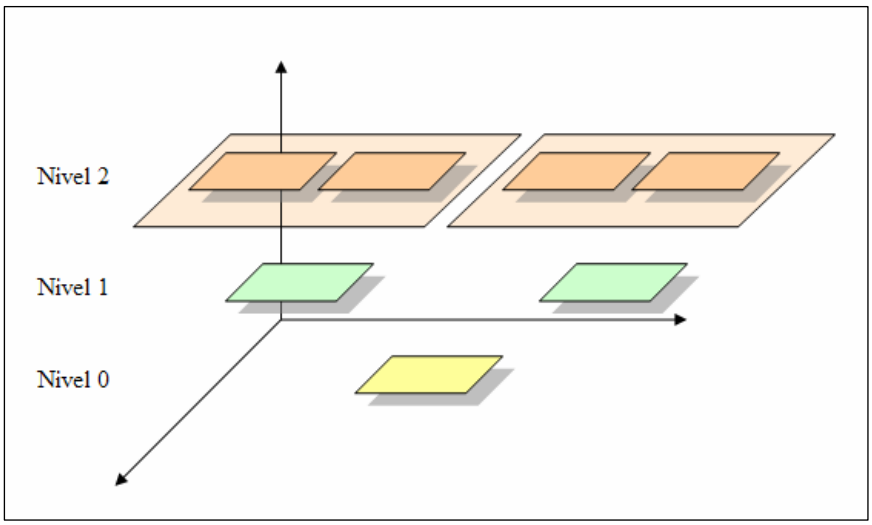


Figura 14.4 – Esquema de escritura en múltiples dimensiones. Cada nivel contiene más información. Los niveles inferiores contienen información sumarizada.

En el caso del catálogo, se pueden crear vistas especializadas (§ 14.4) para cada nivel vertical. Estas vistas estarán especializadas en la generación de código de visualización para dispositivos específicos, optimizando así la visualización en cada caso. De este modo, podemos cumplir con la definición de gestión de conocimiento postulada por Joyanes [Joyanes03], que dice "*compartir información en cualquier momento, en cualquier lugar y con cualquier dispositivo*".

14.6 Soporte para Comunidad y Colaboración

En el **capítulo 13** presentamos las capacidades de colaboración y comunidad del catálogo. El visor del catálogo explota esas capacidades, proveyendo un frontal para interactuar con ellas y añadiendo otras nuevas.

14.6.1 Comunidad

La interfaz Web proporciona una fachada sobre las herramientas de colaboración del catálogo. De esta forma, los usuarios pueden participar en los foros asociados a cada entidad o actualizar su información en el mapa de conocimiento.

- **Foros**: espacios de discusión donde los usuarios pueden discutir sobre temas concretos.

- **Mapa de Conocimiento**: directorio de todos los usuarios con sus conocimientos y habilidades

- **Debates**: espacios de discusión interactivos. Estos espacios montan en forma programada. En un debate participan varios usuarios intercambiando información sobre un asunto en tiempo real. Para implementar la funcionalidad de conversación (Chat) puede utilizarse el protocolo IRC [IRC] o cualquier motor de mensajería instantánea.

- **Estadísticas sobre uso de entidades**: mediante el uso de las estadísticas de acceso, los usuarios pueden saber que entidad es más popular. Por ejemplo, ante dos entidades que resuelven problemas similares, un usuario puede elegir la más popular (que es la que más accesos tiene).

14.6.2 Modo Shepherding

El proceso de *shepherding* [Hillside] es el que se utiliza en todas las variantes de las conferencias PLoP para mejorar los trabajos enviados. El visor del catálogo puede dar soporte a este proceso mediante la orquestación de funcionalidades básicas del catálogo. El proceso utilizando el catálogo podría ser el siguiente:

1. El autor publica la versión inicial de su patrón en el catálogo
    a. El autor indica quien o quienes serán los shepherders de su trabajo. Estos serán los únicos que podrán visualizar su patrón.
    b. Se indica quien será el miembro del comité de programa que revisara las interacciones durante el proceso.

2. El shepherder revisa el patrón original. Una vez que lo ha revisado, puede realizar las siguientes acciones:
    a. Modificar el patrón original
    b. Añadir secciones con anotaciones para el autor

3. Una vez que el shepherder ha terminado de revisar el patrón, sube una versión de este al catálogo (utilizando el sistema de versionado).

4. El autor se entera de esta nueva versión mediante el sistema de sindicación de contenidos asociado al catálogo o mediante una suscripción.

5. El autor revisa los cambios y sube una nueva versión al catálogo.

6. Se vuelve al paso 2 hasta que se completan las iteraciones necesarias o se acaba el plazo del proceso de shepherding.

14.6.3 Folksonomies

Folksonomía[218], es un neologismo que da nombre a la categorización colaborativa por medio de etiquetas simples en un espacio de nombres llano, sin jerarquías ni relaciones de parentesco predeterminadas. Se trata de una práctica que se produce en entornos de software social cuyos mejores exponentes son los sitios compartidos como del.icio.us (enlaces favoritos) [Delicious] y Flickr (fotos) [Flickr].

Según [Estatella05], las folksonomías son un nuevo paradigma de clasificación de la información que permite a los internautas crear libremente etiquetas para categorizar todo tipo de contenidos, desde enlaces de noticias a fotografías, pasando por canciones, artículos especializados, etcétera. Este uso colectivo de etiquetas genera un sistema de categorización no jerárquico [Estatella05].

Derivado de *taxonomía*, el término *folksonomy* ha sido atribuído a Thomas Vander Wal. Taxonomía procede del griego "taxis" y "nomos": Taxis significa clasificación y nomos (o nomia), ordenar, gestionar; por su parte, "folc" proviene del alemán "pueblo" (volks). En consecuencia, de acuerdo con su formación etimológica, folcsonomía (folc+taxo+nomía) significa literalmente "clasificación gestionada por el pueblo (o democrática)".

EML provee un sub-lenguaje especializado en la definición de etiquetas para entidades (EML-AL, § 12.2.4.3). El visor del catálogo da soporte a la creación de folksonomías a partir de una infraestructura que permite el etiquetado colaborativo de las entidades basada en la edición de la sección de EML-AL de una meta-especificación EML (**capítulo 12**).

14.7 Navegación por las Entidades del Catálogo

El visor del catálogo ofrece al usuario tres modelos diferentes de navegación. Cada uno tiene a su vez opciones de parametrización, permitiendo al usuario ajustar la forma en que navega al utilizar un paradigma específico. De esta forma el esquema de navegación no es fijo, permitiendo al usuario decidir en qué forma quiere recorrer los contenidos del catálogo.

[218] Folksonomy (en inglés)

14.7.1 Navegación Jerárquica (Categorizada)

En este caso la navegación se realiza mediante una estructura de jerárquica (árbol) que se muestra a la izquierda de la pantalla, donde se agrupan los lenguajes de patrones con sus respectivas categorías y patrones, como se muestra en la figura a continuación. En esta categoría se puede navegar por los contenidos del catálogo siguiendo una estructura lógica de categorización. Esta categorización se arma dinámicamente a partir de las relaciones almacenadas en el catálogo.

En la siguiente figura se muestra un ejemplo de navegación jerárquica. En este caso, la categorización se construye a partir de las relaciones existentes entre "*Lenguajes de Patrones*", "*Categorías*" y "*Patrones*".

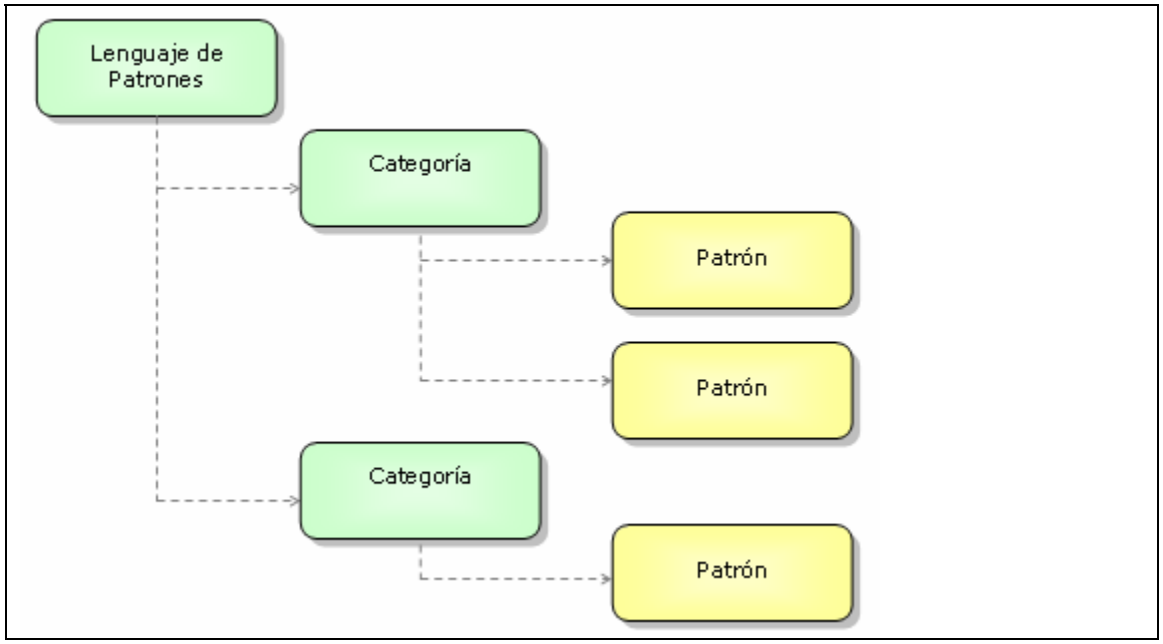


Figura 14.5 – Ejemplo de esquema de navegación por categorización (en este caso, la categorización esta dada por la relación existente entre lenguajes de patrones, categorías y patrones).

14.7.1.1 ¿Cómo Funciona la Categorización?

Las jerarquías de categorización se arman dinámicamente a partir de los iteradores virtuales (§ 13.4.4) expuestos por el catálogo. El usuario puede seleccionar el iterador virtual que desea utilizar (y en consecuencia, la jerarquía) para poder navegar por los contenidos del catálogo.

14.7.2 Navegación por Descubrimiento

Este modelo de navegación permite al usuario ir navegando a través de las relaciones del catálogo y de esta forma ir "***descubriendo***" las entidades que se encuentran alojadas en éste. Utilizando este modelo de navegación el usuario puede explorar los contenidos del catálogo sin tener un objetivo específico y encontrar entidades que no estaba buscando concretamente.

Formalmente, podemos decir que este modelo equivale a recorrer un grafo conexo, donde las aristas son las relaciones que comunican a los vértices que son las entidades, permitiendo al usuario navegar libremente por los contenidos del catálogo a partir de las relaciones existentes entre éstos.

Cuando un usuario visualiza una entidad, tiene la opción de visualizar también enlaces a todas las entidades relacionadas. De esta forma, se puede navegar en forma sencilla de una entidad a otra. Adicionalmente, el sistema "presenta" al usuario entidades relacionadas con la que está siendo visualizada, permitiendo al usuario aumentar su "conocimiento" sobre la entidad que está consultando.

14.7.3 Navegación Mediante Árboles Hiperbólicos

En este caso, la navegación se realiza mediante una estructura basada en árboles hiperbólicos. El árbol se arma dinámicamente a partir de las relaciones[219] entre las entidades almacenadas en el catálogo.

El usuario puede seleccionar una entidad que se convierte en el centro del árbol hiperbólico. Al seleccionar esta entidad, el sistema ofrece la posibilidad de visualizar la información completa de ésta utilizando alguna de las vistas que tiene asignadas.

14.7.3.1 Sobre los Árboles Hiperbólicos

Los árboles hiperbólicos son un método para mostrar una jerarquía de cualquier tamaño en un área finita (un círculo). El nodo en que se establece el foco está centrado y a medida que nos movemos del nodo central hacia un extremo del círculo cada nodo se hace exponencialmente más pequeño. Esto permite una mayor representación cerca del área donde se hace el foco mientras se sigue visualizando la estructura total del árbol [UFHT][220].

El árbol hiperbólico es ideal para organizar y mostrar las grandes cantidades de información interconectada y datos en aplicaciones de gestión del conocimiento[221]. [Tabor]

Los árboles hiperbólicos permiten visualizar en forma eficiente grandes cantidades de información relacionada. En la figura 14.5 se muestra un ejemplo extremo: un grafo con la estructura de Internet en Junio de 1999. La utilización de árboles hiperbólicos permitiría navegar en forma eficiente por esta estructura.

---

[219] Estas relaciones se especifican mediante

[220] Texto Original: "*a method of displaying a hierarchy (such as a corporate org-chart or a directory structure) of any size within a finite area (a circle). The node that is focused on is centered, and as you move away from the central node and toward the edge of the circle, each node is made exponentially smaller. This allows larger representation near the focused area while still displaying the overall structure of the tree.*"

[221] Esto también es aplicable a otros dominios como inteligencia de negocios, CRM, e-commerce, etc.

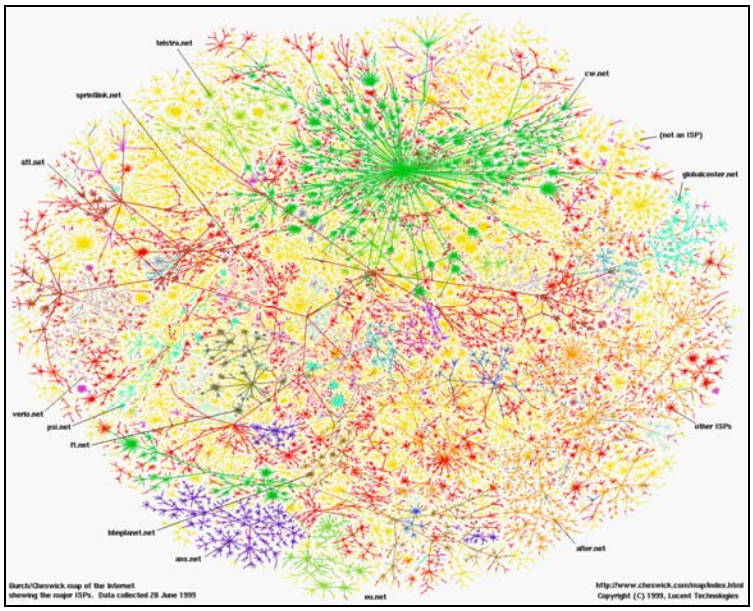


Figura 14.6 - Mapa de Internet, tomado de [BC99]

14.8 Buscador

La interfaz del buscador permite interactuar directamente con el motor de búsqueda del catálogo. El usuario puede seleccionar varias modalidades para realizar la búsqueda:

- **Búsqueda Rápida**: busca por cualquier tipo de coincidencia de la sarta introducida por el usuario. En este caso, el usuario introduce un texto y éste es buscado utilizando simple coincidencia de textos (incluye búsqueda en las etiquetas y relaciones)

- **Búsqueda por etiquetas**[222]: la búsqueda se realiza utilizando las etiquetas que decoran las entidades (§ 12.2.4.3 y § 13.4.3)

- **Búsqueda avanzada**: el usuario puede introducir una combinación avanzada de parámetros para realizar una búsqueda contextual (§ 13.4.3, § 12.2.4.2). La interfaz de usuario simplifica la introducción de estos parámetros, permitiendo al usuario buscar a partir de textos, etiquetas o relaciones.

- **Inferencias**: el sistema puede realizar inferencias a partir de las relaciones y de la ontología existente (§ 13.4.3, § 12.2.2.1, § 12.2.4.2). La interfaz de usuario Web ofrece al usuario un método sencillo para introducir la información de búsqueda.

14.9 Conclusión: Revisión de los Requisitos

[222] Tags (en inglés)

El visor del catálogo que hemos propuesto en este capítulo cumple con todos los requisitos que hemos establecido al inicio de este capítulo. En la tabla 14.1 a continuación, verificaremos la forma en que se realiza cada uno de los requisitos planteados al inicio de este capítulo. Para cada requisito se incluye una frase breve que indica como se realiza en el sistema, con una referencia a la sección donde es presentado en este capítulo

Requisito	**Descripción**	**Realización**
RV-1	Multiplataforma	La aplicación Web puede visualizarse desde cualquier plataforma a través de un navegador de Internet.
RV-2	Accesible a través de la Web	El visor es una aplicación Web (por lo tanto inherentemente accesible desde la Web).
RV-3	Usabilidad	El visor del catálogo ha sido diseñado siguiendo pautas de usabilidad establecidas por los principales autores (Nielsen, Krug y Cooper) y de amplia aceptación (§ 14.3)
RV-4	Múltiples vistas de la información	El visor permite aplicar múltiples vistas sobre las entidades. De esta forma, el usuario puede tener varias vistas sobre la misma entidad. Las vistas pueden vincularse a las entidades dinámicamente (§ 14.4)
RV-5	Navegabilidad	El visor del catálogo ofrece diversas opciones para navegar a través de los contenidos almacenados en el catálogo. En cada caso se explota un modelo cognitivo distinto (§ 14.7).
RV-6	Facilidades de búsqueda	El visor del catálogo incluye una interfaz de búsqueda (§ 14.8). Estas búsquedas pueden utilizar los distintos niveles del gradiente semántico.
RV-7	Comunidad	El visor del catálogo incluye funcionalidades orientadas a dar soporte a la generación de comunidad como los foros, los debates y la edición y anotación colaborativa (§ 14.6)
RV-8	Personalización de resultados	La vista personalizada (§ 14.4) permite seleccionar las partes más relevantes de una entidad en un contexto específico. La vista de código fuente (§ 14.4) permite al usuario seleccionar el lenguaje y la implementación a utilizar para generar el código y ajustar el comportamiento de la entidad para modificar el código generado.

Tabla 14.1 – Revisión del cumplimiento de los requisitos del visor del catálogo.

Parte IV
Desarrollo de un Prototipo

"No basta saber, se debe también aplicar.
No es suficiente querer, se debe también hacer."
Johann Wolfgang Goethe

Capítulo 15
Descripción del Prototipo

"Tener grandes ideas es excelente,
transformarlas en realidad una virtud"
Anónimo

A efectos de validar el modelo propuesto en forma empírica y pragmática, se ha desarrollado un prototipo que permite describir (meta-especificar), gestionar y compartir un catálogo de patrones especificados en EML. El prototipo es una implementación de referencia del modelo completo presentado en los capítulos anteriores de esta tesis, a efectos de demostrar su factibilidad y su potencial.

El prototipo ha sido enteramente desarrollado utilizando el marco de trabajo .NET [NET]. Los componentes activos del catálogo y objetos de negocio son componentes desarrollados con C#. La interfaz Web y la fachada de servicios Web [W3C] han sido construidas sobre ASP.NET [MSAN]. La interfaz gráfica de registro ha sido escrita usando WinForms [MSWF]. La infraestructura de soporte al catálogo (que es donde se almacenan los componentes pasivos) se compone de un sistema de ficheros y de una base de datos relacional Sql Server [MSSQL].

En este capítulo presentaremos una descripción general del prototipo, que servirá como introducción para el capítulo siguiente, donde se estudian en detalle aspectos específicos del prototipo desarrollado. Al terminar de leer este capítulo, el lector tendrá una idea general de qué es lo que incluye el prototipo desarrollado durante la investigación.

❖ ❖ ❖

15.1 Objetivos del Prototipo

El objetivo principal del prototipo es demostrar la factibilidad técnica y la utilidad práctica del modelo propuesto en esta tesis doctoral. Para ello, se ha realizado una implementación de referencia de la arquitectura propuesta en los **capítulos 11**, **12**, **13** y **14**.

Los objetivos generales de implementación son los siguientes:

- **Lenguaje de Descripción**: Crear la especificación completa del lenguaje de meta-especificación de patrones y conceptos
- **Catálogo (Pasivo)**: Confeccionar un catálogo de patrones y entidades descritas con el lenguaje de meta-especificación mencionado arriba.
- **Catálogo (Activo)**: Crear la infraestructura necesaria para dar soporte a la catalogación, clasificación y compartición de los contenidos del catálogo
- **Visor del Catálogo**: Crear un cliente de visualización que permita compartir los contenidos del catálogo en forma eficiente a usuarios finales.

15.1.1 Funcionalidades Incluidas en el Prototipo

En esta sección se enumeran y comentan brevemente las principales funcionalidad incluidas en el prototipo desarrollado durante esta tesis doctoral.

15.1.1.1 Lenguaje de Descripción

- **EML 1.0**: especificación completa de EML y todos sus sub-DSLs (EML-RDL, EML-PDL, EML-SDL y EML-BDL, presentados en el **capítulo 12**). Soporte a nivel de lenguaje para el gradiente semántico (§ 12.2.5).

15.1.1.2 Catálogo (Pasivo)

- **Catálogo de Entidades**: catálogo de patrones y conceptos de soporte (lenguajes de patrones, categorías, refactorizaciones, autores, etc.) descritos utilizando EML (§ 13.3.1).
- **Definición de los Iteradores Virtuales**: especificación de los iteradores virtuales utilizando el DSL de especificación de iteradores virtuales (§ 13.3.3)
- **Relaciones**: relaciones entre las entidades del catálogo (especificadas con EML-RDL, § 12.2.4.2, § 13.3.4).

- **Gradiente Semántico**: posibilidad de expresar los distintos niveles del gradiente semántico (§ 12.2.5) utilizando EML.

- **Representación de Fuerzas**: representación de las fuerzas [Alexander79] [Hillside] que resuelve un patrón. Esta representación se realiza mediante entidades EML y relaciones (EML-RDL, § 12.2.4.2). Cada fuerza es una instancia de una entidad de tipo "*Fuerza*" que se puede relacionarse con otra de tipo "*Patrón*" mediante una relación "*es resuelta por*".

- **Registro de Entidades:** registro de las entidades disponibles en el sistema (§ 13.3.1). Para que una entidad esté disponible en el sistema debe estar incluida en este registro. Una vez que la entidad ha sido registrada dispone de una "dirección" en el catálogo y puede ser gestionada a través de éste.

15.1.1.3 Catálogo (Activo)

- **Interfaz de Registro de Entidades**: interfaz para manipular el registro de entidades (registrar y desregistrar del catálogo, § 13.4.1). Incluye una interfaz gráfica de usuario para realizar estas tareas.

- **Iteradores Virtuales**: implementación de los iteradores virtuales (§ 13.4.4), que permite crear jerarquías de navegación dinámicamente a partir de las entidades y relaciones existentes en el catálogo y de información almacenada en metadatos de configuración. Los iteradores virtuales están implementados sobre una arquitectura flexible y extensible basada en los patrones ITERATOR [GoF95], VIRTUAL PROXY [POSA96], BUILDER [GoF95] y DEPENDENCY INJECTION [Fowler04].

- **Analizador de Entidades**: analizador de EML (§ 13.4.2.2). El analizador ha sido creado sobre una arquitectura flexible, lo cual permite modificar los pasos de análisis en función de condiciones particulares (basado en una combinación de los patrones PIPES AND FILTERS [POSA96], REFLECTION [POSA96], DEPENDENCY INJECTION [Fowler04], SIMPLE FACTORY [Welicki05g], POLYMORPHISM [Larman99] y STRATEGY [GoF95]). El analizador es parte de FREP (§ 13.4.2).

- **Representación de Entidades en Tiempo de Ejecución**: representación de las entidades en tiempo de ejecución utilizando modelos de objetos adaptativos (AOM). La representación de entidades es parte de FREP (§ 13.4.2)

- **Generación de Código**: motor de generación de a partir de las meta-especificaciones de las entidades (§ 13.4.10). El motor de generación código esta implementado sobre una arquitectura flexible y extensible que permite inyectar nuevos generadores en el sistema a partir de metadatos de configuración. Esta microarquitectura esta basada en una combinación de los

patrones REFLECTION [POSA96], DEPENDENCY INJECTION [Fowler04], SIMPLE FACTORY [Welicki05g], POLYMORPHISM [Larman99], STRATEGY [GoF95], TEMPLATE METHOD [GoF95] y BRIDGE [GoF95].

- **Buscador**: buscador de entidades (§ 13.4.3). El buscador permite buscar en los distintos niveles del gradiente semántico (texto libre, etiquetas o relaciones)

- **Colaboración**: infraestructura de colaboración a través de foros y etiquetado colaborativo (§ 13.4.6).

- **Versionado**: versionado de entidades (§ 13.4.9). El sistema permite tener varias versiones de la misma entidad registradas simultáneamente. Los usuarios pueden acceder a cualquier versión de la entidad.

- **Direccionamiento**: cada entidad tiene un identificador que permite identificarla en forma unívoca e inequívoca (§ 13.4.1.3). Este identificador es utilizado como la "dirección" de la entidad en el catálogo.

- **Suscripción y Sindicación**: los usuarios registrados en el sistema pueden suscribirse a cambios (§ 13.4.7) en las entidades (mediante un sistema similar al descrito por el patrón OBSERVER [GoF95]). Adicionalmente, el catálogo expone sus cambios a través de feeds RSS.

- **Gestión de Usuarios**: gestión de usuarios del sistema (§ 13.4.5). Los usuarios deben estar registrados en el sistema para que puedan realizar cualquier tipo de modificación (por ejemplo, etiquetado colaborativo, suscripción, participación en foros, etc.)

- **Auditoria**: el catálogo mantiene información de auditoría sobre su actividad (§ 13.4.5). Esta información puede ser utilizada en contextos diversos (desde verificación de aspectos de seguridad hasta identificación de las entidades más populares).

- **Interfaz SOA**: interfaz que permite exponer contenidos del catálogo a través de servicios Web (§ 13.4.8). Esta interfaz puede ser utilizada en el contexto de aplicaciones basadas en arquitecturas orientadas a servicios desde fuentes heterogéneas como se propone en [Welicki05c].

15.1.1.4 Visor del Catálogo

- **Múltiples Modelos de Navegación**: el visor permite al usuario navegar por los contenidos del catálogo utilizando múltiples modelos de navegación, en función de sus preferencias (§ 14.7). El usuario puede navegar por el catálogo en forma jerárquica utilizando los iteradores virtuales (puede seleccionar

cualquier de los que están disponibles en el sistema) o navegar a partir de las relaciones.

- **Motor de Vistas Dinámicas**: el visor del catálogo permite adjuntar múltiples vistas a una misma entidad (§ 14.4). Esto permite al usuario seleccionar la forma en que desea ver la entidad, eligiendo en qué aspecto de ésta quiere poner el foco. Las vistas se vinculan a las entidades en tiempo de ejecución. Por lo tanto, pueden añadirse nuevas vistas al sistema sin necesidad de reiniciarlo. Para cada tipo de entidad o entidad concreta pueden asignarse un conjunto de vistas. Por ejemplo en el caso de los patrones se pueden vincular las vistas de plantilla completa, plantilla resumida, tarjetas CRC y código fuente. El motor de vistas dinámicas está implementado sobre una arquitectura flexible y extensible basada en una combinación de los patrones REFLECTION [POSA96], DEPENDENCY INJECTION [Fowler04], SIMPLE FACTORY [Welicki05g], POLYMORPHISM [Larman99], STRATEGY [GoF95] y BRIDGE [GoF95].

- **Buscador**: el visor incluye funcionalidades de búsqueda (§ 14.8). A través de la interfaz de búsqueda el usuario puede buscar entidades utilizando los distintos niveles del gradiente semántico (texto libre, etiquetas y relaciones).

- **Discusión y Comunidad**: cada entidad tiene asociado un foro de discusión multi-hilo donde los usuarios pueden discutir sobre diversos aspectos de ésta (§ 14.6). Además los usuarios pueden votar las entidades, creando así rankings ad-hoc de los contenidos del catálogo.

- **Vista General**: existe una vista general que puede ser adjuntada a cualquier entidad para mostrar sus contenidos. Por lo tanto, cualquier entidad registrada en el sistema puede ser presentada en este visor.

- **Vista de Plantilla Basada en Metadatos**: existe una vista que puede ser aplicada a las entidades y que se configura mediante metadatos. El usuario puede crear un fichero de layout donde configura qué elementos de la entidad serán mostrados en la vista y aspectos generales (como clases CSS, columnas a utilizar, etc.). El implementador de la vista interpreta estos metadatos y genera la salida adecuada tomando como entrada a una entidad concreta.

- **Texto N-Dimensional**: esta característica se implementa a partir de una combinación de vistas (utilizando el motor de vistas dinámicas). En el caso de los patrones hay dos vistas que están a distintas "alturas" (§ 14.5) en el modelo de texto n-dimensional: la vista completa y la vista resumida. Adicionalmente el sistema permite crear nuevas vistas que estén a mayor o menor "altura" y vincularlas a las entidades.

15.2 Arquitectura

El prototipo incluye implementaciones completas y funcionales de todos los elementos que conforman la solución propuesta en esta tesis (**capítulos 11**, **12**, **13** y **14**). Ha sido implementado a través de un conjunto de especificaciones (EML 1.0, iteradores virtuales, etc.) y tecnologías (FREP, registro de entidades, etc.).

La arquitectura de software está fundada en varios patrones y prácticas ampliamente aceptadas en la industria y la academia. En la figura a continuación se muestra la visión de alto nivel de la arquitectura del sistema completo.

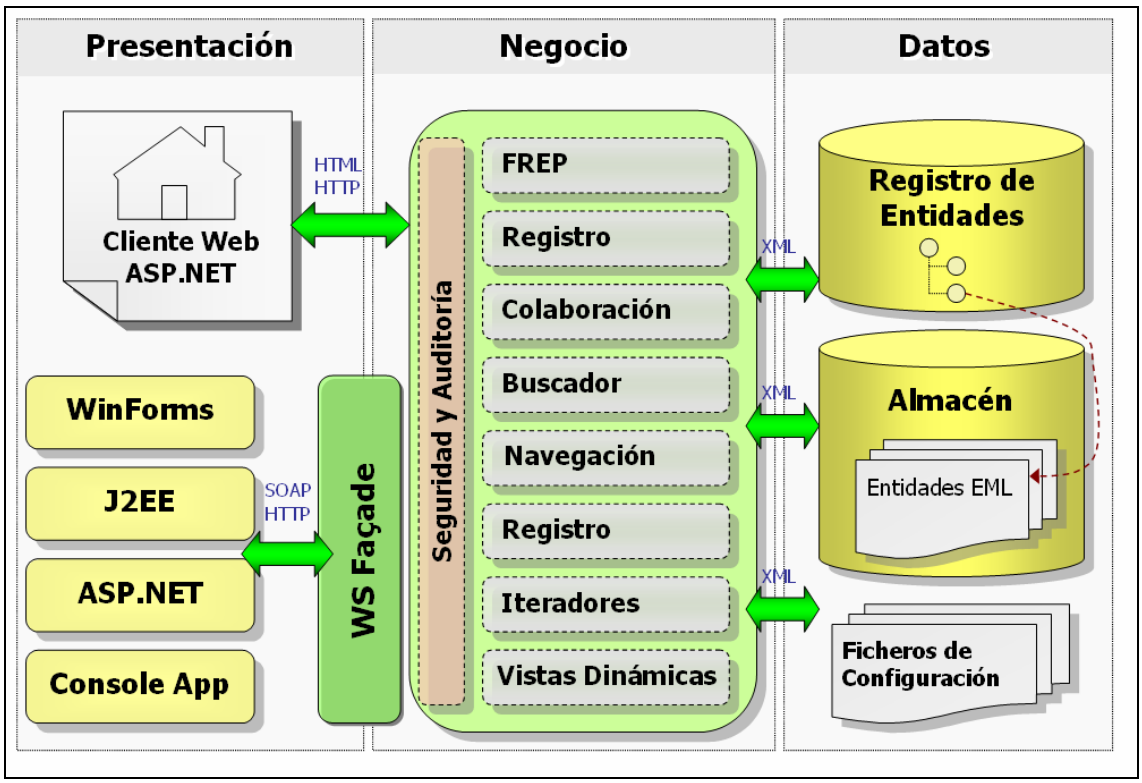


Figura 15.1 – Arquitectura de alto nivel del prototipo

En la figura 15.1 puede observarse la división en capas (siguiendo el patrón LAYERS [POSA96]), donde cada capa tiene responsabilidades bien definidas. Esta división sigue los lineamientos de las arquitecturas de tres capas tradicionales, donde los componentes pasivos del catálogo hacen las veces de capa de datos, los componentes activos hacen de capa de negocios y el visor del catálogo hace las veces de capa de presentación. Dado que el catálogo expone sus servicios mediante una fachada (patrón FAÇADE [GoF95]) de servicios Web, sus contenidos son accesibles desde cualquier entorno.

Las capas tienen acoplamiento mínimo y se comunican mediante mensajes [HW03] basados en XML[223]. En la figura se incluyen las principales funcionalidades del

[223] Esto también es válido cuando se transfieren las entidades EML dado que éste está basado en XML

componente activo en la capa de negocio, pero esta incluye a todas las funcionalidades que se mencionan previamente en el **capítulo 13** y posteriormente en el **capítulo 16**.

Es importante destacar que el componente pasivo (capa de datos) solo es accesible mediante el componente activo (la capa de negocios). De esta forma podemos controlar y auditar todos los accesos que se hacen a éste. Este control lo realiza el módulo de seguridad y auditoría. Este componente es transversal a todos los que conforman el componente activo. Esto implica que toda operación estará afectada por una verificación de la seguridad y por los procedimientos de auditoría establecidos para el sistema[224].

La interfaz de usuario (capa de presentación) es independiente de la capa de negocio y se basa en adaptaciones de los patrones MODEL VIEW CONTROLLER [POSA96] [Fowler02], PAGE CONTROLLER [Fowler02] y FRONT CONTROLLER [Fowler02] combinados con REFLECTION [POSA96], STRATEGY [GoF95] y DEPENDENCY INJECTION [Fowler04] para dar como resultado un sistema de visualización ágil, flexible y extensible[225].

La facha de servicios Web expone los contenidos del catálogo en forma débilmente acoplada e independiente de la plataforma (debido a las características inherentes a los servicios Web [W3C]). Esta interfaz expone funcionalidad a un mayor nivel de abstracción que la interfaz local de objetos del catálogo (la primera es una interfaz de grano grueso, mientras que la segundo es de grano fino).

En la capa de datos (que se refiere al componente pasivo del catálogo) puede observarse claramente tres elementos: un registro de entidades (contiene toda la información para localizar y clasificar entidades), un almacén de entidades (contiene las meta-especificaciones de las entidades) y un conjunto de ficheros de configuración. Esto refleja la arquitectura de despliegue utilizada para el componente pasivo del catálogo, donde toda la información de búsqueda, clasificación y catalogación se encuentra en bases de datos relaciones y las meta-especificaciones en ficheros[226].

En el **capítulo 16** se presentan en detalle los módulos principales de esta arquitectura, presentando los patrones y principios de diseño en que se funda cada uno y los mecanismos de extensibilidad que proveen.

[224] En principio, se registran los accesos a efectos de saber cuales son las entidades más populares y conocer pautas de uso de los patrones. De esta forma haciendo análisis de los datos podríamos llegar a determinar secuencias de consultas de patrones o criterios de afinidad a partir de la cercanía temporal de las consultas en una sesión de usuario.
[225] El motor de vistas dinámicas se presenta en detalle en el capítulo siguiente (§ 16.7).
[226] Esto se presenta en mayor detalle en el capítulo siguiente (§ 16.1)

15.2.1 Tecnologías Utilizadas

El prototipo ha sido enteramente desarrollado utilizando el marco de trabajo .NET [NET]. La interfaz Web y la fachada de servicios Web [W3C] han sido construidas con ASP.NET [MSAN], la interfaz de registro ha sido escrito usando WinForms [MSWF] y la lógica de la aplicación ha sido escrita en C#.

La aplicación ha sido implementada utilizando en forma conjunta las siguientes tecnologías:

- **Microsoft .NET** como plataforma de desarrollo.
    - **C#** como lenguaje de programación.
    - **ASP.NET** como marco de desarrollo para Web (UI y fachada de servicios Web).
        - **WebForms**: para construir las páginas que conforman la interfaz de usuario Web.
        - **Servicios Web** para poder exponer el catálogo a través de la Web.
    - **WinForms** para desarrollar el cliente de la interfaz de registro
- **XML** para representar los datos de configuración y metadatos de la aplicación.
- **Internet Information Server** como servidor Web, tanto para el cliente Web como para el servicio Web (fachada).
- **SQL Server** como motor de base de datos relacional para alojar la estructura del catálogo.
- **Index Server** para los catálogos de búsqueda

.

Capítulo 16
Implementación del Prototipo

"La falta de la humanidad de la computadora consiste en que una vez se le programa y trabaja adecuadamence, su honradez es intachable"
Isaac Asimov

En el capítulo anterior presentamos una breve descripción conceptual de los aspectos generales del prototipo desarrollado. En este capítulo estudiaremos la arquitectura, diseño e implementación de los módulos más relevantes, a efectos de ofrecer al lector una visión detallada del funcionamiento y alcance del prototipo. Luego presentaremos al lector algunas de las vistas y pantallas principales de la herramienta de visualización del catálogo del catálogo, a efectos de ofrecer una clara visión de cómo funciona y se utiliza el sistema. Finalmente se presentan algunos fragmentos seleccionados de meta-especificaciones EML para dar al lector una idea clara de cómo son las meta-especificaciones de patrones y entidades gestionadas por el sistema.

❖ ❖ ❖

16.1 Registro de Entidades

El registro de entidades es el punto de partida del sistema. Allí se encuentran registrados todos los patrones y conceptos de soporte. Para que un patrón o concepto esté disponible en el sistema debe estar incluido en el "***Registro de Entidades***"[227] (§ 13.3.1). El registro o repositorio de entidades es parte del componente pasivo del catálogo (§ 13.3).

16.1.1 Arquitectura del Registro de Entidades

El repositorio del registro combina una base de datos relacional [KS93] con un sistema de ficheros para almacenar a las entidades[228]. La estrategia de persistencia utilizada consiste en guardar las meta-especificaciones EML[229] completas en ficheros planos y las referencias a entidades (el registro y todas las estructuras necesarias para cumplir los requisitos de catalogación) en una base de datos relacional.

- **Base de Datos Relacional**: contiene información general de referencia a las entidades que permite recuperarlas, relacionarlas y realizar búsquedas. Adicionalmente contiene la gestión de usuarios, elementos de colaboración (foros) e información de auditoría sobre la utilización del sistema. Cada entidad tiene una referencia a un fichero donde esta alojada la meta-especificación EML completa.

- **Sistema de Ficheros**: contiene las meta-especificaciones EML completas de cada entidad (la entidad almacenada en la base de datos hace referencia al fichero donde se encuentra la meta-especificación). Contiene también imágenes asociadas a las meta-especificaciones (por ejemplo, gráficos de diagramas, fotos, etc.), definiciones de iteradores virtuales e información de configuración del sistema.

En la figura 16.1 a continuación se presenta un diagrama de arquitectura de alto nivel de la solución general para la plataforma de registro de entidades.

[227] Para incluir al patrón debe utilizarse la interfaz de registro, presentada más adelante en este capítulo
[228] Esta es una de las cuatro opciones de implementación presentadas en la sección 13.5.
[229] EML (Entity Meta-specification Language) se presenta en el capítulo 12 de esta tesis.

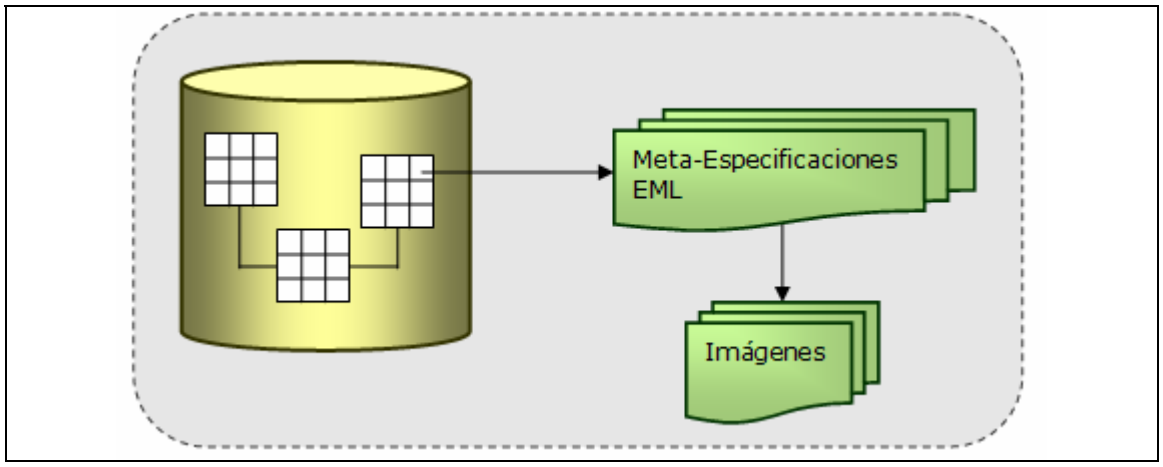


Figura 16.1 – Arquitectura general del registro de entidades.

Este esquema permite ver en todo momento el contenido de una entidad sin perder la potencia que ofrece un gestor de base de datos relacionales [KS93] para búsqueda, recuperación y vinculación de información (haciendo también efectivo el principio de diseño "*Mantener el conocimiento en texto plano*" [HT00]).

16.1.2 Estructura de Base de Datos del Registro de Entidades

La base de datos del registro de entidades contiene información de referencia[230] que permite localizar fácilmente a una entidad. Permite también representar las relaciones entre entidades, almacenar las etiquetas[231] y una tabla donde se almacenan el texto de las diferentes partes que conforman la entidad para realizar búsquedas por coincidencia de texto. En la figura 16.2 a continuación se presenta el diagrama entidad relación [KS93] del registro de entidades.

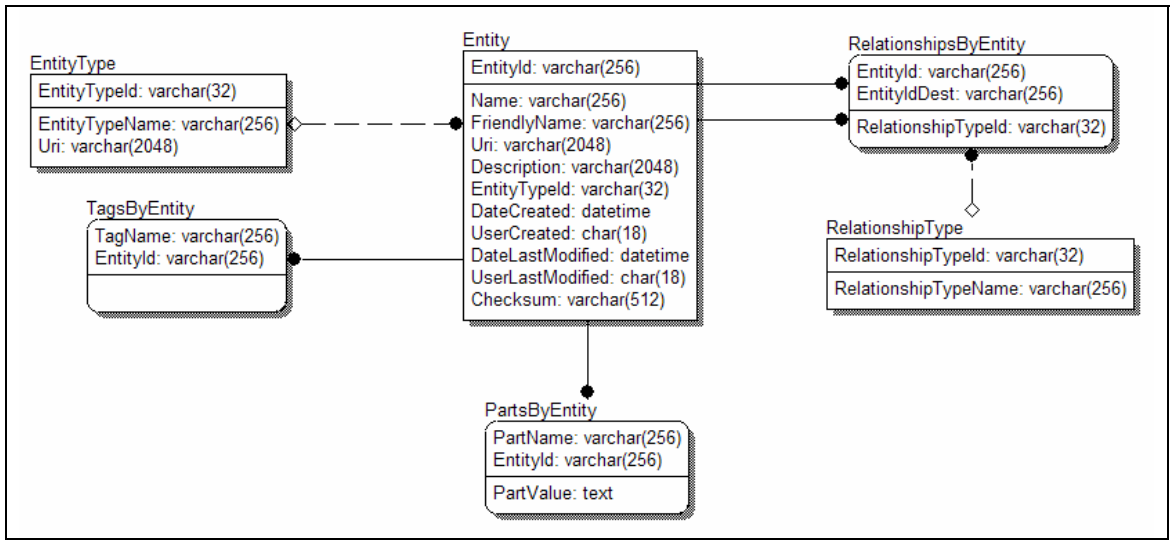


Figura 16.2 – Diagrama entidad relación de la base de datos del registro.

[230] La información de referencia es similar a la que expone el objeto "EntityReference", presentado más adelante en este capítulo.
[231] Tags (en inglés)

A continuación incluimos una breve explicación de cada tabla, a efectos de clarificar el diagrama presentado en la figura anterior

- **EntityType**: tipo de entidad (por ejemplo, patrón, autor, categoría, refactorización, etc.).

- **Entity**: referencia a una entidad. Contiene datos generales de referencia (nombre, resumen, tipo de entidad, etc.), datos de auditoría (autor y fecha de creación, fecha de ultima modificación, checksum, etc.) y un puntero al fichero donde se encuentra la meta-especificación EML completa (campo uri).

- **TagsByEntity**: etiquetas vinculadas a una entidad. Se utilizan para búsquedas. Pueden ser utilizadas también para categorización.

- **PartsByEntity**: bloques de texto plano (sin formato) de las varias partes que conforman a una entidad. Se utiliza para búsquedas por coincidencia de texto.

- **RelationshipsByEntity**: relaciones de una entidad con otras. Para cada una se indican los extremos y el tipo de relación.

- **RelationshipType**: tipos de relaciones (por ejemplo, "*es un*", "*es implementado por*", "*está incluido en*", etc.).

En el diagrama presentado en la figura 16.2 sólo se incluyen las tablas relacionadas con la gestión de referencias de entidades. Además de esta información, la base de datos contiene tablas para gestionar los siguientes módulos:

- Gestión Usuarios, Roles y Permisos
- Información de Auditoría
- Foros de Discusión

16.1.3 Estructura de Ficheros del Registro de Entidades

Las entidades registradas se almacenan en un sistema de ficheros. En la figura 16.3 se muestra la estructura de dicho sistema de ficheros:

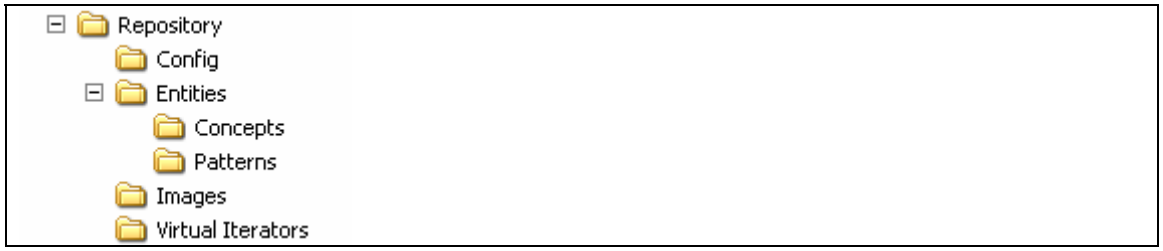


Figura 16.3 – Estructura de ficheros del registro de patrones.

Si bien las entidades están almacenadas en ficheros, se accede a estas a través de la referencia almacenada en la base de datos (presentada en la sección anterior).

16.2 La Interfaz de Registro de Entidades

El registro de una entidad se refiere a su inclusión o exclusión en el catálogo de entidades[232]. Una entidad puede ser registrada (incluida) o desregistrada (excluida). Una vez que una entidad ha sido registrada está lista para ser utilizada por los clientes del catálogo.

La interfaz de registro de entidades (presentada previamente en § 13.4.1) permite introducir, modificar o quitar entidades (patrón o concepto) al catálogo. Esta interfaz manipula los elementos pasivos, añadiendo nueva información o modificando la existente a través de un componente creado a tales efectos. Este componente tiene un acoplamiento significativo con la estructura del catálogo, dado que es el único punto a través del cual pueden modificarse sus contenidos. (§ 13.4.1)

La interfaz de registro tiene un GUI[233] (cliente pesado[234]) a través del cual los usuarios interactúan con el sistema. Este cliente permite cargar una entidad, revisar su contenido y finalmente, registrarla en el catálogo. Delega las funcionalidades de gestión y registro de entidades en los componentes generales y sólo se especializa en mostrar información al usuario y gestionar eventos de entrada y salida (en una arquitectura MVC [POSA96] hace las veces de *Vista* y *Controlador*).

En la figura 16.4 a continuación se muestra un ejemplo de la interfaz de usuario de este componente. En dicha figura la aplicación ha sido cargada con la meta-especificación EML del patrón COMPOSITE [GoF95] y en consecuencia muestra su estructura y contenido (mediante la utilización de un árbol para la estructura y una hoja de propiedades para los contenidos).

[232] El registro de una entidad se refiere a incluirla en el catálogo. El registro de entidades (16.1) se refiere al espacio donde se almacena la información sobre dichas entidades.
[233] Graphic User Interface (Intefaz Gráfica de Usuario)
[234] Fat Client (en inglés)

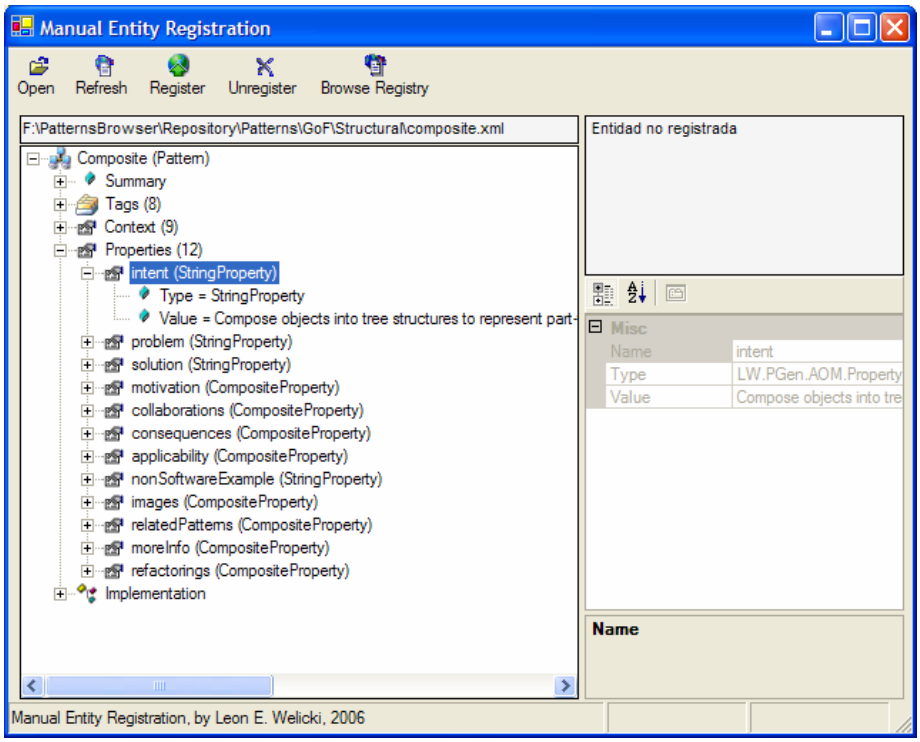


Figura 16.4 – Interfaz de registro de entidades (cliente pesado).

16.2.1 Funcionalidades de la Interfaz de Registro

El cliente para registro de entidades ofrece las siguientes opciones:

- **Registrar entidades**: registrar nuevas entidades o modificar entidades ya registradas en el catálogo.
- **Desregistrar entidades**: eliminar entidades existentes en el catálogo.
- **Navegar por los contenidos del catálogo**: navegar por los contenidos del catálogo y visualizar el contenido de las entidades registradas.
- **Visualizar una entidad EML en forma de árbol**: un fichero con una meta-especificación EML puede ser cargado utilizando esta interfaz y visualizado en forma de árbol (como se muestra en la figura 16.4, arriba).

16.3 Representación de Entidades

La representación de entidades en tiempo de ejecución es parte de FREP[235] (§ 13.4.2). Para representar las entidad en memoria en tiempo de ejecución hemos utilizado un modelo de objetos adaptativo (AOM). El modelo de objetos adaptativo (AOM) diseñado es una adaptación con varias mejoras respecto al modelo original propuesto en [AOM] [YBJ01] [YJ02] y [YJ03]. Las mejoras más significativas son la

[235] Flexible Runtime Execution Platform , presentada anteriormente en la sección 13.4.2

inclusión de un nivel abstracto de visualización (respecto de los dos niveles tradicionales existentes en el AOM [AOM] [YBJ01] [YJ02]) y la utilización de inyección de dependencias[236] [Fowler04] para poder "*inyectar*" módulos de código en tiempo de ejecución (estas inyecciones se realizan a partir de metadatos y reflectividad [POSA96] [Ortin02]). La implementación de las funcionalidades del nivel de implementación se delega en la capa de presentación [Fowler02].

La implementación propuesta se basa principalmente en la forma canónica de AOM (**capítulo 8**), principalmente en los patrones PROPERTY [FY98], TYPE OBJECT [JW98] y en una adaptación del patrón TYPE SQUARE [YBJ01] a nuestro dominio de problema. Contiene varios tipos de objetos "*propiedad*" orientados a representar los diferentes elementos que pueden conformar la plantilla de un patrón. Estas propiedades pueden ser compuestas en forma recursiva para representar estructuras complejas. Las tipos de propiedades en el sistema no son un conjunto cerrado: pueden incluirse nuevos tipos de propiedades extendiendo a la propiedad base. Para gestionar las relaciones entre entidades hemos utilizado una variante del patrón ACCOUNTABILITY [Fowler96].

16.3.1 Representación del Nivel de Conocimiento

El nivel de conocimiento (plantilla, relaciones, etiquetas, etc.) es representado utilizando una instancia de un AOM. Este AOM contiene propiedades especiales para cada uno de los tipos de elementos que pueden ser adjuntados a una plantilla, siguiendo los patrones TYPE SQUARE, TYPE OBJECT y VALUE OBJECT. En el AOM se pueden crear distintos tipos de entidades derivando de la clase `Entity`. Para crear nuevas propiedades debe hacerse lo mismo con la clase `Property`. Es posible crear propiedades complejas a partir de la composición dinámica de propiedade. El tipo de propiedad elemental es `StringProperty`, que puede alojar una cadena de texto. Existe una propiedad destinada a ser un contenedor genérico (`CompositeProperty`) que permite agrupar cualquier subtipo de propiedad (es una implementación del patrón COMPOSITE [GoF95]). Existen otros tipos de propiedades especiales[237] especializadas en la representación de distintos aspectos de una entidad (imágenes, consecuencias, más información, etc.).

Las relaciones entre entidades se representan utilizando el patrón ACCOUNTABILITY. Cada relación es una instancia de un `Accountability`, permitiendo crear tipos de relaciones concretos. Debido a que las relaciones son entre tipos de entidades genéricos, se pueden relacionar elementos de diferentes tipos (lo cual permite relacionar a los patrones con los conceptos de soporte). Adicionalmente, se pueden crear tipos de relaciones (`AccountabilityType`) donde pueden incluirse reglas especiales para validación, restricción, composición, etc.

En la siguiente figura se muestra un diagrama de clases general del AOM que alberga el nivel de conocimiento de los patrones y entidades. En esta figura se

[236] Dependency Injection (en inglés)
[237] Estas propiedades alojan las propiedades que se definen mediante los "nombre reservados" mencionados en § 12.2.4.1.2

incluye también el nivel de visualización, cuya responsabilidad es la forma en que se muestra la información alojada en el AOM.

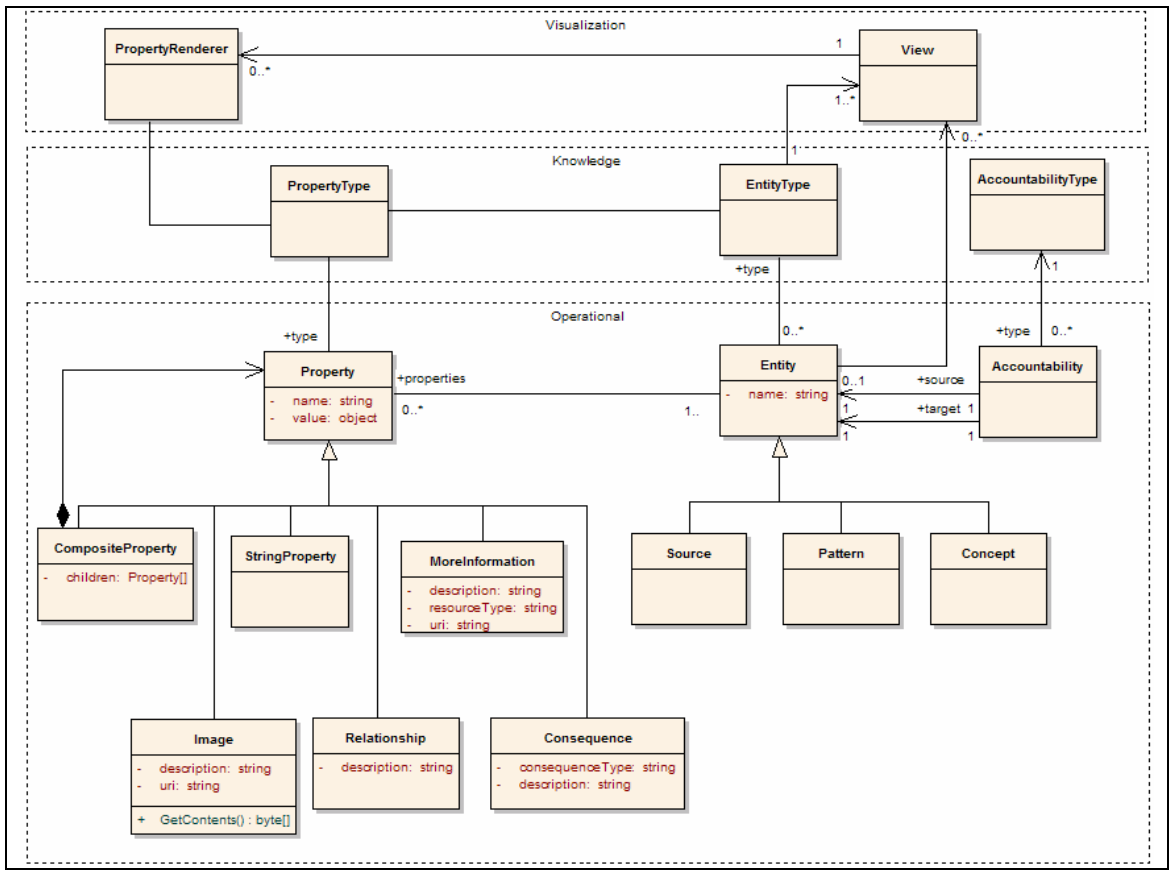


Figura 16.5 – Diagrama de clases del AOM para representar el nivel de conocimiento

16.3.2 Representación del Nivel de Implementación

Las meta-especificaciones de entidades pueden tener asociado un componente de comportamiento que se describe mediante el uso conjunto de EML-SDL (§ 12.2.4.4) y EML-BDL (§ 12.2.4.5). La estructura en tiempo de ejecución de las entidades incluye contenedores para albergar esta información.

Las entidades que tienen estructura tienen un conjunto de participantes (`Participant`), que son los participantes en la solución al problema planteado por el patrón y por tanto los contenedores de dicho comportamiento. Un participante puede[238] tener asociado un módulo de código (`CodeModule`), que es la entidad que aloja elementos de comportamiento de bajo nivel. Estos elementos pueden ser propiedades (`Property`) o métodos (`Method`) y se agrupan en tres miembros de `CodeModule` (`properties`, `constructors` y `methods`). Los métodos pueden recibir parámetros (`Parameter`). Finalmente, dentro de estos elementos de comportamiento de bajo nivel (`Property` y `Method`) se almacena una cadena donde se especifica el comportamiento en EML-BDL.

[238] En los casos en que tiene estructura (EML-SDL) y no comportamiento (EML-BDL), el participante no tiene asociado el módulo de código. Este caso se da para los patrones de [Evitts99]

En la siguiente figura se muestra el diagrama de clases UML de la estructura de representación del comportamiento de las entidades.

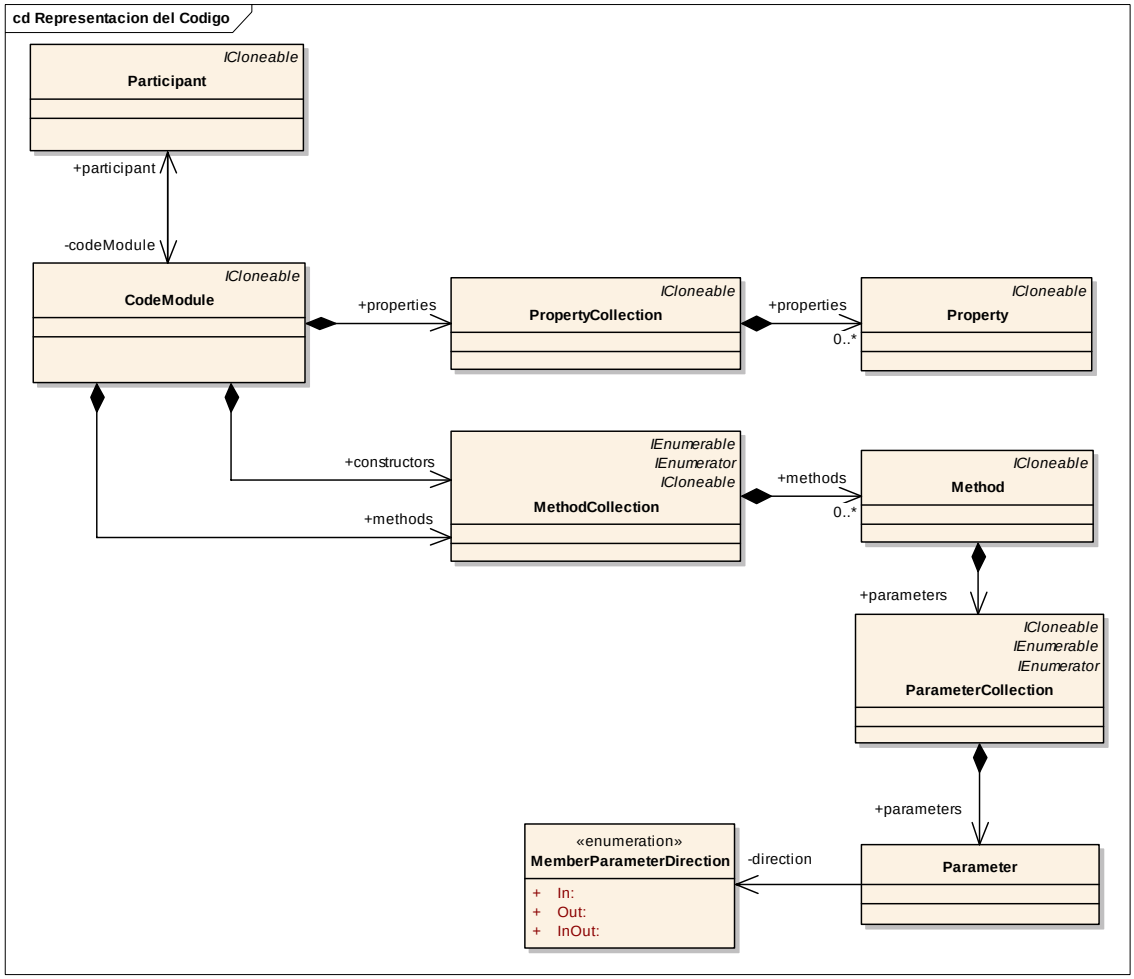


Figura 16.6 – Diagrama de clases UML para representar el comportamiento de una entidad.

16.3.2.1 Detalle de las Clases

- **Participant**: representa a un participante de la solución. Los participantes permiten definir la estructura de la implementación del comportamiento de una entidad. El comportamiento del participante se describe en `CodeModules`.

- **CodeModule:** representación de un módulo de código, que contiene agrupaciones de unidades de representación de comportamiento (instancias `Property` y `Method`).

- **Property:** representación de un atributo en un módulo de código. Permite describir al atributo con su tipo y un conjunto de metadatos que definen su nivel de protección y modificares asociados. Puede contener comportamiento EML-BDL asociado.

- **Method**: representación de un método. Permite describir la firma del método y su comportamiento (mediante EML-BDL).

- **Parameter:** parámetro de entrada o salida para un método (`Method`). Puede especificarse su dirección, tipo y nombre.

- **PropertyCollection:** colección de propiedades (`Property`).

- **MethodCollection:** colección de metodos (`Method`).

- **ParameterCollection:** colección de parámetros (`Parameter`).

16.4 El Analizador de Entidades

El analizador de entidades (§ 13.4.2.2) es parte de FREP[239] (§ 13.4.2) y es el encargado de analizar los ficheros EML e hidratar el modelo de objetos adaptativo para la representación de entidades. El analizador está implementado sobre una arquitectura flexible que ofrece varios puntos de extensibilidad.

16.4.1 Arquitectura y Diseño

Los patrones fundamentales que guían la arquitectura de este módulo son PIPES & FILTERS [POSA96] y DEPENDENCY INJECTION [Fowler04]. Los principios que guían la arquitectura son *Dependency Inversion Principle*, *Open-Closed Principle* y *Liskov Substitution Principle* [Martin02].

Para que una clase sea reconocida como analizador de entidades por el sistema debe implementar la interfaz `IEntityParser`. Esta interfaz contiene el contrato que especifica el comportamiento de los analizadores de entidades. En el prototipo hemos creado una implementación de un analizador basado en una extensión del patrón PIPES & FILTERS [POSA96]. En este caso, los filtros se crean y vinculan dinámicamente (utilizando las capacidades reflectivas de .NET [Box02] [Lowy03]) a partir de metadatos almacenados en un fichero de configuración. Para que una clase sea reconocida como filtro debe implementar la interfaz `IParseStep` (paso de análisis). Cada paso en el análisis es un filtro (especializado en la realización de una porción concreta del análisis) y la combinación de un conjunto de filtros produce como resultado una instancia de un AOM a partir de una entidad EML. El contenedor que aloja y que orquesta la ejecución de los pasos de análisis (filtros) es la clase `ParsePipeline`. Para la validación léxica y sintáctica [Cueva98] de EML se utilizan esquemas XML [W3C]. En la figura 16.7 se muestra el diagrama de clases UML del módulo de análisis de entidades EML.

[239] Flexible Runtime Execution Platform , presentada anteriormente en la sección 13.4.2

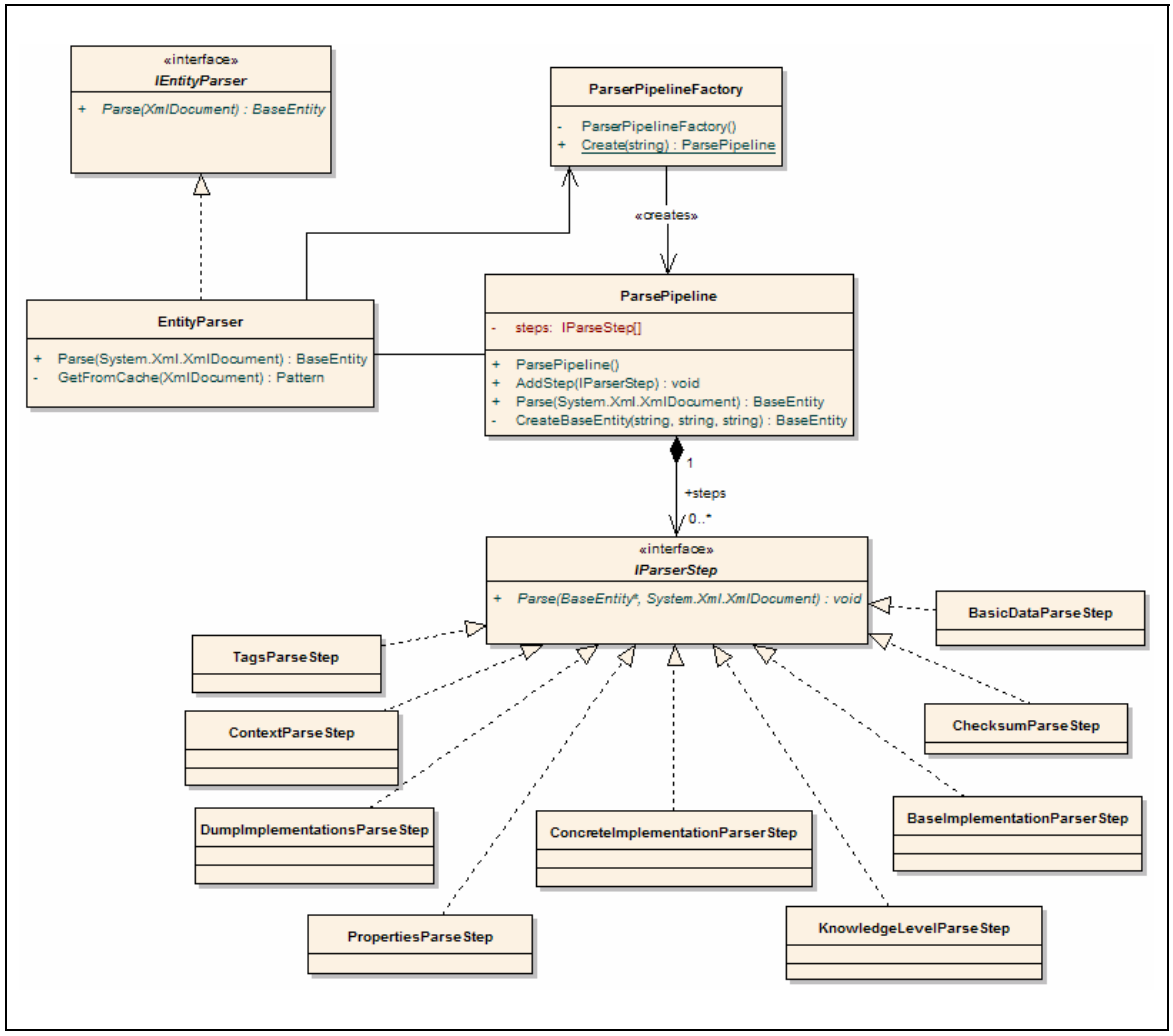


Figura 16.7 – Diagrama de clases UML del analizador de patrones.

16.4.2 Detalle de las Clases

- **IEntityParser**: Interfaz que define el contrato que deben cumplir los analizadores de entidades.

- **EntityParser**: Implementación de un analizador de entidades que delega el análisis en un pipeline configurable. Implementa la interfaz `IEntityParser`.

- **ParserPipelineFactory**: tiene la responsabilidad de crear instancias del pipeline de análisis de entidades. Tiene el conocimiento para manipular el fichero que determina que pasos ha de tener un analizador para un tipo específico de entidad y a partir de esos datos construye una instancia del pipeline. Para crear las instancias de los pasos utiliza las capacidades reflectivas de .NET [Box02] [Lowy03]. Esta clase es una implementación del patrón SIMPLE FACTORY [Welicki05g].

- **ParsePipeline**: pipeline para realizar el análisis de una entidad. Aloja un conjunto de pasos (clases que implementan `IParseStep`), a los cuales ejecuta en forma secuencial para realizar el análisis de una entidad. Previamente a la ejecución de esos pasos valida a la entidad utilizando esquemas XML [W3C]. Las instancias de esta clase son construidas por `ParsePipelineFactory`. La implementación del pipeline combina a los patrones PIPES AND FILTERS [POSA96] y STRATEGY [GoF95].

- **IParseStep**: Interfaz que define el contrato que deben implementar los pasos de análisis en el `ParsePipeline`.

- **Implementaciones de Pasos de Análisis**: implementación de los pasos concretos del análisis. Cada clase se especializa en el análisis de un aspecto específico de la entidad. Los pasos de análisis deben implementar la interfaz `IParseStep`. Como norma, deben tener como sufijo en su nombre la cadena "`ParseStep`". Estos pasos se invocan mediante el método `Parse` de `ParsePipeline`.

16.4.3 Configuración del Analizador

Los pasos de análisis a realizar para cada tipo de entidad se configuran en un fichero llamado `ParserInfo.xml`. En este fichero se define la lista ordenada de pasos a realizar para cada tipo de entidad. En el siguiente bloque de código se muestra la configuración del analizador en el prototipo:

```
<parserConfig>

  <entityType id="Pattern">
    <parseStep type="LW.PGen.Parser.BasicDataParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.PropertiesParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.TagsParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.ContextParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.KnowledgeLevelParserStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.BaseImplementationParserStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.ConcreteImplParserStep, LW.PGen"/>
  </entityType>

  <default>
    <parseStep type="LW.PGen.Parser.BasicDataParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.PropertiesParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.TagsParseStep, LW.PGen"/>
    <parseStep type="LW.PGen.Parser.ContextParseStep, LW.PGen"/>
  </default>

</parserConfig>
```

Código 16.1 – Configuración del analizador de entidades EML.

En este fichero se determina la siguiente regla de configuración: si la entidad que se está analizando es un patrón, aplicar los pasos que se especifican para ese tipo de entidad (especificados en la sección `entityType id="Pattern"`). Para cualquier

otra entidad, aplicar los pasos predeterminados (especificados en la sección "`default`")

16.4.4. Extensibilidad

El módulo de análisis ofrece varios puntos de extensibilidad. En esta sección se presentan los tres más importantes:

- Añadir nuevos pasos al análisis.
- Establecer la secuencia de pasos a realizar para analizar un tipo de entidad.
- Crear un nuevo analizador.

16.4.4.1 Añadir un Nuevo Paso de Análisis

Para crear un nuevo paso de análisis y vincularlo al análisis completo de una entidad deben realizarse en forma ordenada los siguientes pasos:

1. **Crear la implementación paso**: para crear un nuevo paso de análisis debe crearse una clase que implemente la interfaz `IParseStep`. Esta clase puede estar codificada en cualquier lenguaje debido a las capacidades multilenguajes de .NET [Meyer02] [Meyer02b]. El nombre de la clase debe tener como sufijo "`ParseStep`".

2. **Registrar el paso**: para registrar el paso, hay que incluirlo en la lista de pasos a ejecutar para un tipo de entidad en el fichero `ParserInfo.xml`. Una vez registrado el paso en el fichero la fábrica de pipelines de análisis (`ParserPipelineFactory`) lo incluirá en la lista de pasos a ejecutar en una instancia de un analizador.

Para modificar un paso sólo es necesario realizar una variación del primer punto de la lista presentada arriba, que consiste en modificar el código del filtro adecuado (en el contexto del patrón de arquitectura PIPES & FILTERS [POSA96]).

16.4.4.2 Especificar la Secuencia de Pasos para Analizar un Tipo de Entidad

Para especificar una nueva secuencia de pasos a realizar para analizar una entidad es necesario modificar el fichero `ParserInfo.xml` (§ 16.4.3). Para añadir una secuencia de pasos de análisis relacionados con un tipo de entidad es necesario añadir una nueva sección `entityType` y dentro de ésta especificar en forma ordenada cada uno de los pasos a realizar (en un elemento `parseStep`).

16.3.4.3 Crear un Nuevo Analizador

Para crear un nuevo analizador sólo es necesario crear una clase que implemente la interfaz `IEntityParser`. El comportamiento de esta clase será determinado por su

diseñador en forma totalmente arbitraria. Por ejemplo, podría crearse un nuevo analizador que no utilice el pipeline y realice el análisis mediante otra técnica[240].

16.5 Iteradores Virtuales

Los *Iteradores Virtuales* (§ 13.4.4) son una extensión al patrón ITERATOR [GoF95] que permiten recorrer el catálogo a partir de reglas codificadas en metadatos. Estos metadatos se codifican utilizando un DSL[241] especial (§ 13.4.4.1), donde se indica qué relaciones se utilizarán para obtener los objetos en cada nivel. Los iteradores virtuales soportan navegación en múltiples niveles y en consecuencia pueden ser utilizados para recorrer estructuras jerárquicas.

16.5.1 Arquitectura y Diseño

Los patrones que guían la arquitectura de este módulo son el COMPOSITE [GoF95], el BUILDER [GoF95] y el VIRTUAL PROXY [POSA96].

Los iteradores virtuales (`VirtualIterator`) consisten de un conjunto de referencias a entidades (`EntityReference`) agrupados jerárquicamente. Cada referencia hace las veces de VIRTUAL PROXY [POSA96] y es a su vez una implementación del patrón COMPOSITE [GoF95]. Existe una clase que tiene la responsabilidad de construir los iteradores virtuales a partir de metadatos de configuración (`VirtualIteratorBuilder`) a través de la implementación de un BUILDER [GoF95] simplificado (como se presenta en [Kerievsky04]). En la figura 16.8 a continuación, se muestra el diagrama de clases UML del módulo de Iteradores Virtuales.

[240] Este mecanismo de extensibilidad nos permitió tener un analizador monolítico en la primera versión del prototipo y luego refactorizarlo a una arquitectura de Pipes and Filters sin necesidad de modificar al resto de clases que utilizaban al analizador (gracias al acoplamiento abstracto y a la inversión de dependencias)
[241] DSL (Domain Specific Language). Este concepto ha sido estudiado previamente en el capítulo 7.

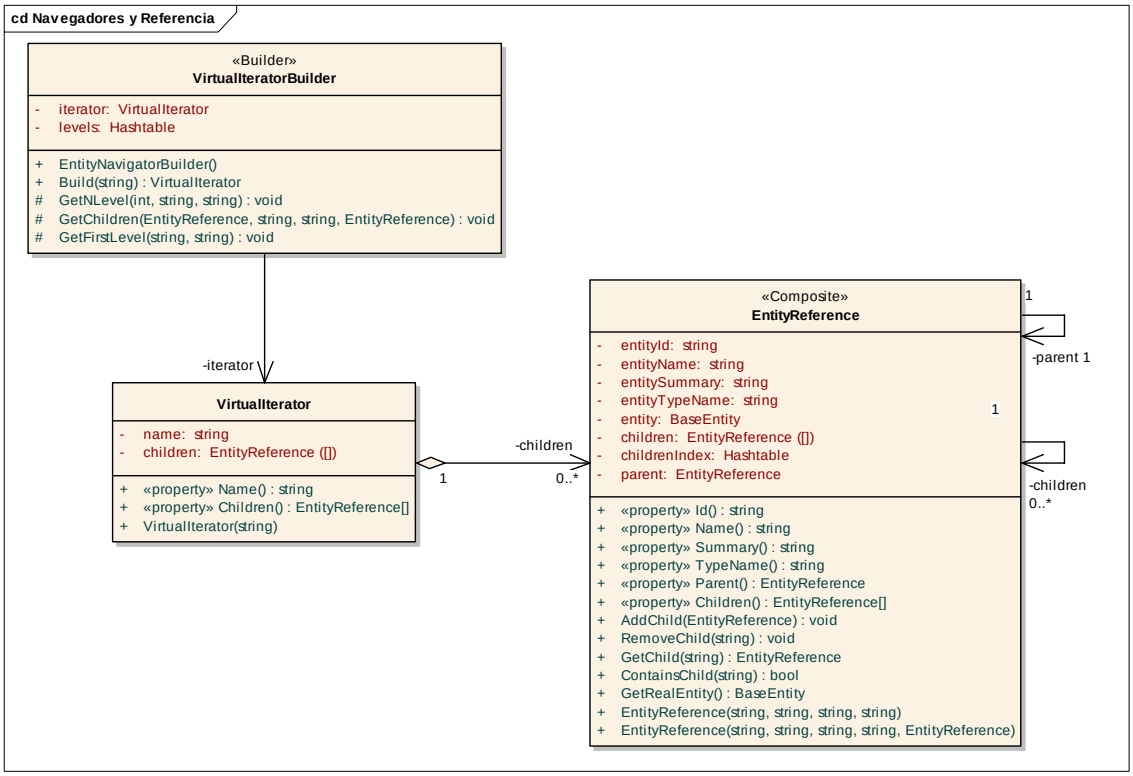


Figura 16.8 – Diagrama de clases UML del módulo de iteradores virtuales

16.5.2 Detalle de las Clases

Las clases `VirtualIterator` y `EntityReference` son serializables (soportan serialización SOAP), con lo cual esta estructura puede ser enviada a través de un servicio Web o de tecnologías de objetos distribuidos como .NET Remoting [NetRemoting].

- **VirtualIteratorBuilder**: es el encargado de construir las instancias de los iteradores virtuales a partir de los metadatos de configuración. Es una implementación del patrón BUILDER [GoF95] como se propone en [Kerievsky04].

- **VirtualIterator**: instancia del iterador virtual. Contiene un conjunto de referencias a entidades (que a su vez, contienen otras referencias). Los iteradores virtuales tiene una propiedad "`Name`" que permite identificarlos.

- **EntityReference**: referencia a una entidad. Contiene un método llamado `GetRealEntity` que permite obtener la entidad real a la que se hace referencia (es un VIRTUAL PROXY [POSA96] que funciona en forma similar a los proxies del CLR de .NET [Box02]). Es una implementación del patrón COMPOSITE [GoF95] simplificada como se propone en [Kerievsky04].

16.5.3 Configuración de los Iteradores Virtudes

Como mencionamos al inicio de esta sección, los iteradores se construyen a partir de metadatos codificados con un DSL creado a tales efectos (sección 13.4.4.1). En el siguiente bloque de código se muestra el fichero completo de definición utilizado en el prototipo (`Iterators.xml`), donde se especifican cuatro iteradores.

```
<?xml version="1.0" encoding="utf-8"?>

<iterators>
  <iterator name="Pattern Language">
    <level
      id="1"
      entityTypeId="PatternLanguage"
      patternRelationshipTypeId="isContainedIn"/>
    <level
      id="2"
      entityTypeId="Category"
      relationshipTypeId="isContainedIn"
      patternRelationshipTypeId="isContainedIn"/>
  </iterator>

  <iterator name="Source">
    <level
      id="1"
      entityTypeId="Source"
      patternRelationshipTypeId="isPublishedIn"/>
    <level
      id="2"
      entityTypeId="Category"
      relationshipTypeId="isPublishedIn"
      patternRelationshipTypeId="isContainedIn"/>
  </iterator>

  <iterator name="Author">
    <level
      id="1"
      entityTypeId="AuthorGroup"
      patternRelationshipTypeId="wasWrittenBy"/>
  </iterator>

  <iterator name="OO Principle">
    <level
      id="1"
      entityTypeId="OOPrinciple"
      patternRelationshipTypeId="conforms"/>
  </iterator>

</iterators>
```

Código 16.2 – Configuración de los Iteradores Virtuales en el prototipo.

16.5.4 Extensibilidad: Añadiendo, Modificando y Quitando Iteradores Virtuales

Para añadir, modificar o eliminar iteradores virtuales no es necesario escribir código: todos estos cambios se realizan en forma declarativa (mediante la manipulación de metadatos). Sólo es necesario modificar el fichero `Iterators.xml`, añadiendo, modificando o eliminando la especificación del iterador en cuestión.

16.6 Generación de Código

La generación de código se realiza a partir de la representación abstracta de las entidades (nivel de implementación). Para generar código a partir de un patrón, este debe haber pasado previamente por el analizador (§13.4.2.2, § 16.4) y debemos tener una representación en memoria de la entidad (§13.4.2.1, § 16.3). Como precondición la entidad debe incluir información sobre la estructura y su implementación (nivel de implementación completo). En la versión inicial del prototipo, se proveen tres generadores de código: C#, VisualBasic.NET y Java.

16.6.1 Arquitectura y Diseño

La generación de código se realiza mediante de la traducción[242] de EML (en concreto EML-SDL y EML-BDL) a un lenguaje destino. Para que un objeto sea reconocido como generador, sólo debe implementar la interfaz `ICodeGenerator`, que tiene un único método llamado `GenerateCode`, que retorna una cadena con el código fuente completo del patrón. Cualquier objeto que herede esta interfaz puede ser registrado en el sistema como un generador de código válido (el registro se realiza añadiendo una entrada en el fichero de generadores, que se explica en § 16.6.3). Cada clase generadora se especializa en la generación de código para un lenguaje específico. Las clases generadoras deben llevar el sufijo "`CodeGen`".

Adicionalmente, se ha desarrollado una clase ayudante[243] basada en el patrón TEMPLATE METHOD [GoF95] para facilitar el desarrollo de traductores de EML a lenguajes orientado a objetos. Los traductores incluidos en el prototipo han sido escritos utilizando C#.

En la siguiente figura, se muestra la estructura estática (mediante un diagrama de clases UML) del módulo de generación de código.

[242] Un traductor es un programa que procesa un texto fuente y genera un texto objeto [Cueva98]
[243] Helper Class, en inglés

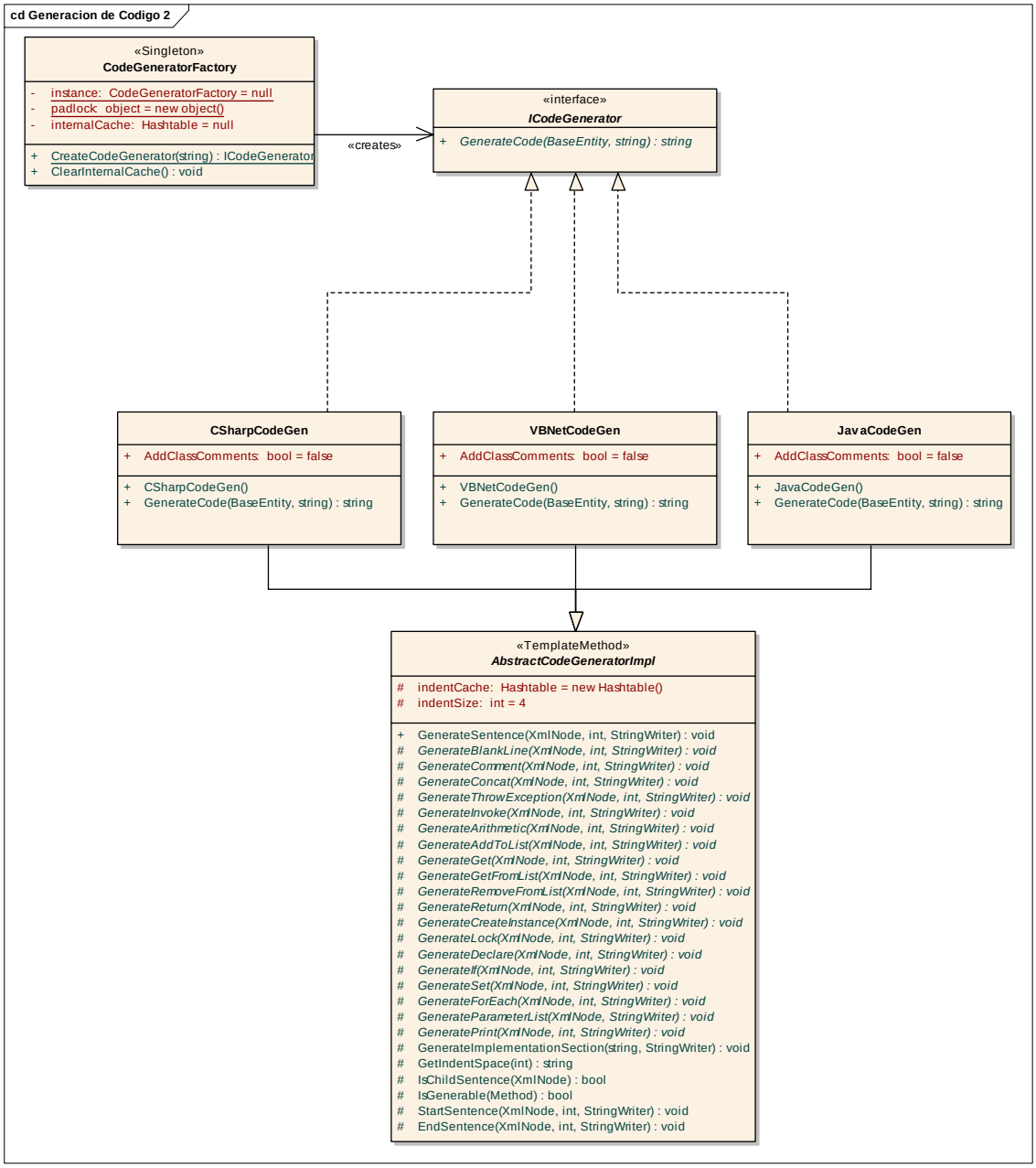


Figura 16.9 – Diagrama de clases UML del módulo de generación de código.

16.6.2 Detalle de las Clases e Interfaces

- **CodeGeneratorFactory**: Clase de fabricación, encargada de crear instancias de generadores. Dado el nombre del generador a crear[244], obtiene la información del fichero de configuración de los generadores (§ 16.6.3) y utiliza las capacidades reflectivas de .NET [Box02] [Lowy03] para crear las instancias. Contiene una cache de instancias, a efectos de optimizar los tiempos de creación. Es una implementación de los patrones SINGLETON [GoF95] y SIMPLE FACTORY [Welicki05g].

[244] El nombre del generador coincide con el nombre del lenguaje destino

- **ICodeGenerator**: Interfaz que define el contrato que deben cumplir todos los generadores de código. Para que una clase sea identificada en el sistema como generador de código debe implementar esta interfaz.

- **AbstractCodeGeneratorImpl**: Clase ayudante[245]. Contiene funciones generales para asistir a la traducción de EML-DBL a un lenguaje objetivo. Tiene el rol de `AbstractClass` en el patrón TEMPLATE METHOD [GoF95].

- **Implementación de los Traductores**: Traductores de EML-DBL a lenguajes de programación de alto nivel. Implementan la interfaz `ICodeGenerator` y pueden utilizar los servicios de `AbstractCodeGeneratorImpl` (cuando los utilizan tienen el rol de `ConcreteClass` en el patrón TEMPLATE METHOD [GoF95]. En la figura 16.10 hay tres: `CSharpCodeGen` (C#), `VBNetCodeGen` (VB.NET) y `JavaCodeGen` (Java). Es importante aclarar que no es obligatorio que los traductores utilicen los servicios de `AbstractCodeGeneratorImpl`. Esta clase es una ayuda que ofrece el framework para simplificar la traducción, pero es decisión de quién implementa el traductor si desea o no utilizarla. Como norma, los traductores deben tener como sufijo en su nombre la cadena "`CodeGen`".

16.6.3 Configuración de los Generadores de Código

El fichero `Generators.xml` contiene la lista de los generadores registrados en el sistema. Pueden registrarse nuevos generados en tiempo de ejecución agregando entradas a este fichero de configuración. En el bloque de código 16.3 a continuación se muestra su contenido.

```
<codeGenerators>
  <codeGenerator
    name="CSharp"
    displayName="C#"
    language="C# 1.1"
    assembly="LW.PGen.CodeGen"
    className="LW.PGen.CodeGen.CSharpCodeGen"/>
  <codeGenerator
    name="VBNET"
    displayName="VB.NET"
    language="VB.NET 1.1"
    assembly="LW.PGen.CodeGen"
    className="LW.PGen.CodeGen.VBNetCodeGen"/>
  <codeGenerator
    name="Java"
    displayName="Java"
    language="Java"
    assembly="LW.PGen.CodeGen"
    className="LW.PGen.CodeGen.JavaCodeGen"/>
</codeGenerators>
```

Código 16.3 – Configuración del registro de generadores de código.

[245] Helper Class (en inglés)

En cada entrada se especifica el nombre del generador, un nombre amigable (para mostrar al usuario), el nombre del lenguaje de programación destino, el ensamblado (dll[246]) [Box02] donde puede encontrarse el generador y finalmente, el nombre de la clase del generador.

16.6.4 Extensibilidad: Añadir, Modificar y Quitar Generadores

Para crear un nuevo generador de código (traductor de EML-BDL y EML-SDL a algún lenguaje de alto nivel) y registrarlo en el sistema deben realizarse en forma ordenada los siguientes pasos:

1. **Crear la implementación del generador**: para crear un nuevo traductor debe crearse una clase que implemente la interfaz `ICodeGenerator`. Esta clase puede estar codificada en cualquier lenguaje debido a las capacidades multilenguaje de .NET [Meyer02] [Meyer02b]. El nombre de la clase debe tener como sufijo "`CodeGen`". Si el desarrollador lo considera oportuno puede derivar el nuevo generador de la clase ayudante `AbstractCodeGeneratorImpl`, que contiene un esqueleto genérico de un traductor siguiendo la guía del patrón TEMPLATE METHOD [GoF95].

2. **Registrar el generador**: para que el generador esté disponible en el sistema hay que incluirlo en la lista de generadores disponibles en el fichero `Generators.xml`. Una vez registrado el traductor ya está listo para ser utilizado en el sistema y podrán crearse instancias dinámicamente de éste mediante la clase `CodeGeneratorFactory`.

Para modificar un traductor existente sólo es necesario realizar una variación del primer paso, que consiste en modificar la implementación del traductor apropiado. Para eliminar un traductor basta con quitarlo del fichero `Generators.xml` (aunque es aconsejable quitar también el ensamblado donde se encuentra la implementación si no es utilizado).

16.7 Motor de Vistas Dinámicas

El motor de vistas dinámicas permite adjuntar vistas dinámicamente a las entidades en tiempo de ejecución. Dada una entidad (patrón o concepto) al aplicarle una vista se obtiene el código de presentación para mostrarla al usuario. Cada vista se focaliza en un aspecto concreto de la entidad y es posible tener tantas vistas como aspectos relevantes tengamos en consideración.

16.7.1 Diseño y Arquitectura

El elemento principal del motor de vistas es la interfaz `IView`, que define el contrato que debe cumplir toda vista disponible en el sistema. Esta interfaz tiene

[246] DLL (Dynamic Link Library)

dos métodos, uno para envío de parámetros (`SetParameters`) y otro para generar la salida (`Render`). Todas las vistas deben implementar esta interfaz. Las vistas se registran en el sistema en la misma base de datos relacional donde se registran las entidades (§ 16.1.2).

En la figura 16.10 se muestra la estructura estática (mediante un diagrama de clases UML) del módulo de vistas dinámicas.

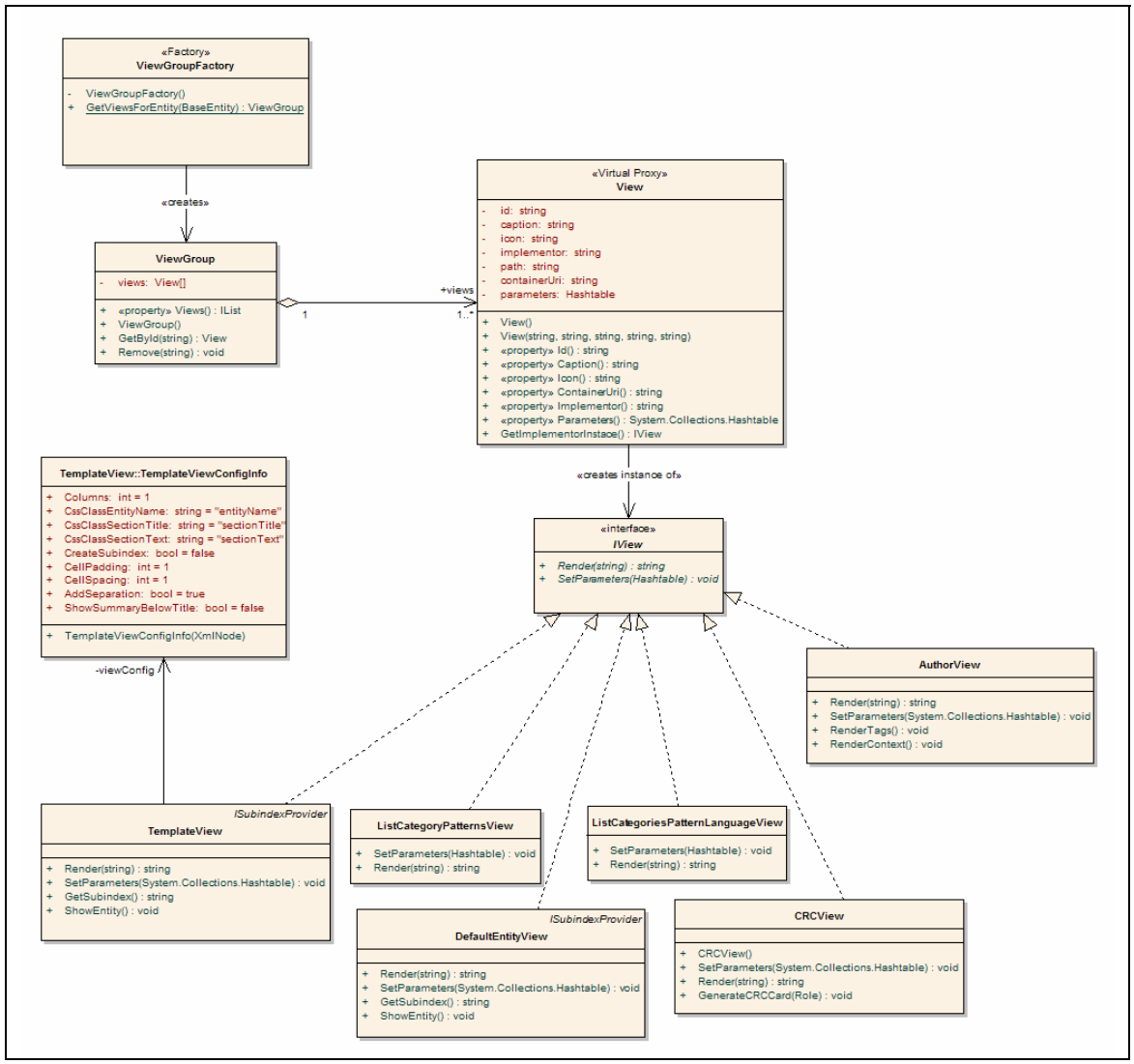


Figura 16.10 – Diagrama de clases UML del módulo de vistas dinámicas.

16.7.2 Detalle de las Clases e Interfaces

- **IView**: Interfaz que define el contrato que deben cumplir todas las vistas. Para que una clase sea identificada en el sistema como una vista debe implementar esta interfaz.

- **View**: Referencia a una vista. Contiene información de referencia sobre la vista (nombre, contenedor, nombre del tipo del implementador, etc.). Contiene un método especial (`GetImplementorInstance`) que crea y

retorna una instancia a partir de la información de referencia de la vista utilizando las capacidades reflectivas de .NET (retorna una instancia de una clase que implementa `IView`). Es un VIRTUAL PROXY [POSA96].

- **ViewGroup**: Grupo de objetos de referencia a vistas (`View`). Es posible obtener el grupo de vistas para una entidad y por lo tanto conocer las posibilidades de visualización existentes.

- **ViewGroupFactory**: Clase de fabricación [Welicki05g] que crea instancias de grupos de vistas. Dada una entidad retorna el grupo de referencias a vistas (instancias de `View`) que pueden aplicarse a ésta. Es una implementación del patrón SIMPLE FACTORY [Welicki05g].

- **Implementación de las Vistas**: implementaciones concretas de las vistas. Una vista debe implementar la interfaz `IView`. En la figura 16.10 se muestran algunas de las vistas existentes en el sistema como por ejemplo la vista de tarjetas CRC, la vista de entidades, la vista de patrones por categoría, etc. Como norma, los traductores deben tener como sufijo en su nombre la cadena "`View`".

16.7.3 Configuración de las Vistas

La asociación entre vistas y entidades se realiza mediante tablas almacenadas en la base de datos donde se encuentran las referencias a las entidades (§ 16.1.2). Las vistas pueden asociarse a una entidad concreta (**asociación específica**) o a un tipo de entidad (**asociación general**). En la figura 16.11 a continuación se muestra el esquema donde se almacenan esa información.

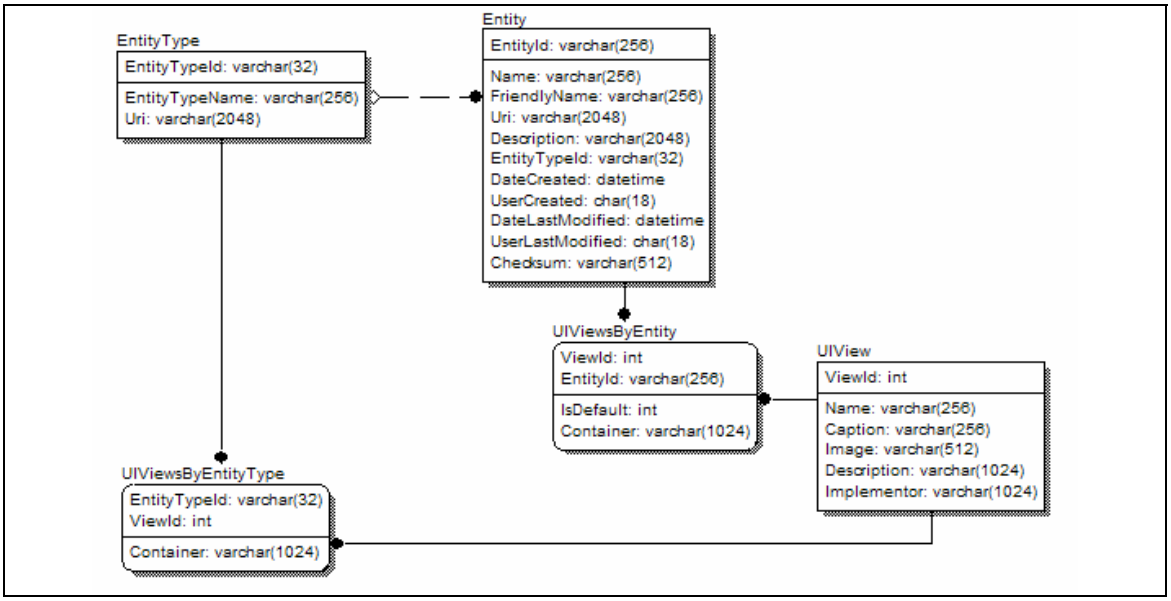


Figura 16.11 – Diagrama Entidad Relación de la estructura del repositorio de vistas.

A continuación se explican brevemente las tablas relacionadas con la gestión de las vistas:

- **UIView**: contiene la definición de las vistas en el sistema. Para cada vista se especifica un identificador, un nombre, una descripción y un implementador[247].

- **UIViewsByEntityType**: asociación entre vistas y tipos de entidades. Este tipo de asociación se denomina "*asociación general*".

- **UIViewByEntity**: asociación entre vistas y entidades. Este tipo de asociación se denomina "*asociación específica*". Si una entidad tiene elementos en esta tabla no se utilizan en ese caso los de `UIViewsByEntityType`.

16.7.4 Extensibilidad

El módulo de vistas dinámicas ofrece varios puntos de extensibilidad. En esta sección se presentan los tres más importantes:

- Crear nuevas vistas.
- Asociar vistas a un tipo de entidad (asociación general).
- Asociar vistas a una entidad (asociación específica).

16.7.4.1 Crear Nuevas Vistas

Para crear una nueva vista y registrarla en el sistema deben realizarse en forma ordenada los siguientes pasos:

1. **Crear la implementación de la vista**: para crear una nueva vista debe crearse una clase que implemente la interfaz `IView`. Esta clase puede estar codificada en cualquier lenguaje de programación[248] debido a las capacidades multilenguajes de .NET [Meyer02] [Meyer02b]. El nombre de la clase debe tener como sufijo "`View`".

2. **Registrar la vista**: para que la vista esté disponible en el sistema hay que incluirlo en la lista de vistas en la tabla `UIViews` (ver figura 16.11). Una vez registrada, ya está lista para ser asociada a entidades (asociación específica) o tipos de entidades (asociación general).

3. **Asociar la vista**: para que la vista se aplique es necesario asociarla a un tipo de entidad o a una entidad concreta (como se explica en § 16.7.3).

Para modificar el comportamiento de una vista paso, sólo es necesario realizar una variación del paso uno, que consiste en modificar la implementación de la vista. Para eliminar una vista del sistema es necesario desasociarlo de las entidades y tipos

[247] El implementador es la clase que contiene la implementación de la vista. El módulo de vista puede crear instancias del implementador en tiempo de ejecución utilizando las capacidades reflectivas de .NET.
[248] Cualquiera de los lenguajes soportados por .NET

(tablas `UIViewsByEntity` y `UIViewsByEntityType` respectivamente) y luego quitarla del registro (tabla `UIView`).

16.7.4.2 Asociar Vistas Existentes a un Tipo de Entidad

Para asociar una vista existente a un tipo de entidad sólo es necesario añadir registros en la tabla de "vistas por tipo de entidad" (`UIViewsByEntityType`). Dicho registro tendrá como clave principal los códigos de tipo de la entidad y de la vista que se desea vincular.

16.7.4.3 Asociar Vistas Existentes a una Entidad Concreta

Para asociar una vista existente a un tipo de entidad sólo es necesario añadir registros en la tabla de "vistas por entidad" (`UIViewsByEntity`). Dicho registro tendrá como clave principal los códigos de entidad y de la vista que se desea vincular.

16.8 La Interfaz de Usuario Web

La interfaz Web permite a los usuarios acceder a los contenidos del catálogo desde cualquier ordenador, sin necesidad de instalar ningún software cliente (sólo es necesario un navegador de Internet). La aplicación ha sido desarrollada siguiendo las pautas de diseño, usabilidad e interacción establecidas anteriormente en el **capítulo 14** de esta tesis.

En la figura 16.12 a continuación se muestran los principales elementos de navegación de la interfaz de usuario Web:

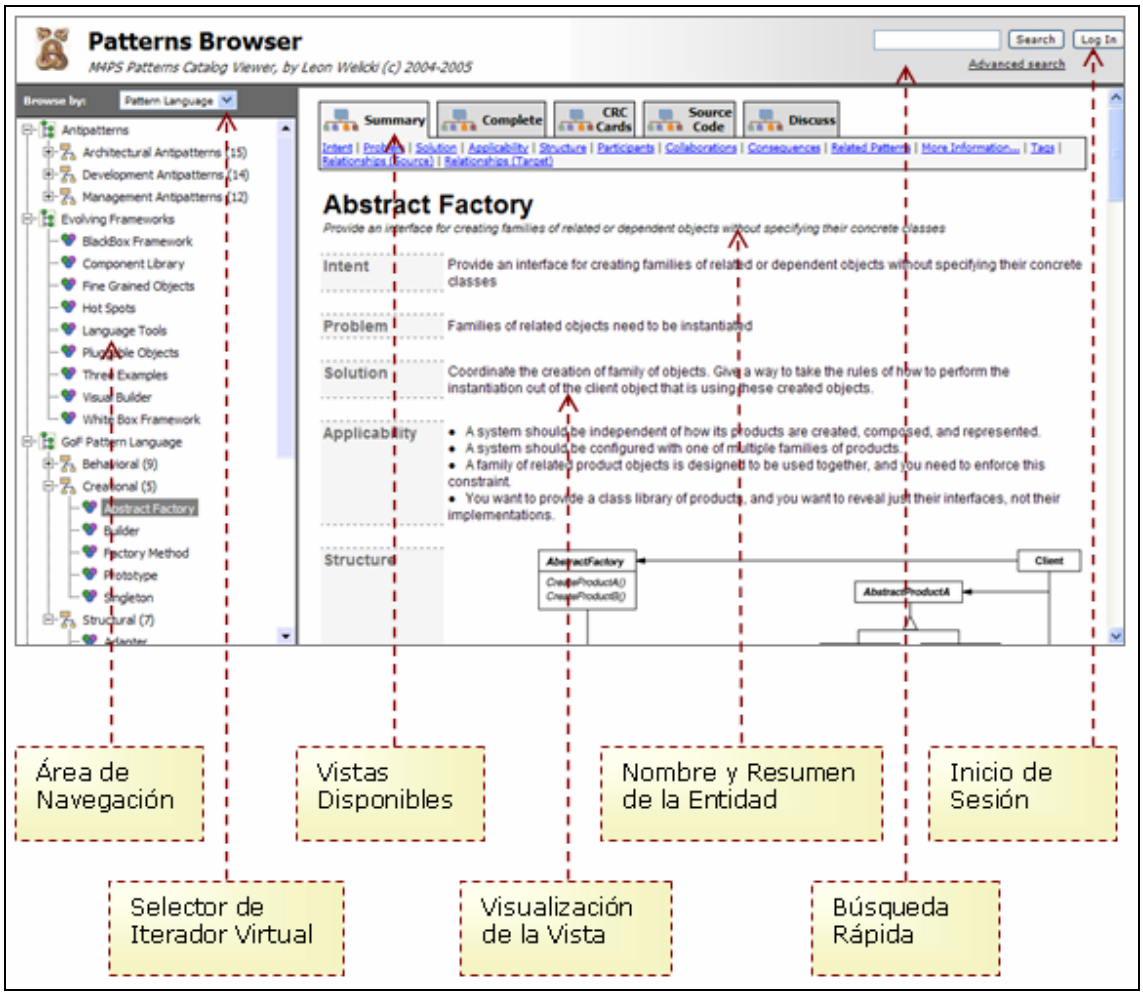


Figura 16.12 – Principales elementos de navegación de la interfaz Web del catálogo.

16.8.1 Vistas Dinámicas

Uno de los aspectos principales y más importantes de la herramienta de visualización del catálogo es su fuerte integración con el motor de vistas dinámicas. A partir de éste es posible vincular múltiples vistas a los distintos tipos de entidades (asociación general) o a entidades concretas (asociación específica), ofreciendo al usuario múltiples vistas sobre la misma información.

En las siguientes figuras se presenta un detalle de las vistas disponibles para distintas entidades. El objetivo de la inclusión de estas imágenes es mostrar en forma sencilla al lector cómo el motor de vistas puede ajustarse en función de la entidad que se está visualizando.

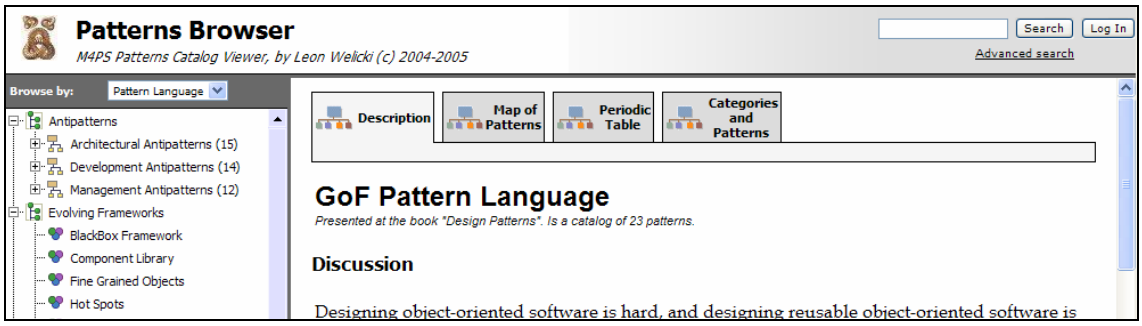


Figura 16.13 – Vistas aplicables al lenguaje de patrones GoF (asignación específica)

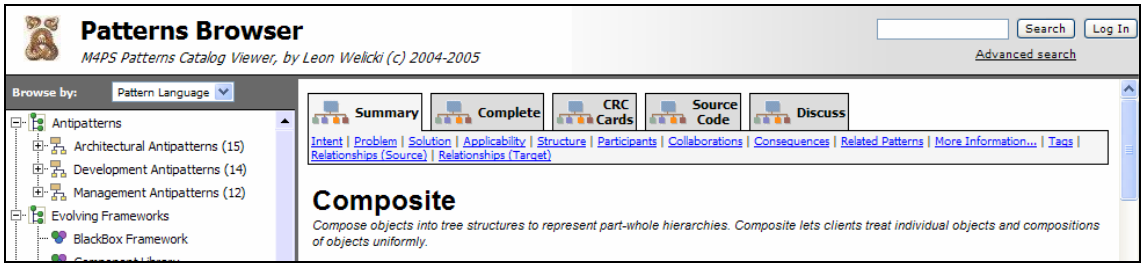


Figura 16.14 – Vistas aplicables al patrón Composite (asignación genérica). Las vistas de Tarjetas CRC y Código Fuente están presentes porque este patrón tiene especificada su estructura y comportamiento (en EML-SDL y EML-BDL respectivamente).

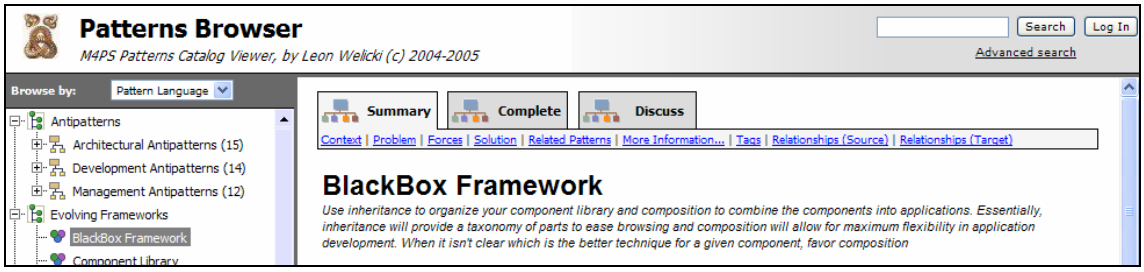


Figura 16.15 – Vistas aplicables al patrón Black Box Framework (asignación genérica). A diferencia del caso anterior, las vistas de Tarjetas CRC y Código Fuente no están presentes porque este no patrón tiene especificada su estructura y comportamiento.

El visor del catálogo incluye una amplia variedad de vistas que se presentan en más adelante en este capítulo en § 16.8.3 y § 16.8.4.

16.8.2 Categorización y Navegación del Catalogo de Patrones

El visor del catálogo ofrece dos modos de navegación por las entidades y varios modos de categorización. A continuación presentaremos los modos de navegación y las opciones de categorización disponibles en cada caso.

16.8.2.1 Navegación Mediante Iteradores Virtuales

En este caso la navegación se realiza mediante una estructura de árbol que se muestra a la izquierda de la pantalla, donde se agrupan los lenguajes de patrones con sus respectivas categorías y patrones, como se muestra en la figura a continuación.

El usuario puede elegir de una lista desplegable qué iterador virtual (§13.4.4, § 16.5) desea utilizar. Los iteradores que aparecen en la lista son los que están registrados en el fichero `Iterators.xml`. En la figura 16.16 a continuación, se muestra la lista de iteradores que se despliega a partir de los datos del fichero mostrado como ejemplo en § 16.5.3.

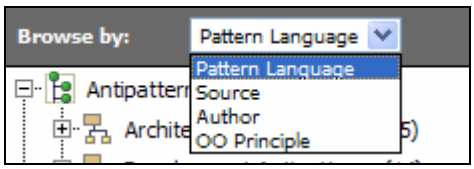


Figura 16.16 – Lista de selección de iteradores virtuales disponibles en el sistema.

La jerarquía de navegación (categorización) que se presenta al usuario varía en función del iterador seleccionado. En la figura 16.17 se muestran ejemplos de los resultados de la visualización de tres iteradores virtuales diferentes:

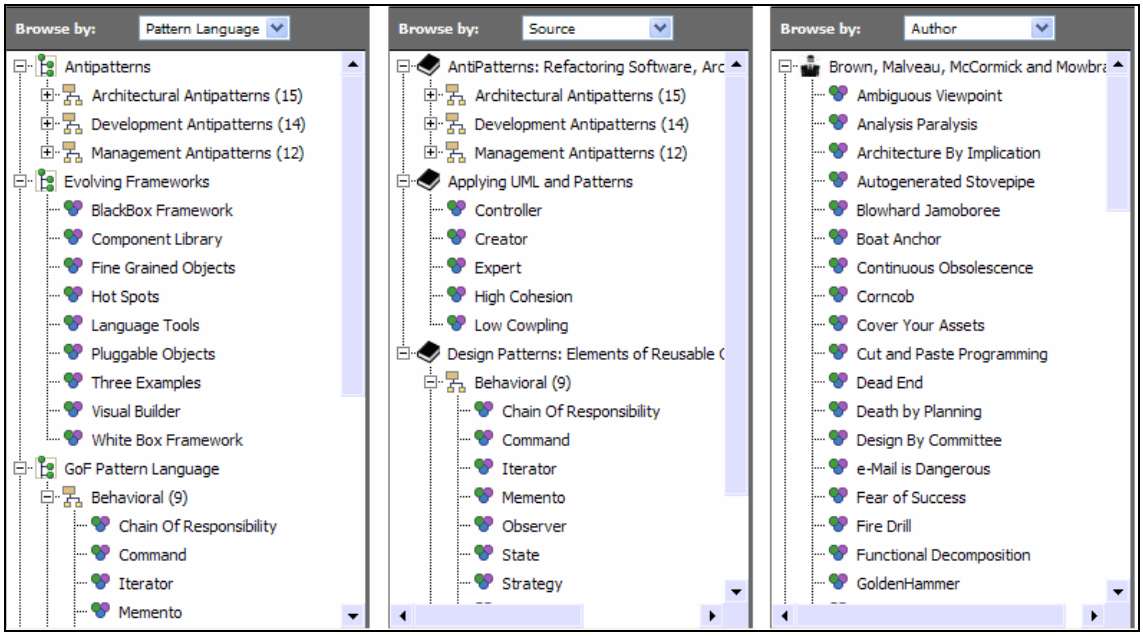


Figura 16.17 – Lista de selección de iteradores virtuales disponibles en el sistema.

El usuario puede seleccionar cualquiera entidad de las que se muestran en el árbol generado por la interpretación[249] del iterador virtual. Al seleccionar la entidad, el sistema la presenta en el área central junto con la lista de vistas disponibles (como se muestra en las figuras 16.13, 16.14 y 16.15).

16.8.2.2 Navegación por Descubrimiento (A Través de las Relaciones)

Es posible navegar por el catálogo a través de las relaciones entre entidades mediante el modelo de "navegación por descubrimiento" (§ 14.7.2). Al presentar una entidad el motor de vistas incluye la siguiente información general sobre ésta:

- Lista de Etiquetas

[249] Rendering (en inglés)

- Relaciones Salientes
- Relaciones Entrantes

En la siguiente figura se muestra cómo se presenta esta información para el patrón ABSTRACT FACTORY:

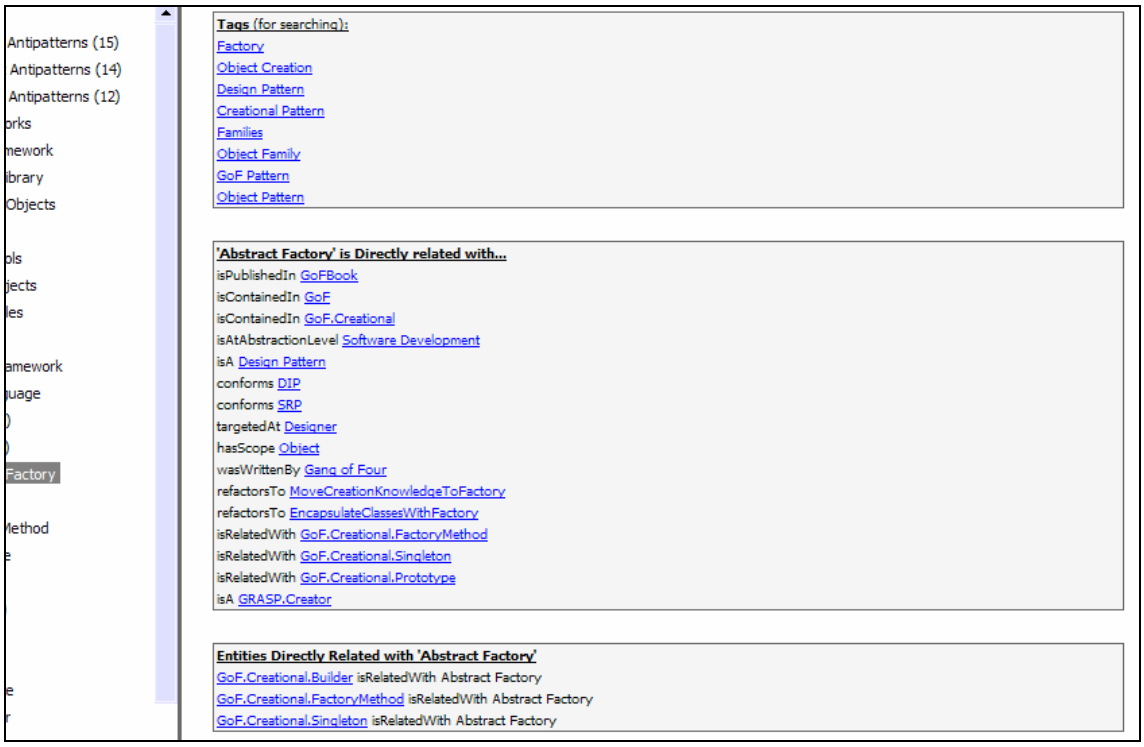


Figura 16.18 – Etiquetas, relaciones entrantes y relaciones salientes del patrón Abstract Factory.

Esta información nos permite ir navegando entre entidades, "*descubriendo*" y "*explorando*" las relaciones sin ninguna regla predeterminada. Este tipo de navegación la llamamos "*Navegación por Descubrimiento*" (§ 14.7.2).

En la figura 16.19 a continuación se presenta un escenario real de ejemplo de este modelo de navegación. Las vistas que se utilizan en este escenario se describen más adelante en este capítulo (§ 16.8.3 y § 16.8.4).

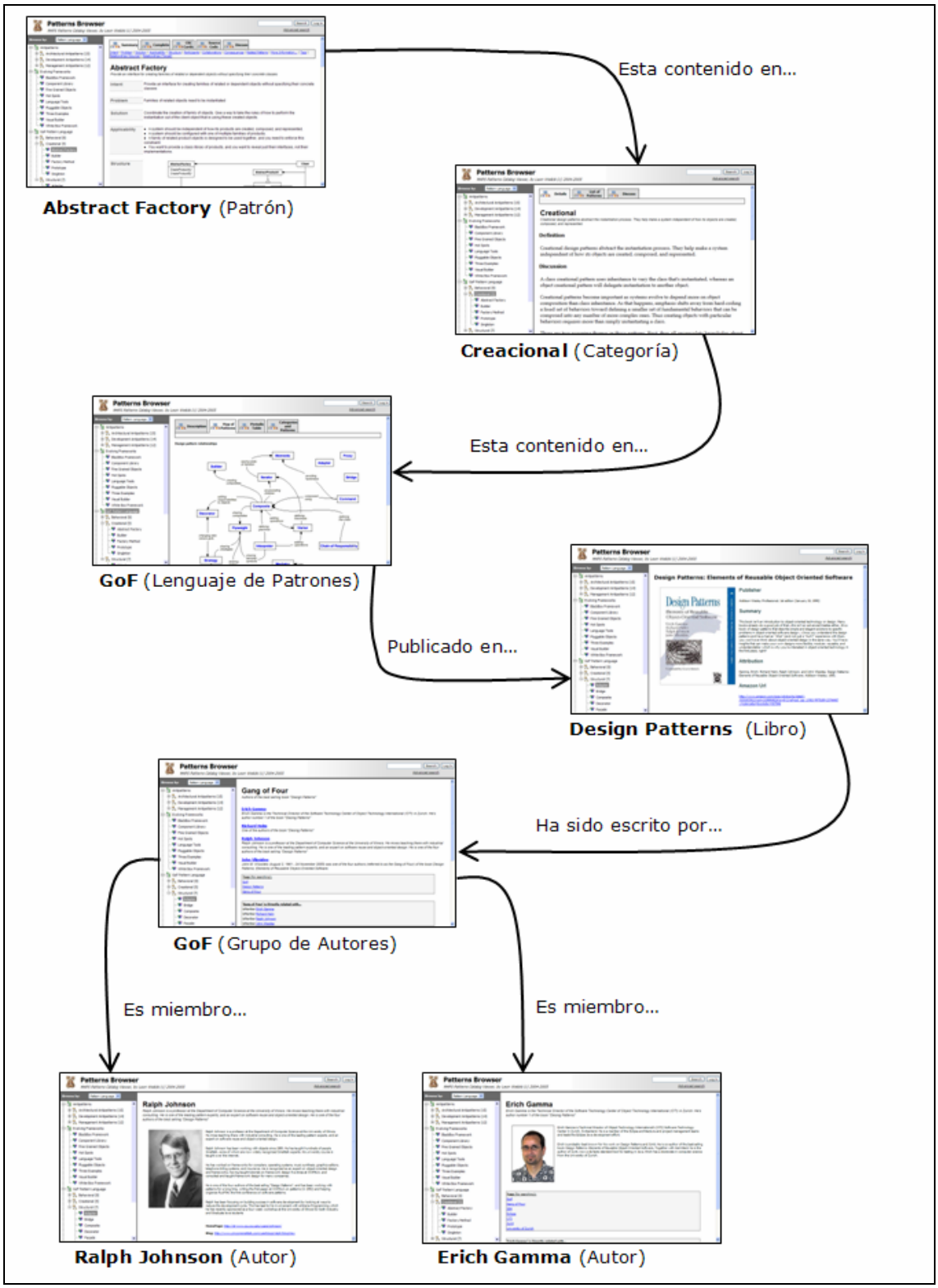


Figura 16.19 – Escenario de ejemplo real de navegación por descubrimiento.

16.8.3 Vistas Aplicables a los Patrones

En esta sección analizaremos las diferentes vistas desarrolladas en el prototipo que pueden aplicarse a un patrón. En cada caso explicaremos brevemente el propósito

de la vista e ilustraremos con figuras de ejemplo (que son capturas de pantallas de la interfaz Web del prototipo desarrollado).

16.8.3.1 Vista Completa

En esta vista, se muestra la plantilla completa del patrón (todas las propiedades de la plantilla, código fuente, participantes, etiquetas, relaciones, etc.). En la figura a continuación, se muestra un fragmento de esta vista aplicada al patrón ABSTRACT FACTORY [GoF95].

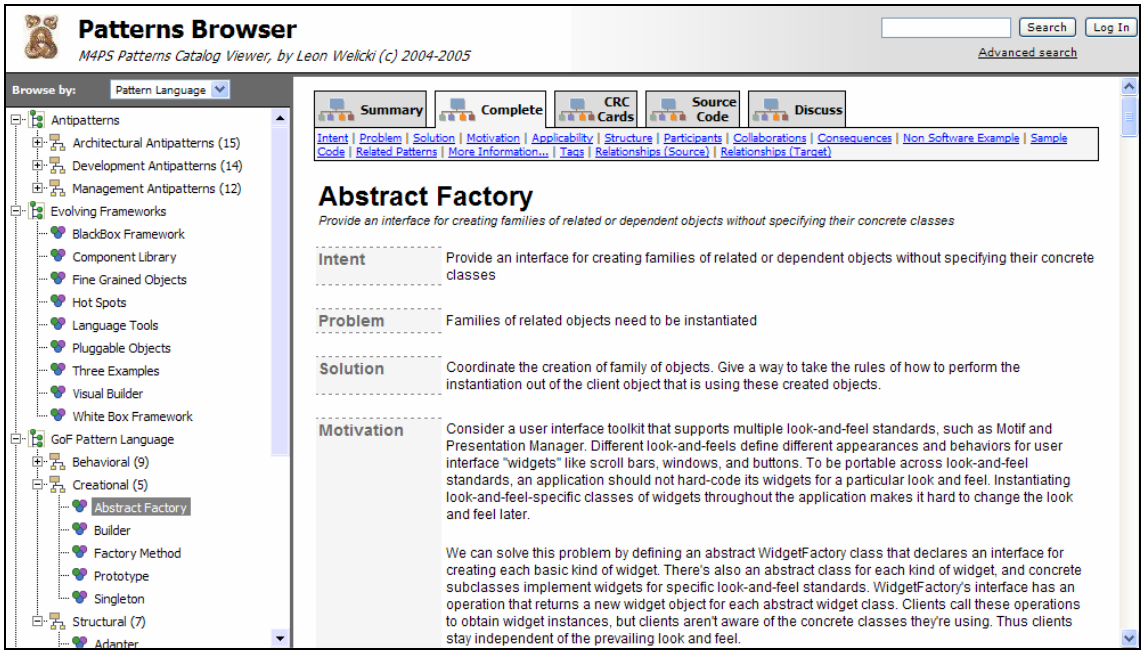


Figura 16.20 – Vista "completa" del patrón Abstract Factory.

En la vista se muestra también una sección con enlaces a sitios donde puede conseguirse más información sobre la entidad. En la figura 16.21 a continuación se muestran los enlaces de la sección "**Mas información**", donde puede obtenerse más información sobre el patrón que se está visualizando (agrupando conocimiento existente disperso en diversas fuentes en un sitio centralizado). En el caso de la figura siguiente, se trata nuevamente del patrón ABSTRACT FACTORY [GoF95]

Figura 16.21 – Visualización de la sección "Mas Información" del patrón Abstract Factory.

Finalmente, se presentan los patrones relacionados, de forma tal que se pueda navegar desde el patrón actual al relacionado. Cada relación se incluye con su respectiva explicación. En la figura a continuación se muestra la sección de "**Patrones Relacionados**" para el patrón COMPOSITE [GoF95]. Al hacer clic en cada uno de los links se accede a la entidad solicitada.

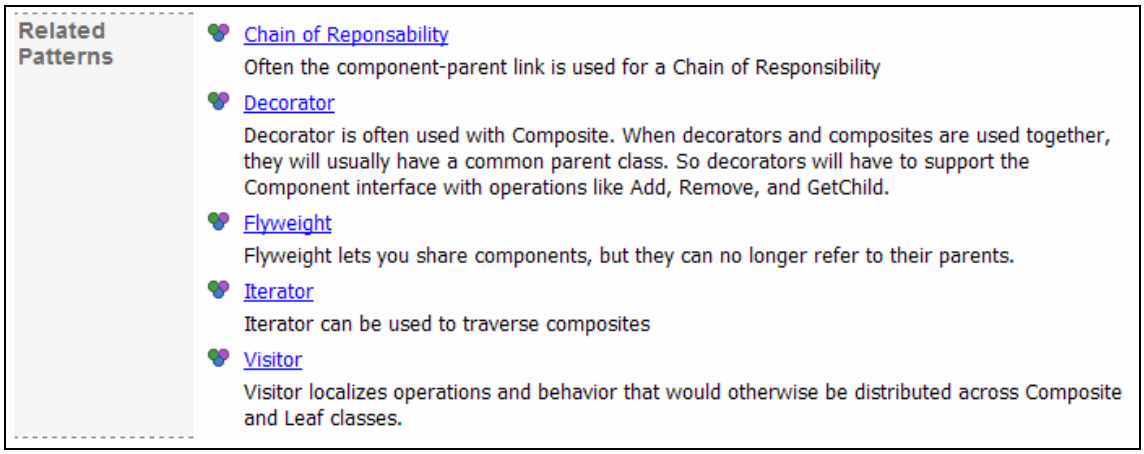


Figura 16.22 – Sección de patrones relacionados del patrón Composite.

16.8.3.2 Vista de Resumen

Esta vista es similar a la anterior, pero contiene menos información. Permite visualizar en forma más rápida la entidad. En la siguiente figura se muestra un ejemplo de la aplicación de esta vista al patrón ABSTRACT FACTORY.

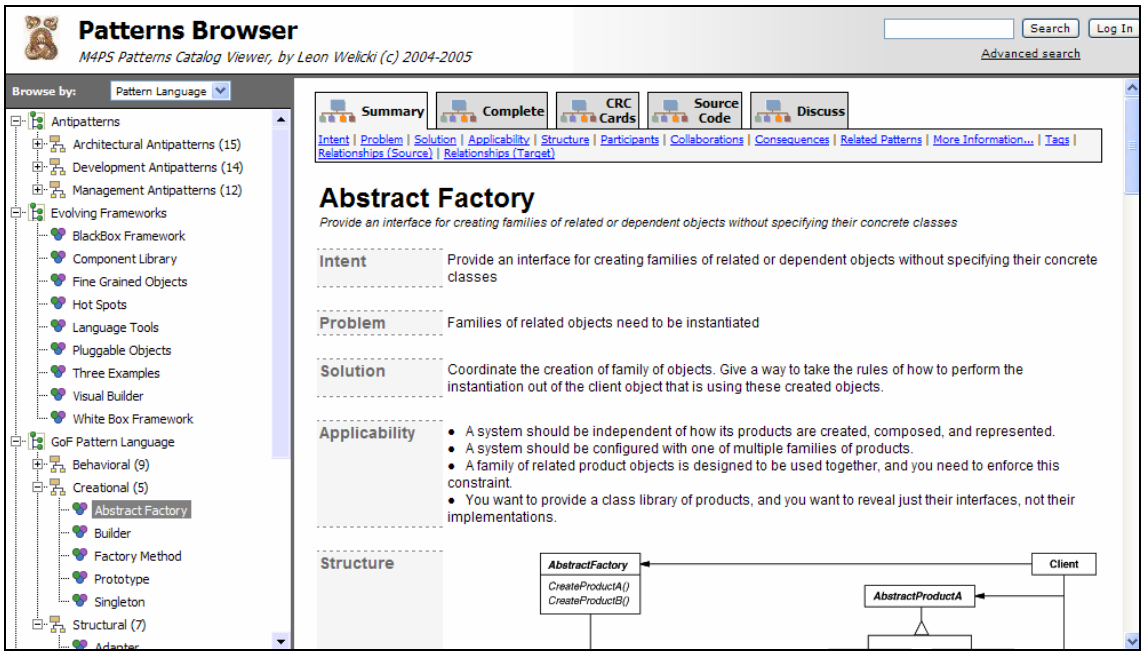


Figura 16.23 – Vista "resumen" del patrón Abstract Factory.

La relación entre la vista resumida y la vista completa es similar a la de los niveles de visualización presentada en el modelo de texto n-dimensional anteriormente en esta tesis (§ 14.5). La vista completa presenta la información a un "*nivel más alto*" (con más información) que la vista de resumen.

16.8.3.3 Vista de Tarjetas CRC

Para los patrones que tienen definidos sus participantes (con sus respectivas responsabilidades) se generan tarjetas CRC (Class-Responsibility-Collaboration). En la figura a continuación se muestran las **Tarjetas CRC** para el patrón TEMPLATE METHOD [GoF95].

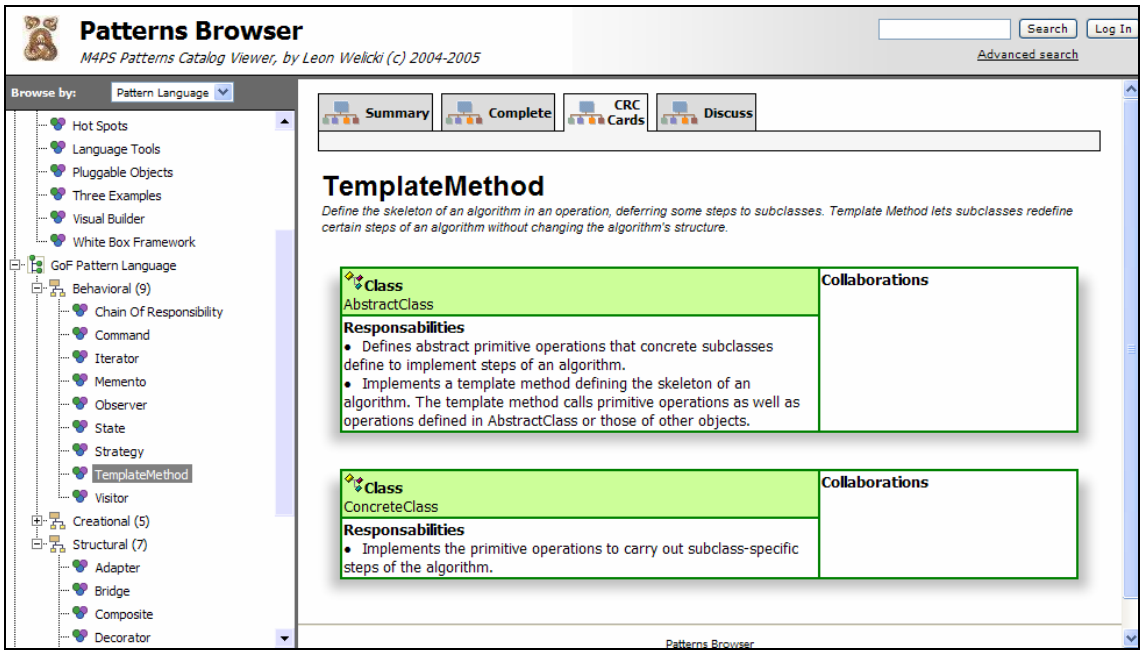


Figura 16.24 – Vista de "Tarjetas CRC" del patrón Template Method.

16.8.3.4 Vista de Código Fuente

Esta vista permite visualizar el código fuente que se genera (§ 13.4.10, § 16.6) a partir de la meta-especificación EML de una entidad. Es aplicable a las entidades que tienen especificado su nivel de implementación completo (utilizando EML-SDL y EML-BDL). En la figura a continuación se muestra un ejemplo de la aplicación de esta vista al patrón ABSTRACT FACTORY.

La vista permite al usuario ajustar el lenguaje destino y la implementación concreta a utilizar para generar el código. En la siguiente figura se muestra la interfaz de selección de lenguaje destino e implementación concreta a utilizar para generar el código.

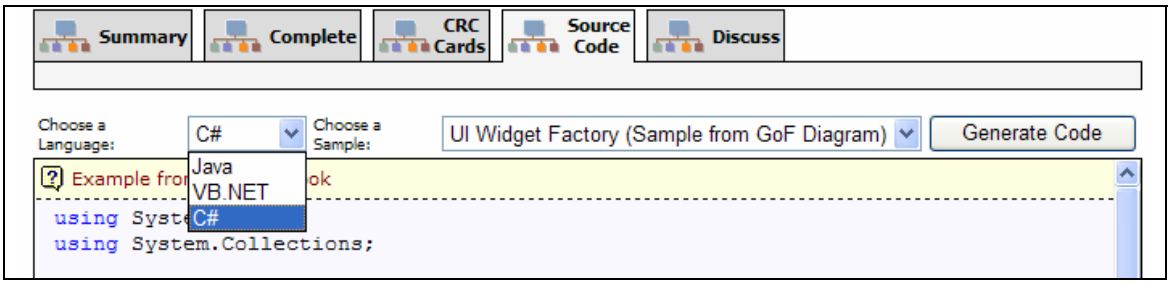


Figura 16.25 – Selección de opciones para la generación de código a partir de una entidad.

Una vez seleccionados el lenguaje y la implementación a utilizar se puede generar el código. En la siguiente figura se muestra el código generado para el patrón ABSTRACT FACTORY a partir de las opciones seleccionadas en la figura anterior.

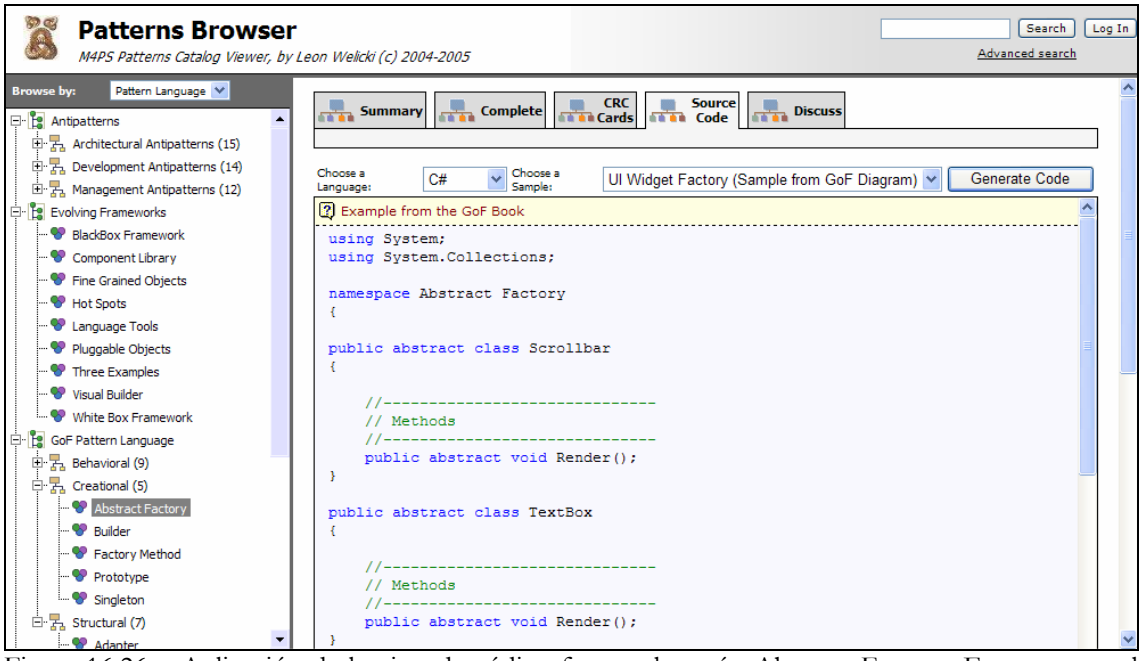


Figura 16.26 – Aplicación de la vista de código fuente al patrón Abstract Factory. En este caso el lenguaje destino es C# y la implementación se llama "UI Widget Factory"

Además del código fuente generado el sistema presenta un cuadro de texto con los resultados esperados al ejecutar dicho código (figura 16.27).

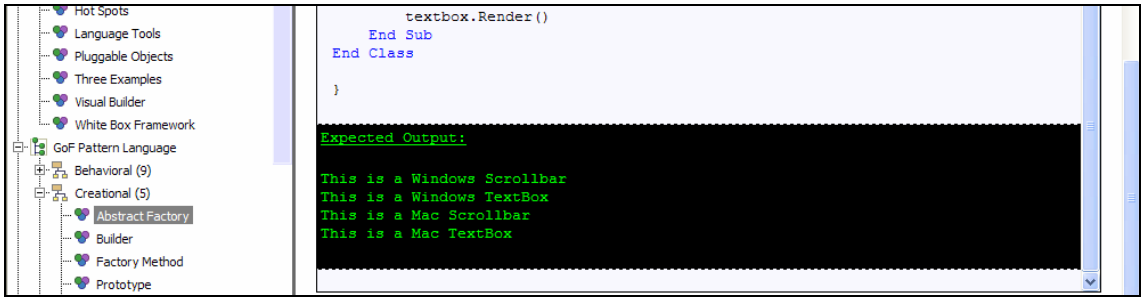


Figura 16.27 – Resultados esperados al ejecutar el código generado para el patrón Abstract Factory.

El usuario puede seleccionar en cualquier momento otro lenguaje destino u otra implementación concreta, como se muestra en la figura 16.28, a continuación.

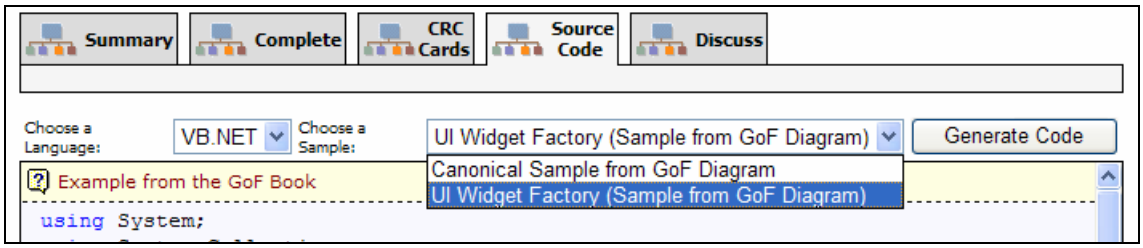


Figura 16.28 – Selección de otro lenguaje e implementación concreta.

Al generar el código con los nuevos parámetros seleccionados, el sistema actualiza el código fuente de la entidad. En la siguiente figura se muestra un bloque de código en VB.NET generado para el ejemplo canónico incluido en el libro "*Design Patterns*" para el patrón ABSTRACT FACTORY.

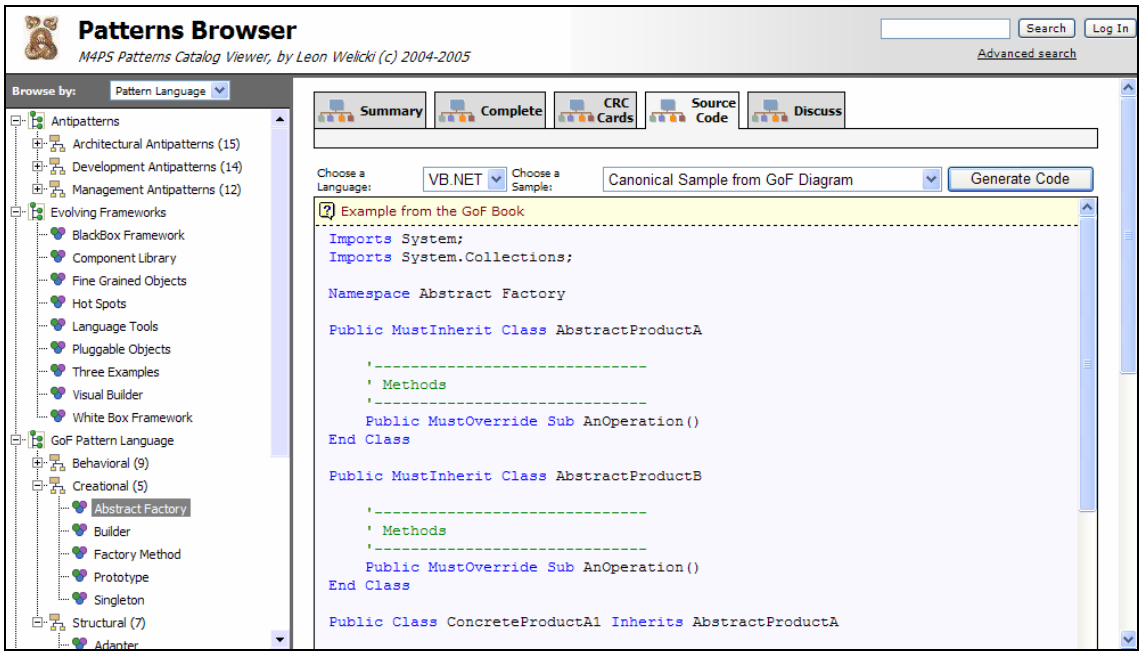


Figura 16.29 – Generación de código VB.NET del ejemplo canónico del patrón Abstract Factory a partir de la meta-especificación EML del patrón.

16.8.4 Vistas de Entidades

En esta sección analizaremos algunas de vistas desarrolladas en el prototipo que pueden aplicarse a distintas entidades. En cada caso explicaremos brevemente el propósito de la vista e ilustraremos con figuras de ejemplo (que son capturas de pantallas de la interfaz Web del prototipo desarrollado).

16.8.4.1 Vista Predeterminada

Existe una vista genérica que puede aplicarse a cualquier entidad. Esta vista toma todos los elementos del nivel de conocimiento y los presenta en forma ordenada en una interfaz Web. En la figura a continuación se muestra un ejemplo de esta vista aplicada a la categoría "Estructural" del lenguaje de patrones del libro "*Design Patterns*".

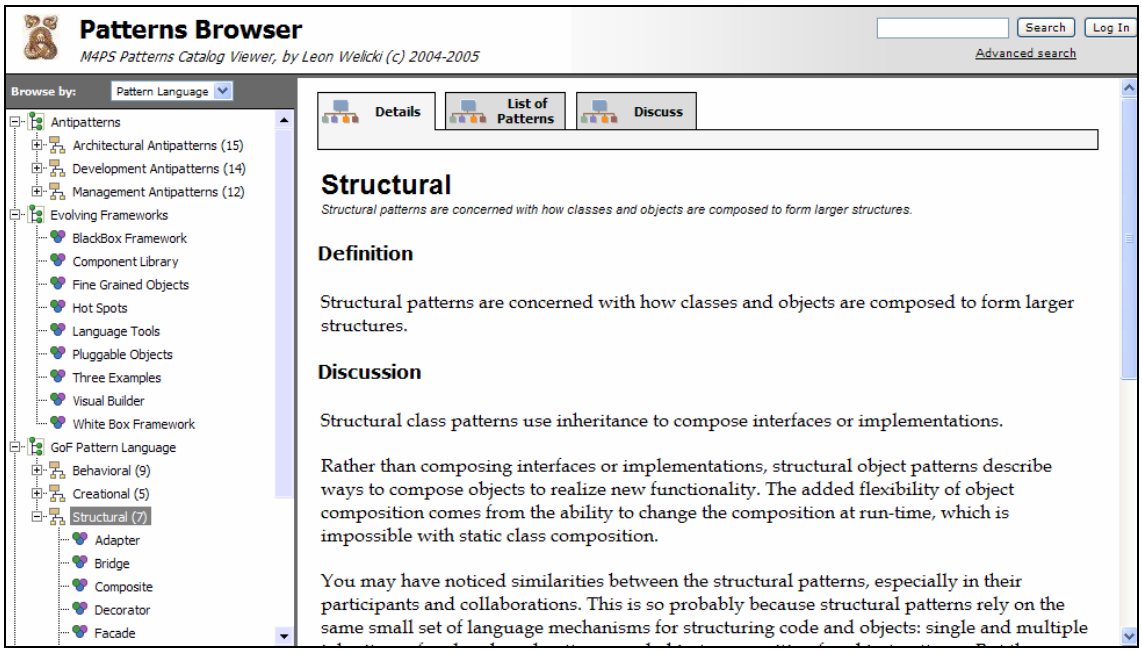


Figura 16.30 – Vista genérica aplicada a la categoría de patrones Estructurales del GoF.

16.8.4.2 Vista de Mapa de Patrones del GoF

En el libro "*Design Patterns*" [GoF95] se presenta un mapa de los patrones para asistir al usuario en el uso y navegación del catálogo. Hemos creado una vista que reproduce dicho mapa y la hemos asociado al lenguaje de patrones del GoF (utilizando el motor de vistas presentado anteriormente en este capítulo). En la siguiente figura se muestra el resultado de la aplicación de esta vista.

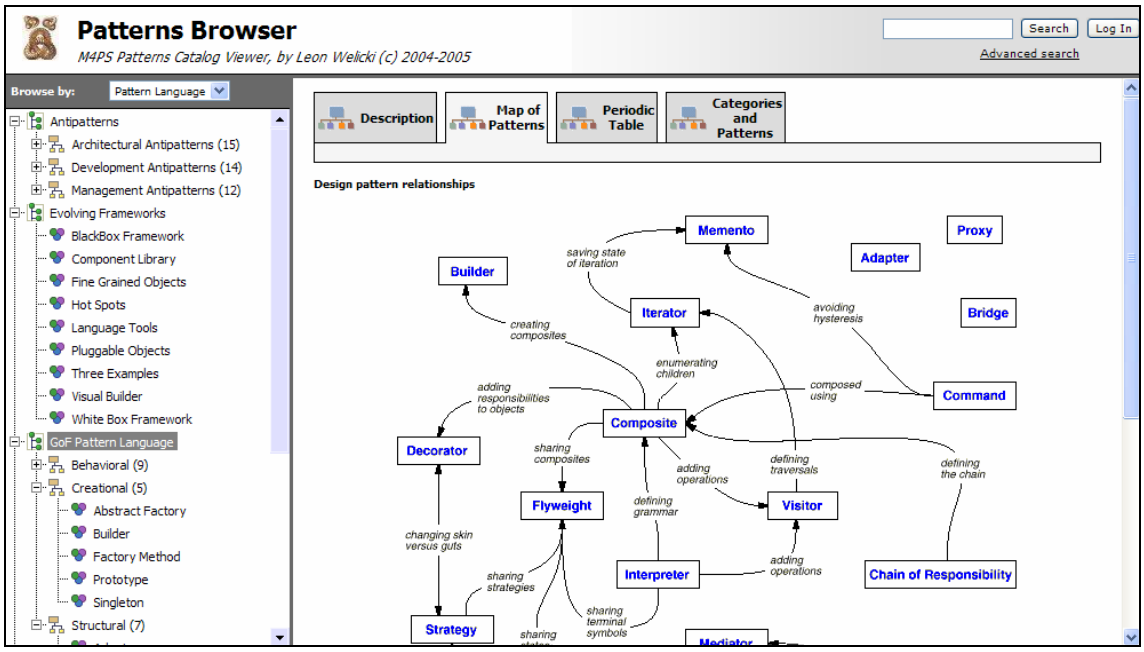


Figura 16.31 – Vista del mapa de patrones del libro "Patrones de Diseño"

16.8.4.3 Vista de Lista de Patrones "Incluidos en…"

La interfaz de visualización incluya una vista donde dada una entidad, se muestran todos los patrones que tienen una relación "*esta incluido en*" con dicha entidad[250]. En la siguiente figura se muestra esta vista aplicada a la categoría "Patrones Estructurales" del catálogo del GoF.

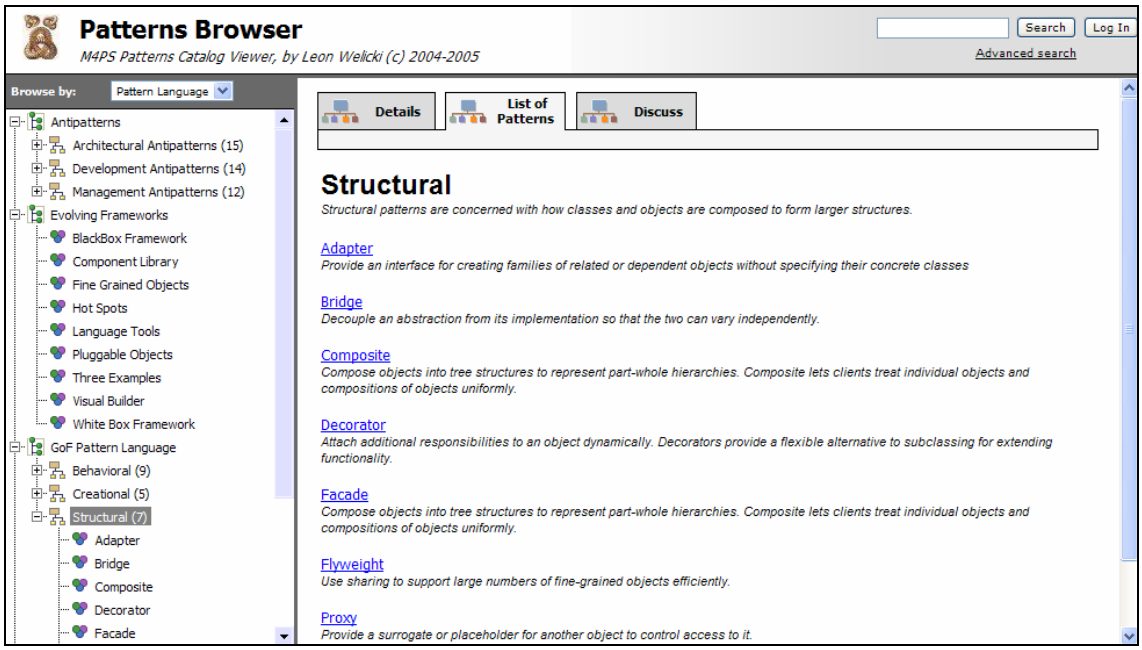


Figura 16.32 – Vista de "lista de patrones incluidos" aplicada a la categoría de patrones Estructurales.

[250] Las categorías y los lenguajes de patrones tienen este tipo de relación con los patrones que contienen.

16.8.4.4 Vista de Fuente de Información

La siguiente vista se utiliza para mostrar una fuente de información, ya sea un libro, una revista, un trabajo de investigación o un sitio Web.

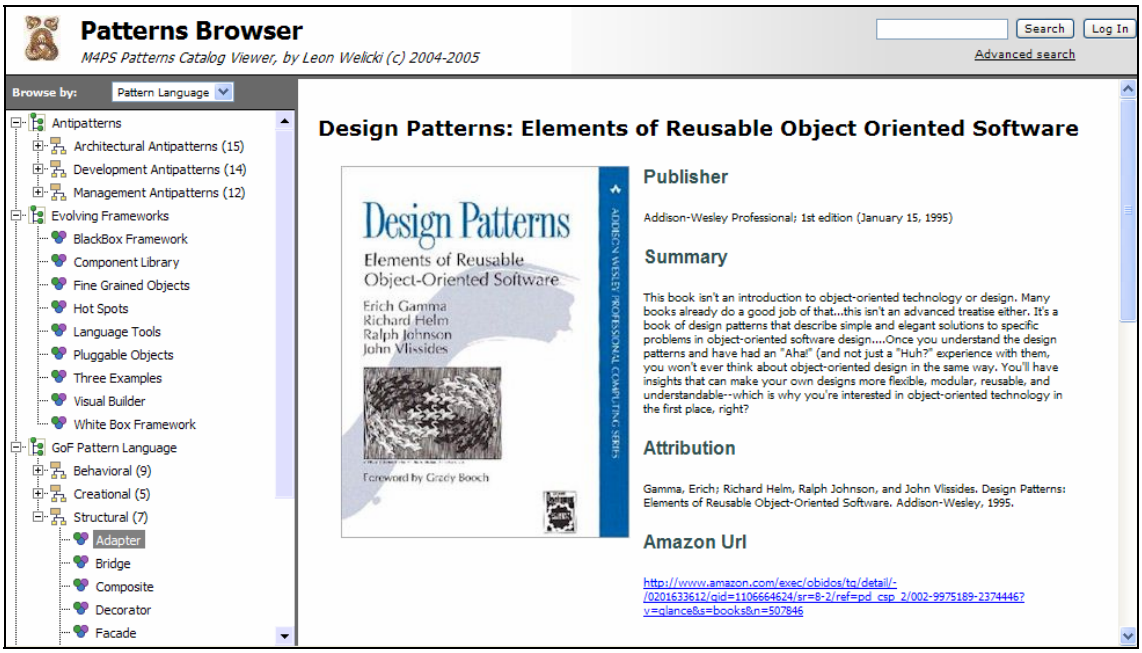


Figura 16.33 – Vista de fuente de información aplicada al libro "Patrones de Diseño"

16.8.4.5 Vista de Grupo de Autores

La siguiente vista se utiliza para mostrar un grupo de autores. Para cada uno de los autores que conforman el grupo se muestra su nombre y resumen breve. El nombre del autor es un enlace que conduce a la visualización de la meta-especificación EML con su información.

Figura 16.34 – Vista de grupo de autores aplicada al "Gang of Four"

16.8.4.6 Vista de Autor

La siguiente vista se utiliza para mostrar la información de un autor. En caso de tener una foto asociada, ésta se muestra a la izquierda de la vista. Si el autor tiene página Web o blog éstos se muestran como enlaces de hipertexto en la parte inferior de la pantalla.

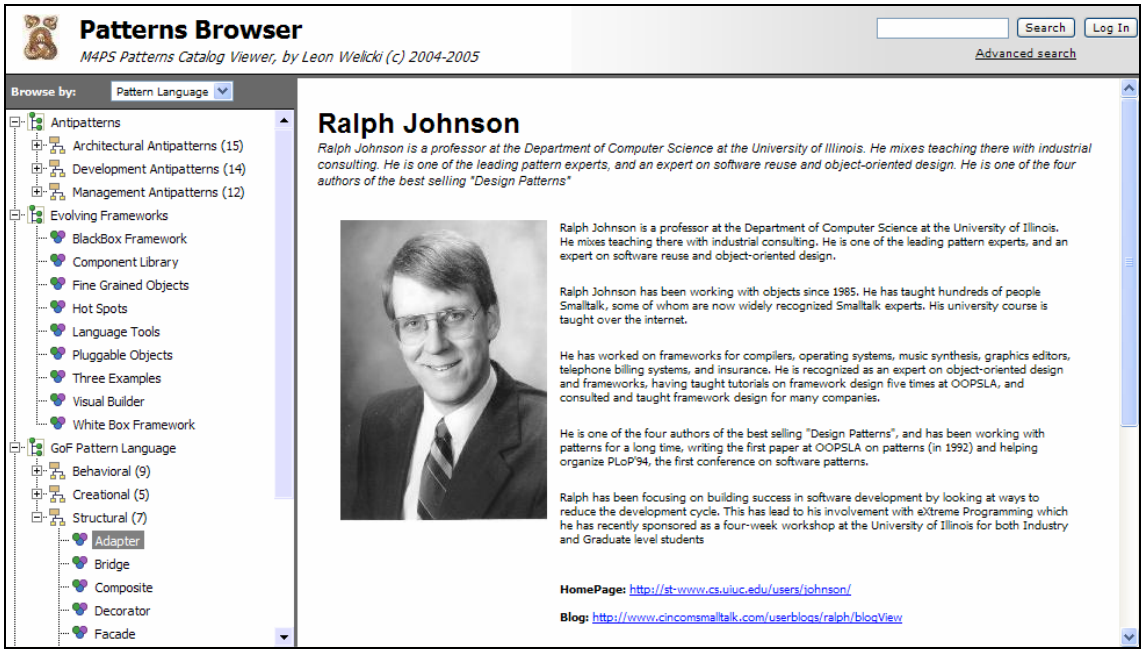


Figura 16.35 – Vista de autor aplicada a Ralph Johnson

16.9 Fachada de Servicios Web

La fachada de servicios Web expone los contenidos del catálogo utilizando SOAP sobre HTTP. El servicio es una implementación del patrón FAÇADE [GoF95], que expone funcionalidades de los módulos que hemos presentado durante este capítulo en forma sencilla.

El servicio Web, llamado `WSFacade` ha sido desarrollado utilizando ASP.NET, utilizando su infraestructura para servicios Web [MSAN]. La interfaz del servicio ha sido diseñada siguiendo los principios de las *Arquitecturas Orientadas a Servicios* (SOA).

16.9.1 Servicios Ofrecidos por WSFacade

En la versión actual, se exponen las siguientes funcionalidades:

- Obtener iteradores virtuales, para poder navegar por las referencias a entidades del catálogo.
- Obtener una entidad EML.
- Obtener el código fuente a partir de un patrón en un lenguaje concreto.

- Obtener la lista de lenguajes en que se puede obtener código fuente.
- Obtener la lista de iteradores virtuales disponibles.
- Búsqueda por coincidencia de texto o etiquetas.

16.9.2 Escenarios de Uso de la Interfaz de Servicios

La interfaz de servicios Web permite utilizar el catálogo desde aplicaciones heterogéneas sin importar la plataforma o lenguaje con que han sido construidas. A continuación, se enumeran brevemente algunos escenarios de uso de esta interfaz:

- **Integración con Entornos de Desarrollo**: utilización de los contenidos y funcionalidades del catálogo desde entornos de desarrollo como Eclipse [Eclipse] o Visual Studio .NET [VSNET].

- **Integración con Herramientas de Modelado**: utilización de los contenidos y funcionalidades del catálogo desde herramientas de modelado como Enterprise Architect [EA], Togheter [Together] o Rational XDE[Rational].

- **Cliente Pesado**: desarrollo de un cliente pesado con funcionalidades avanzadas de UI que permita utilizar los contenidos del catálogo en modo desconectado (off-line).

- **Repositorio Centralizado de Conocimiento**: utilización de los contenidos y funcionalidades del catálogo por aplicaciones heterogéneas como repositorio centralizado de conocimiento.

16.10 Describiendo Entidades con EML

En esta sección mostraremos en forma rápida y resumida cómo describir patrones y conceptos utilizando EML. Una descripción EML completa es muy larga. Por lo tanto, hemos seleccionado algunos ejemplos representativos para cada una de las secciones de la meta-especificación de entidades, para que el lector pueda tener una idea precisa de cómo es una meta-especificación EML.

16.10.1 Describiendo Patrones con EML

En esta sección mostraremos cómo se describe un patrón utilizando EML. Para facilitar la lectura de esta sección enumeraremos nuevamente las partes que componente a una meta-especificación EML (§ 12.2.3):

- **Identificación:** información general para identificar a la entidad (nombre de la entidad, tipo de entidad).

- **Etiquetas (Tags):** etiquetas para anotar a las entidades [Technoratti]. Decora a la entidad con metadatos descriptivos que pueden ser utilizados para búsquedas o categorización.

- **Relaciones**: relaciones entre la entidad que esta siendo descripta y otras entidad (por ejemplo "ABSTRACT FACTORY está contenido en el lenguaje de patrones del GoF", "FACTORY METHOD es un patrón de diseño", "ABSTRACT FACTORY contiene a FACTORY METHOD", etc.)

- **Resumen**: un resumen de la entidad que se está describiendo.

- **Plantilla (Template):** información literaria de la entidad. En el caso de los patrones, contiene la información de la plantilla [C2PatternForm] (para más detalles sobre las plantillas existentes, ver el **capítulo 4**).

- **Estructura**: Estructura de la entidad (participantes con sus relaciones y responsabilidades).

- **Implementación**: información sobre el comportamiento de la entidad (nivel de implementación). En esta sección tenemos dos partes bien diferenciadas: la implementación base, que es el modelo general de comportamiento y las implementaciones concretas, que son ejemplos específicos del comportamiento de la entidad[251].

En las siguientes sub-secciones mostraremos como describir con EML cada una de estas partes, utilizando como ejemplo fragmentos de meta-especificaciones de varios patrones.

16.10.1.1 Identificación de la Entidad

La identificación de la entidad es el primer paso necesario para meta-especificar un patrón. Estos datos se incluyen en el elemento raíz del documento EML (recordar que EML está basado en XML).

```
<entity
    type="pattern"
    id="GoF.Creational.AbstractFactory"
    name="Abstract Factory">
```

Código 16.4 – Primera línea de la definición de una entidad EML. En este caso es la primera línea de la definición del patrón Abstract Factory.

[251] Podría establecerse un paralelismo a modo de ejemplo entre la implementación base y una clase base abstracta y la implementación concreta y una clase concreta que implementa la clase base abstracta.

16.10.1.2 Añadiendo las Etiquetas

Los patrones se anotan (decoran) con etiquetas que pueden ser utilizadas para búsquedas o categorización. El siguiente fragmento es una vista parcial de las etiquetas utilizadas para anotar el patrón ABSTRACT FACTORY [GoF95]:

```
<tags>
  <tag>Factory</tag>
  <tag>Object Creation</tag>
  <tag>Families</tag>
  <tag>Object Family</tag>
  <tag>GoF Pattern</tag>
...
... Resto de las etiquetas omitidas para mayor claridad
...
</tags>
```

Código 16.5 – Sección "Tags" en la definición EML del patrón Abstract Factory.

16.10.1.3 Estableciendo Relaciones

En la sección `context` se establecen las relaciones entre la entidad que esta siendo descripta y otras entidades almacenadas en el catálogo. A continuación se muestra un extracto de las relaciones del patrón ABSTRACT FACTORY.

```
<context>
  <relationship
             targetType="Source"
             targetID="GoFBook"
             type="isPublishedIn"/>
  <relationship
             targetType="PatternLanguage"
             targetID="GoF"
             type="isContainedIn"/>
  <relationship
             targetType="Category"
             targetID="GoF.Creational"
             type="isContainedIn"/>
  <relationship
             targetType="PatternType"
             targetID="Design Pattern"
             type="isA"/>
  <relationship
             targetType="OOPrinciple"
             targetID="DIP"
             type="conforms"/>
...
... Rest of the context section elided for simplicity
...
</context>
```

Código 16.6 – Definición de las relaciones de una entidad.

16.10.1.4 Resumen

El resumen es una frase corta que sumariza a la entidad que esta siendo descripta. En el siguiente bloque se muestra el resumen del patrón ABSTRACT FACTORY.

```
<summary>Provide an interface for creating families of related or dependent
objects without specifying their concrete classes</summary>
```

Código 16.7 – Resumen de un patrón.

16.10.1.5 Describiendo la Plantilla

La plantilla contiene información literaria del patrón que está siendo descrito. En el bloque a continuación se muestra un extracto de la descripción de la plantilla del patrón ABSTRACT FACTORY.

```
<template>
  <intent>Provide an interface for creating families of related or dependent
objects without specifying their concrete classes</intent>
  <problem>Families of related objects need to be instantiated</problem>
  <solution>Coordinate the creation of family of objects. Give a way to take
the rules of how to perform the instantiation out of the client object that is
using these created objects.</solution>
  <applicability>
    <useWhen>A system should be independent of how its products are created,
composed, and represented.</useWhen>
    <useWhen>A system should be configured with one of multiple families of
products.</useWhen>
    <useWhen>A family of related product objects is designed to be used
together, and you need to enforce this constraint.</useWhen>
    <useWhen>You want to provide a class library of products, and you want to
reveal just their interfaces, not their implementations.</useWhen>
  </applicability>
...
... Rest of the template section elided for simplicity
...

</template>
```

Código 16.8 – Descripción de la plantilla de un patrón. La plantilla se compone de un conjunto de propiedades.

La plantilla se compone de propiedades (§ 12.2.4.1). Cada nodo en la definición EML de la plantilla es una propiedad (§ 12.2.4.1.2).

16.10.1.6 Describiendo la Estructura

La sección "`structure`" contiene información sobre la estructura del patrón. Se divide en tres partes:

- **Participants**: lista de los participantes del patrón.
- **Relationships**: relaciones entre participantes. Los tipos de relaciones s herencia, composición, agregación, asociación y creación.
- **Responsibilities**: definición textual de las responsabilidades del participantes.

El siguiente bloque muestra la descripción de la estructura del patrón ABSTRACT FACTORY:

```
<structure>
  <participants>
    <participant role="AbstractFactory" isAbstract="true"  cardinality="1"/>
    <participant role="ConcreteFactory" isAbstract="false" cardinality="+"/>
    <participant role="AbstractProduct" isAbstract="false" cardinality="+"/>
    <participant role="ConcreteProduct" isAbstract="false" cardinality="+"/>
    <participant role="Client"          isAbstract="false" cardinality="1"/>
  </participants>

  <relationships>
    <relationship
              type="inheritance"
              child="ConcreteFactory"
              parent="AbstractFactory"/>
    <relationship
              type="inheritance"
              child="ConcreteProduct"
              parent="AbstractProduct"/>
    <relationship
              type="creation"
              source="ConcreteFactory"
              target="ConcreteProduct"
              cardinality="1"/>
    <relationship
              type="association"
              source="Client"
              target="AbstractFactory"
              cardinality="1"/>
  </relationships>

  <responsibilities>
    <role name="AbstractFactory">
      <responsibility>Declares an interface for operations that create
abstract product objects.</responsibility>
    </role>
    <role name="ConcreteFactory">
      <responsibility>Declares an interface for operations that create
abstract product objects.</responsibility>
    </role>
    <role name="AbstractProduct">
      <responsibility>Declares an interface for operations that create
abstract product objects.</responsibility>
    </role>
    <role name="ConcreteProduct">
      <responsibility>Declares an interface for operations that create
abstract product objects.</responsibility>
      <responsibility>Implements the AbstractProduct interface.
</responsibility>
    </role>
    <role name="Client">
      <responsibility>Uses only interfaces declared by AbstractFactory and
AbstractProduct classes.</responsibility>
    </role>
  </responsibilities>
</structure>
```

Código 16.9 – Sección "Structure" en la definición EML del patrón Abstract Factory.

Esta descripción de la estructura simplifica la codificación de patrones en EML, permitiendo focalizarse en cada aspecto estructural en forma separada. El usuario primero identifica a los participantes, luego establece las relaciones entre éstos y finalmente añade las responsabilidades a cada uno.

16.10.1.7 Describiendo el Comportamiento

Como hemos mencionado anteriormente, EML contiene un DSL para representar el comportamiento de entidades a un alto nivel de abstracción (EML-BDL, § 12.2.4.5). Los patrones que tienen información de implementación deben tener una sección "`baseImplementation`" que describe cada participante junto con su comportamiento básico general. La descripción del comportamiento de un participante tiene tres secciones: `Properties`, `Constructors` y `Methods`.

En el siguiente bloque se muestra cómo se define la implementación base para el patrón Singleton [GoF95]:

```
<baseImplementation>
  <participant role="Singleton">
    <properties>
      <member scope="private,protected" name="instance"
                type="Singleton" isClass="true"/>
    </properties>
    <constructors>
      <constructor>
        <signature scope="private,protected"/>
      </constructor>
    </constructors>
    <behavior>
      <method name="getInstance">
        <signature scope="public" returns="Singleton"
                    isClass="true" isAbstract="false"/>
        <implementation>
          <if>
            <condition>
              <isNull variable="instance"/>
            </condition>
            <truePart>
              <createInstance variable="instance" type="Singleton"/>
            </truePart>
          </if>
          <return variable="instance"/>
        </implementation>
      </method>
    </behavior>
  </participant>
</baseImplementation>
```

Código 16.10 – Sección "baseImplementation" para la meta-especificacion EML del patrón Singleton. El ejemplo arriba representa la implementación canónica del Singleton en el libro "Design Patterns" [GoF95].

Esta implementación base puede ser redefinida, permitiendo a un patrón tener implementaciones concretas. Pueden añadirse implementaciones adicionales a un patrón utilizando la sección "`additionalImplementations`". En dicha sección se puede redefinir o aumentar cualquier participante definido en la sección "`baseImplementation`". Cada implementación particular se describe dentro de una sección "`implementation`".

En el siguiente bloque se muestra como añadir una implementación que utilice el idioma DOUBLE CHECK LOCK [POSA00] a nuestra meta-especificación EML del patrón SINGLETON:

```
<concreteImplementations>
  <implementation name="Double Check Lock">
    <implementationInfo>
      <friendlyName>Double Check Lock Singleton</friendlyName>
      <description>Implementation of the Singleton using the Double Check
Lock idiom</description>
    </implementationInfo>
    <instanceOf role="Singleton" name="SingletonMulti">
      <properties>
        <member scope="private" name="padlock"
                  type="object" isClass="true"/>
      </properties>
      <behavior>
        <method name="getInstance" is="getInstance">
          <signature scope="public" returns="SingletonMulti"
                       isClass="true" isAbstract="false"/>
          <implementation>
            <if>
              <condition>
                <isNull variable="instance"/>
              </condition>
              <truePart>
                <lock on="padlock">
                  <if>
                    <condition>
                      <isNull variable="instance"/>
                    </condition>
                    <truePart>
                      <createInstance variable="instance"
                                              type="SingletonMulti"/>
                    </truePart>
                  </if>
                </lock>
              </truePart>
            </if>
            <return variable="instance"/>
          </implementation>
        </method>
      </behavior>
    </instanceOf>
  </implementation>
</concreteImplementations>
```

Código 16.11 – Sección "additionalImplementation" para la meta-especificación EML del patrón Singleton. El ejemplo representa el Singleton combinado con el idioma Double Check Lock.

16.10.2 Otro Ejemplo: Describiendo un Concepto en EML

Como hemos mencionado anteriormente (**capítulo 12**), EML puede ser utilizado para describir conceptos de soporte a los patrones (lenguajes de patrones, categorías, libros, refactorizaciones, etc.). En el siguiente ejemplo se muestra la meta-especificación EML del libro "*Design Patterns*" [GoF95]

```
<entity
  type="source"
  id="GoFBook"
  name="Design Patterns: Elements of Reusable Object Oriented Software">

  <context>
    <relationship
        targetType="AuthorGroup" targetID="Gang of Four" type="wasWrittenBy"/>
  </context>

  <tags>
    <tag>GoF Book</tag>
    <tag>Book</tag>
    <tag>Gang of Four</tag>
    <tag>Design Patterns</tag>
  </tags>

  <summary><![CDATA[Gamma, Erich; Richard Helm, Ralph Johnson, and John
Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.]]></summary>

  <template>
    <type>Book</type>
    <title>Design Patterns: Elements of Reusable Object Oriented
Software</title>
    <yearPublished>1995</yearPublished>
    <isbn>0201633612</isbn>
    <publisher>Addison-Wesley Professional; 1st edition (January 15,
1995)</publisher>
    <attribution><![CDATA[Gamma, Erich; Richard Helm, Ralph Johnson, and John
Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.]]></attribution>
    <amazonUrl><![CDATA[http://www.amazon.com/exec/obidos/tg/detail/-
/0201633612/qid=1106664624/sr=8-2/ref=pd_csp_2/002-9975189-
2374446?v=glance&s=books&n=507846]]></amazonUrl>
    <summary>This book isn't an introduction to object-oriented technology or
design. Many books already do a good job of that...this isn't an advanced
treatise either. It's a book of design patterns that describe simple and
elegant solutions to specific problems in object-oriented software
design....Once you understand the design patterns and have had an "Aha!" (and
not just a "Huh?" experience with them, you won't ever think about object-
oriented design in the same way. You'll have insights that can make your own
designs more flexible, modular, reusable, and understandable--which is why
you're interested in object-oriented technology in the first place,
right?</summary>
    <images>
      <image name="picture">
        <description>Cover of the book</description>
        <url>/Books/GoF.jpg</url>
      </image>
    </images>
  </template>
</entity>
```

Código 16.12 – Descripción de una entidad. En este caso, corresponde a la descripción del libro "Patrones de Diseño" [GoF95]

Esta descripción contiene todos los elementos del nivel de conocimiento, pero ninguno del nivel de implementación. Otro aspecto que puede observarse es que el libro tiene una relación del tipo "*ha sido escrito por*" con sus autores.

16.10.3 Resumen: Estructura de Entidades en EML

En las secciones anteriores hemos visto como representar las secciones que conforman a un patrón y cómo representar un concepto (en este caso particular, un libro) utilizando EML.

Una entidad EML (ya sea un patrón o un concepto) tiene la siguiente estructura:

- Etiquetas *(opcional)*
- Contexto *(opcional)*
- Relaciones entre entidades
- Resumen
- Plantilla
    - Nivel literario del patrón
    - Problema
    - Solución
    - Motivación
    - Consecuencias
    - Etc.
- Estructura *(opcional)*
    - Participantes
    - Relaciones
    - Responsabilidades
- Implementación *(opcional)*
    - Implementación Base
    - Implementaciones Concretas
    - Implementaciones.

Capítulo 17
Evaluación del Prototipo

"Los viejos desconfían de la juventud
porque han sido jóvenes"
William Shakespeare

En esta sección evaluaremos el prototipo desarrollado en esta tesis (**capítulos 15** y **16**) en función de los requisitos establecidos para cada uno de sus componentes (§ 12.1, § 13.1 y § 14.1). Para llevar a cabo la valoración de un modo comparativo, evaluaremos en forma paralela un conjunto de sistemas reales (analizados previamente en el **capítulo 10**) adicionalmente al presentado.

Dada la amplitud y carácter multidisciplinar del modelo desarrollado hemos dividido la evaluación en tres secciones, a efectos de cubrir todas sus características:

- **Descripción de Patrones**: se evalúa a EML (**capítulo 12**), el lenguaje de meta-especificación para descripción de patrones y entidades creado en esta tesis respecto a otros enfoques existentes.

- **Catalogación**: se evalúa el Catálogo de entidades (**capítulo 13**) respecto a otros enfoques existentes. Dado la complejidad y amplitud de alcance del catálogo la evaluación se realiza en dos pasos: *evaluación contra enfoques generales* y *evaluación contra enfoques existentes*.

- **Visualización y Navegación**: se evalúa al visor del catálogo de patrones y entidades (**capítulo 14**) creado en esta tesis respecto a otros visores existentes.

Una vez cuantificadas y representadas las distintas evaluaciones, analizaremos los beneficios aportados por nuestro sistema en comparación con el resto de los estudiados.

❖ ❖ ❖

17.1 Evaluación del Lenguaje de Meta-Especificación

En esta sección evaluaremos a EML, el lenguaje de meta-especificación para descripción de patrones y entidades creado en esta tesis (**capítulo 12**) respecto a otros enfoques existentes. Los criterios de evaluación que utilizaremos son los requisitos establecidos anteriormente (§ 12.1). Los lenguajes que utilizaremos para realizar la comparación han sido analizados previamente en el **capítulo 10.**

A continuación incluimos una breve reseña de los lenguajes y enfoques con los que hemos comparado a EML:

- **PLML (Pattern Language Meta Language)**: este lenguaje ha sido creado por un conjunto de expertos con el objeto de describir patrones de UI. Es un lenguaje basado en XML. Su objetivo es traer claridad y homogeneidad en la descripción de patrones para poder compartir información (§ 10.1.3).

- **HTML (Hyper Text Markup Language)**: lenguaje de marcas basado en SGML utilizado en la Web para describir documentos de hipertexto. Si bien no es un lenguaje creado especialmente para describir patrones existe un gran número de catálogos públicos de patrones construidos utilizándolo (§ 10.1.2).

- **WIKI**: aplicación de informática colaborativa en un servidor que permite que los documentos allí alojados (las *páginas wiki*) sean escritos de forma colaborativa a través de un navegador, utilizando una notación sencilla para dar formato, crear enlaces, etc. Existen varios catálogos públicos de gran popularidad basados en esta tecnología (§ 10.1.1).

- **ODOL**: es una ontología de diseño orientada a objetos creada en el marco del proyecto *Web of Patterns* [WOP]. ODOL es uno de los pilares fundamentes de ese proyecto (§ 10.1.5).

- **XMI (XML Metadata Interchange)**: estándar del OMG para intercambio de modelos y metadatos en forma estandarizada (modelos cuyo meta-modelo puede expresarse con el MOF). Muchas herramientas de modelado utilizan XMI para describir e intercambiar patrones de software (§ 10.1.4).

17.1.1 Criterios Evaluados

A continuación, se enumeran los criterios a utilizar en la comparación (§ 12.1):

RL-1 **Alto nivel de abstracción**: la descripción de las entidades (patrones y conceptos de soporte) debe realizarse a un alto nivel de abstracción. Esta descripción debe ser independiente de cualquier pauta de utilización.

RL-2 **Soportar todas las plantillas de patrones**: poder describir patrones independientemente de la plantilla que utilicen (§ 4.1). Soportar todas las plantillas existentes de patrones y permitir crear nuevas dinámicamente en función de criterios arbitrarios (por ejemplo, una plantilla que contenga componentes de distintas plantillas[252]).

RL-3 **Descripción de relaciones:** debe tener la posibilidad de incluir relaciones entre entidades en forma sencilla. Las entidades relacionadas (los extremos de las relaciones) deben poder ser de diferentes tipos.

RL-4 **Descripción del comportamiento independiente de plataforma:** la descripción del comportamiento de las entidades involucradas en la solución debe ser totalmente independiente de los lenguajes y entornos de programación.

RL-5 **Anotación mediante etiquetas:** debe proveer la infraestructura necesaria para anotar a las entidades con etiquetas[253] [Tag] que pueden ser utilizadas para búsquedas y clasificación.

RL-6 **Describir entidades con y sin implementación:** representar en forma indistinta patrones que tengan los niveles de conocimiento e implementación (por ejemplo los de [GoF95] y [POSA96]) y los que sólo tengan el de conocimiento (por ejemplo los de [Evitts00] y [Fowler96]).

RL-7 **Reaprovechar el conocimiento existente:** debe tener la posibilidad de agregar apuntadores a sitios donde pueda conseguirse mas información sobre un patrón, aprovechando otras fuentes de conocimiento existentes.

RL-8 **Metadatos para búsquedas y clasificación:** Debe tener metadatos para búsqueda y clasificación. Estos metadatos deben poder describirse en forma independiente del contenido de la entidad, permitiendo que ésta pueda ser encontrada por un buscador semántico o que participe en esquemas de clasificación emergente (entre otros usos).

[252] En el prototipo (presentado en los capítulos 15 y 16) hemos creado una plantilla extendida para los patrones del GoF [GoF95] que incluye propiedades adicionales como ejemplos problema, solución y ejemplo en el mundo real (no software).
[253] Tags (en inglés)

RL-9 **Facilidad de producción:** debe ser fácil escribir meta-especificaciones de entidades con este lenguaje. Las meta-especificaciones deben poder crearse, editarse y modificarse con cualquier editor de textos[254].

RL-10 **Uniformidad:** utilizar un lenguaje uniforme para describir a los niveles de conocimiento e implementación[255].

Estos requisitos se estudian en mayor detalle en el **capítulo 12**.

17.1.2 Resultados de la Evaluación

En la tabla 17.1 a continuación se muestra el resultado de la evaluación realizada. En las filas se ubican cada uno de los requisitos y en las columnas cada uno de los entornos comparados. La última fila (que se muestra en color verde) corresponde a EML. En la figura 17.1 se muestran los resultados en forma gráfica.

	PLML	HTML	WIKI	ODOL	XMI	EML
RL-1	1	0	0,25	1	0,75	1
RL-2	0,5	1	1	0,25	0,25	1
RL-3	0,75	0,75	0,75	1	1	1
RL-4	0	0	0	1	1	1
RL-5	0	0,75	0	1	0,5	1
RL-6	1	1	1	0,5	0,5	1
RL-7	0,5	1	1	0,75	0,5	1
RL-8	0,75	0,5	0,5	1	0,5	1
RL-9	1	1	1	0	0,25	1
RL-10	0,5	1	1	0,75	0,75	1
	6	**7**	**6,5**	**7,25**	**6**	**10**

Tabla 17.1 - Evaluación del lenguaje de meta-especificación.

[254] En un caso extremo, deberían poder ser manipulados con el Bloc de Notas de Windows.

[255] En los enfoques existentes, el nivel de conocimiento se describe en modo textual, el código fuente en algún lenguaje de programación y la estructura en un lenguaje de modelado (como OMT o UML). Este requisito apunta a que toda esta información pueda ser expresada con un mismo lenguaje.

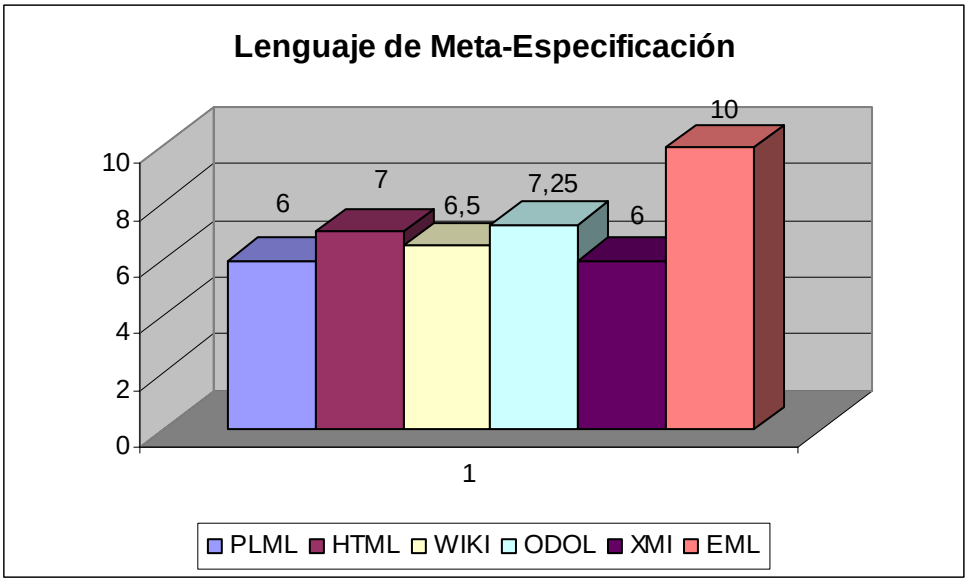


Figura 17.1 - Evaluación del lenguaje de meta-especificación.

17.1.3 Justificación de los Resultados

A continuación presentaremos la justificación de cada uno de los aspectos evaluados. Para cada criterio incluimos su código, título y una explicación en prosa de la evaluación realizada.

RL-1 **Alto nivel de abstracción**: EML, PLML y ODOL permiten describir a los patrones a un alto nivel de abstracción, independientemente de la intención de utilización final de éstos. Si bien XMI está a un alto nivel de abstracción, éste es más bajo que el de las anteriores, dado que su semántica esta orientada a un dominio de utilización específico. Los *wikis* permiten describir las entidades a un nivel muy bajo de abstracción, sin estructura y posibilidades de usos diversos. Los patrones descritos en HTML están orientados a una forma de visualización particular, siendo instancias concretas de documentos de hipertexto.

RL-2 **Soportar todas las plantillas de patrones:** EML permite representar cualquier plantilla de patrones (mediante la composición dinámica de propiedades primitivas) a un alto nivel de abstracción. HTML y WIKI permiten representar cualquier plantilla, pero a un nivel de abstracción muy bajo con alto acoplamiento con la presentación. PLML abarca un amplio número de plantillas, pero no permite describir a las que tengan secciones que no están incluidas en su DTD. ODOL y XMI tienen problemas para representar el nivel de conocimiento (y por consiguiente las plantillas), dado que están orientados principalmente a la descripción de la estructura y comportamiento.

RL-3 **Descripción de relaciones:** EML, XMI y ODOL permiten representar relaciones semánticas entre entidades. PLML, HTML y WIKI permiten representar relaciones sintácticas.

RL-4 **Descripción del comportamiento independiente de plataforma:** EML, ODOL y XMI permiten describir el nivel de implementación de los patrones a un alto nivel de abstracción con total independencia de la plataforma de ejecución y lenguaje de programación. Cuando se utiliza HTML o WIKI el nivel de implementación está orientado a una plataforma y lenguaje fijo, que es el que se describe en el documento que describe al patrón. Finalmente, esta característica no es soportada por PLML, dado que no permite describir el nivel de implementación.

RL-5 **Anotación mediante etiquetas:** EML provee soporte intrínseco para anotación de entidades con etiquetas. HTML permite anotar a las entidades utilizando etiquetas META [W3C99]. ODOL permite anotar a las entidades utilizando tecnologías de Web Semántica [BHL01] [DOS03]. El metamodelo de XMI puede ser ajustado para soportar etiquetas descriptivas. Los demás lenguajes no tienen facilidades para anotación.

RL-6 **Describir entidades con y sin implementación:** EML, PLML, HTML y WIKI permiten describir entidades que tienen los niveles de conocimiento e implementación o sólo el nivel de conocimiento. ODOL y XMI permiten describir una pequeña fracción del nivel de conocimiento (no permiten describirlo adecuadamente).

RL-7 **Reaprovechar el conocimiento existente:** EML, HTML y WIKI brindan la infraestructura para vincular información de fuentes heterogéneas y contextualizarla sin un propósito concreto (el propósito y semántica de utilización lo determina el usuario). ODOL permite vincularla a través de tecnologías de Web Semántica, pero no ofrece la flexibilidad de los anteriores para vincular cualquier tipo de información. XMI y PLML ofrecen soporte limitado para vincular información de fuentes externas.

RL-8 **Metadatos para búsquedas y clasificación:** EML y ODOL proveen una infraestructura de metadatos extensible a nivel de lenguaje que asisten a la clasificación y búsqueda de entidades. El resto de los lenguajes tienen mecanismos más precarios, basados en estructuras externas, anotaciones no estándar y relaciones sintácticas[256].

[256] En este aspecto PLML es el que más datos generales de clasificación aporta, especialmente en la sección `management`

RL-9 **Facilidad de producción:** EML, HTML, WIKI y PLML son fáciles de producir. Las especificaciones de los patrones en cualquiera de estos lenguajes pueden ser construidas con cualquier editor de textos sencillo (como el VI de Unix o el Bloc de Notas de Windows). ODOL y XMI son más complejas de producir y generalmente requieren el uso de una herramienta de edición especializada.

RL-10 **Uniformidad:** EML, HTML y WIKI permiten describir en forma uniforme los niveles de conocimiento e implementación. ODOL y XMI no brindan soporte completo para el nivel de conocimiento. Finalmente, PLML no permite describir el nivel de implementación en forma abstracta[257].

17.2 Evaluación del Catálogo

En esta sección evaluaremos el Catálogo de entidades descrito en el **capítulo 13** respecto a otros enfoques existentes. Dado la complejidad y amplitud de alcance del catálogo, la evaluación se realizará en dos pasos:

- **Evaluación contra enfoques generales**: en este caso, evaluaremos nuestra estrategia de catalogación respecto a otras posibles estrategias ampliamente utilizadas.

- **Evaluación contra implementaciones particulares**: en este caso, evaluaremos nuestra implementación comparándola con otras existentes.

Los criterios de evaluación que utilizaremos son los requisitos establecidos anteriormente en § 13.1. Los lenguajes y enfoques que utilizaremos para realizar la comparación han sido analizados previamente en el **capítulo 10**.

17.2.1 Criterios Evaluados

Los aspectos utilizados para evaluar el catálogo son los requisitos establecidos previamente en el **capítulo 13** de esta memoria. A continuación los incluimos nuevamente para asistir al lector en el estudio de la evaluación:

RC-1 **Soporte para almacenar entidades**: brindar la estructura necesaria para almacenar meta-especificaciones de patrones y conceptos de ingeniería del software descriptos con EML. Este requisito se refiere a la estructura física donde vivirán las especificaciones. La estructura lógica es atendida por los requisitos RC-2, RC-3, RC-4 y RC-7.

[257] El texto asociado al nivel de implementación (por ejemplo, todo el código fuente) se introduce entero en una sección llamada `implementation` (que según el DTD es de tipo ANY). Por lo tanto, hay una amplia brecha entre la descripción PLML y el código de implementación.

RC-2 **Soporte para almacenar relaciones:** almacenar relaciones entre las meta-especificaciones almacenadas en el catálogo[258]. Este requisito se relaciona directamente con RC-1 y RC-4, dado que las entidades a relacionar pueden estar almacenadas en el catálogo y por lo tanto deben haber sido previamente registradas. También se relaciona con RC-6, dado que una entidad puede relacionarse con otra que todavía no ha sido incluida en el catálogo[259] (el catálogo debe saber como gestionar este tipo de situaciones). Las relaciones deben poder establecerse ad-hoc en forma dinámica.

RC-3 **Analizador de entidades EML**: el analizador debe poder verificar la corrección sintáctica y semántica de las entidades EML. El analizador debe ser flexible, pudiendo adaptarse rápidamente a cambios en la especificación de EML.

RC-4 **Registro de entidades:** facilidades para registrar, modificar y desregistrar entidades en el catálogo. Antes de registrar una entidad, ésta debe ser analizada con el analizador (RC-3). Las entidades deben almacenarse en la estructura física creada a tal fin (RC-1).

RC-5 **Soporte para búsquedas:** debe proveer soporte para realizar búsquedas en los contenidos del catálogo, utilizando las definiciones de las entidades, los metadatos y las relaciones entre entidades.

RC-6 **Capacidad para manipular especificaciones incompletas:** no debe ser necesario disponer de toda la información sobre una entidad para introducirla en el catálogo (esta puede ser introducida a posteriori). Esto nos permite también crear la infraestructura para dar soporte al *gradiente semántico* (presentado en el **capítulo 12**, § 12.2.5).

RC-7 **Exponer las entidades mediante un API OO:** exponer las entidades mediante un API de objetos, para que puedan ser utilizadas por otros componentes. Estos objetos pueden ser compartidos utilizando una arquitectura de componentes "en-proceso"[260] o de componentes distribuidos (.NET Remoting [NETRemoting], CORBA [CORBA], DCOM [Microsoft], etc.).

RC-8 **Exponer las entidades mediante servicios Web:** exponer las entidades mediante una interfaz de servicios Web [W3C]. La granularidad de esta interfaz es diferente a la de RC-7[261].

[258] Las entidades pueden ser del mismo o diferente tipo.
[259] Esta característica es soportada por EML-RDL (§ 12.2.4)
[260] In-process (en inglés)
[261] En este caso, estamos ante una interfaz de grano grueso ("coarse-grained interface") mientras que en el otro caso, ante una interfaz de grano fino ("fine-grained interface").

RC-9 **Direccionamiento de entidades:** cada entidad debe tener una dirección que permita hacer referencia a ella en forma unívoca e inequívoca. Esta dirección será utilizada para recuperar la entidad del catálogo.

RC-10 **El catálogo debe ser un "espacio de información vivo":** el catálogo debe proveer mecanismos para mantener "vivo" ese espacio de información, a partir de fomentar y soportar la interacción entre sus usuarios y/o hacer "push" de información.

RC-11 **Generación de código:** generar código fuente a partir de las especificaciones EML[262]. El marco de generación debe ser extensible, permitiendo incluir nuevos generadores o modificar los existentes

17.2.2 Evaluación Contra Enfoques Generales

En esta sección hemos evaluado los aspectos detallados anteriormente contra 5 (cinco) tecnologías ampliamente utilizadas para crear catálogos de patrones. Los enfoques y tecnologías evaluados se estudian en detalle en el **capítulo 10** esta memoria. Es importante destacar que el criterio de evaluación para los catálogos basados en cada una de las tecnologías propuestas es general y no se corresponde con ninguna implementación particular sobre ese entorno. En la sección 17.x.3 se realiza una comparación utilizando implementaciones concretas.

A continuación incluimos una breve reseña de y enfoques con los que hemos comparado al catálogo:

- **Catálogos basados en PLML (Pattern Language Meta Language)**: catálogos de patrones basados en descripciones PLML.

- **Catálogos basados en HTML (Hyper Text Markup Language)**: catálogos de patrones y entidades basados en conjuntos de documentos HTML (como por ejemplo DoFactory [DoFactory], Enterprise Integration Patterns [EIPC], etc.).

- **Catálogos basados WIKI**: catálogos de patrones y entidades implementados sobre tecnología Wiki (como por ejemplo el Portland Pattern Repository [PPR] y PatternShare [Patternshare])

- **Catálogos basados en Ontologías**: catálogos de patrones y entidades implementados utilizando ontologías y tecnologías de Web Semántica (como por ejemplo el de Web of Patterns Project [WOP])

[262] Esto es aplicable a entidades EML que incluyen EML-SDL (§ 12.2.4.4) y EML-BDL (§ 12.2.4.5)

- **XMI (XML Metadata Interchange)**: catálogo basado en XMI (por ejemplo el que utiliza Enterprise Architect [EA])

	PLML	HTML	WIKI	ONTO	XMI	EML
RC-1	1	1	1	1	1	1
RC-2	0,75	0,75	0,75	1	1	1
RC-3	1	0	0	1	1	1
RC-4	1	0	1	1	1	1
RC-5	0	0	0,75	1	0	1
RC-6	0,75	1	1	0	0	1
RC-7	0	0	0	0,5	0,5	1
RC-8	0	0	0	0,5	0	1
RC-9	1	1	1	1	1	1
RC-10	0	0	1	0	0	1
RC-11	0	0	0	1	1	1
	5,5	**3,75**	**6,5**	**8**	**6,5**	**11**

Tabla 17.2 - Evaluación del catálogo

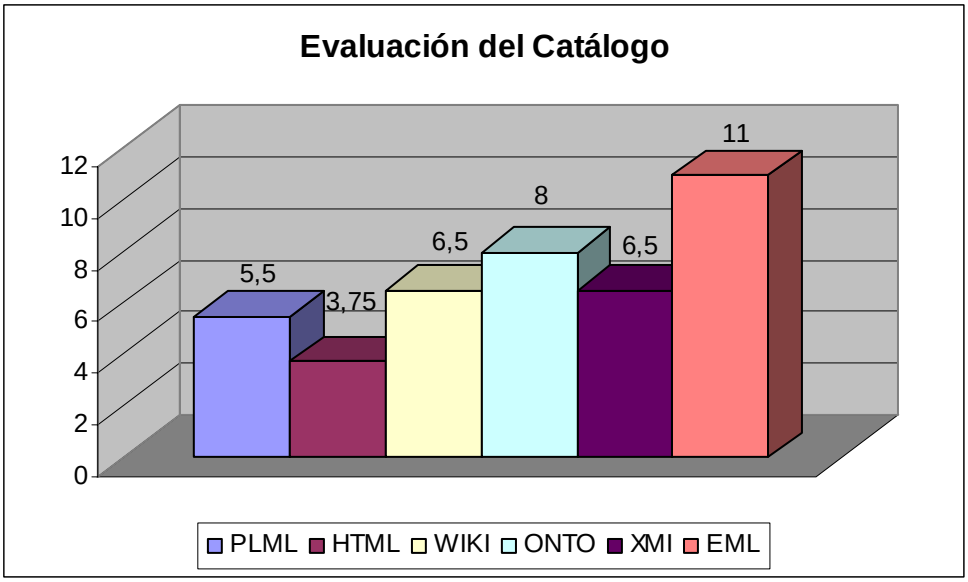


Figura 17.2 – Evaluación del catálogo

17.2.3 Justificación de los Resultados de la Evaluación Contra Enfoques Generales

RC-1 **Soporte para almacenar entidades**: todos los catálogos ofrecen la estructura necesaria para almacenar entidades.

RC-2 **Soporte para almacenar relaciones:** los catálogos basados en EML, XMI y Ontologías permiten describir relaciones semánticas entre entidades en el catálogo. En el resto de los casos sólo pueden describirse relaciones sintácticas.

RC-3 **Analizador de entidades**: los catálogos basados en EML, PLML, XMI y Ontologías incluyen analizadores que pueden verificar la corrección de una entidad que está incluida en el catálogo mediante la realización de análisis léxico y semántico.

RC-4 **Registro de entidades:** los catálogos basados en EML, PLML, WIKI, XMI y Ontologías ofrecen facilidades de registro de entidades. Los catálogo basados en HTML no incluyen está funcionalidad, dado que son un conjunto de páginas estáticas.

RC-5 **Soporte para búsquedas:** los catálogos basados en EML y Ontologías tienen soporte para la realización de búsquedas complejas. Los wikis permiten realizar búsquedas sencillas sobre sus contenidos.

RC-6 **Capacidad para manipular especificaciones incompletas:** los catálogos basados en EML, HTML y Wiki pueden manejar entidades con información incompleta de la misma forma que tratan entidades completas. PLML establece un conjunto de datos opcionales, con lo cual hay partes de la especificación que pueden estar incompletas.

RC-7 **Exponer las entidades mediante un API OO:** el catálogo EML expone sus contenidos mediante un API orientado a objetos.

RC-8 **Exponer las entidades mediante servicios Web:** sólo el catálogo EML expone en forma nativa sus contenidos a través de una fachada de servicios Web.

RC-9 **Direccionamiento de entidades:** todos los catálogos permiten establecer un sistema de direccionamiento que permite referirse en forma unívoca a una entidad en el catálogo.

RC-10 **El catálogo debe ser un "espacio de información vivo":** los catálogos EML y los Wikis proveen mecanismos de feedback intrínsecos que permiten a los usuarios interactuar entre ellos.

RC-11 **Generación de código:** es posible generar código (y otros artefactos de software) a partir de meta-especificaciones EML, ontologías y XMI.

17.2.4 Evaluación Contra Implementaciones Particulares

En esta sección hemos evaluado los aspectos detallados anteriormente contra 14 (catorce) implementaciones concretas de catálogos públicos de patrones. Estas implementaciones incluyen catálogos Web, proyectos de investigación y herramientas de modelado. Las herramientas evaluadas se estudian anteriormente en el **capítulo 10** esta memoria.

A continuación incluimos una breve reseña los catálogos de patrones con los que hemos comparado al nuestro (al cual nos referiremos como **PB**):

- **PPR - Portland Pattern Repository**: El Portland Pattern Repository [PPR] ha sido el primer repositorio de patrones en la Web y también el primer wiki (creado por Ward Cunnigham). Contiene una amplia variedad de contenido.

- **PS - PatternShare**: iniciativa de Microsoft [Patternshare] con el objeto de agrupar patrones de distintos autores y lenguajes de patrones. Esta creado sobre una plataforma de wiki, lo cual permite la edición colaborativa.

- **SJB - Sun's Core J2EE Patterns**: Este catálogo [CJ2P] contiene una versión electrónica de extractos de los contenidos del libro "*Core J2EE Patterns*" [ACM01]. Es un conjunto de páginas HTML que pueden ser navegadas en forma sencilla por el usuario.

- **MDL - MOUDIL (Montreal Online Usability DIgital Library)**: es un sistema para compartir patrones de interfaz de usuario [MOUDIL]. Ha sido creado con dos objetivos: 1) Ofrecer acceso a los desarrolladores a un amplio conjunto de patrones de UI y 2) Actuar como un foro donde los investigadores puedan explorar colecciones de patrones para descubrir más sobre la naturaleza de esas colecciones

- **PA - Patterns Almanac**: Patterns Almanac [Rising00] ha sido una iniciativa para crear un directorio (con referencias a las fuentes originales) de los patrones existentes en un momento determinado. Lamentablemente luego de su lanzamiento no ha sido actualizado[263]. La versión Web del "*Patterns Almanac*" [PAW] está desarrollada utilizando un conjunto de páginas HTML con la información de cada patrón.

- **PEAA - Patterns of Enterprise Application Architecture**: El catálogo de patrones de arquitectura de aplicaciones empresariales [EAAC] es un complemento al libro "*Patterns of Enterprise Application Architecture*" [Fowler02]. Contiene versiones resumidas de los patrones incluidos en esa obra. Los patrones están descritos utilizando HTML y relacionados mediante hipervínculos.

- **EIP - Enterprise Integration Patterns**: El catálogo de patrones de integración de aplicaciones empresariales [EIPC] es un complemento al libro "*Enterprise Integration Patterns*" [HW03]. Contiene versiones resumidas de los patrones incluidos en esa obra. Los patrones están descritos utilizando HTML y relacionados mediante hipervínculos.

[263] La última fecha de actualización es Noviembre de 2004

- **PHSA - Patterns (Handbook of Software Architecture)**: La sección de patrones [PHSA] del "*Handbook of Software Architecture*" [Booch] agrupa diversos patrones de varias obras. Es una iniciativa liderada por Grady Booch similar a la que ha llevado a cabo varios años atrás Linda Rising. Contiene más de 450 patrones de distintos libros, conferencias y trabajos de investigación.

- **PID - Patterns of Interaction Design**: conjunto comprehensivo de patrones de interacción [PID]. Esta implementando utilizando XML y páginas HTML.

- **UIP - Tidwell's UI Patterns**: colección de patrones de UI [Tidwell] de una de las autoras más influyentes en la comunidad de patrones de UI (Jennifer Tidwell). Contiene versiones resumidas de los patrones del libro "*Designing Interfaces*" [Tidwell05].

- **DF - DoFactory Patterns Page**: Esta página [DoFactory] es muy popular en el mundo de desarrollo en ambientes .NET. Contiene las implementaciones y descripciones estructurales de los 23 patrones del GoF. La página pertenece a una empresa dedicada a consultoría y formación en desarrollo, especializada en temas relacionados con ingeniería del software. Es un conjunto de páginas HTML a las cuales se accede mediante hipervínculos.

- **BT - Borland Together**: herramienta de modelado UML de Borland [Together]. Los patrones se especifican utilizando una combinación de ficheros XML, XSD y HTML.

- **EA - Enterprise Architect**: herramienta de modelado UML de *Sparx Systems* [EA]. Los patrones se especifican en un fichero XML, utilizando XMI.

La descripción de todas estas herramientas ha sido incluida previamente en el **capítulo 10** de esta tesis.

	PPR	PS	SJB	MDL	PA	PEAA	EIP	PHSA	PID	UIP	DF	WOP	BT	EA	PB
RC-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RC-2	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1
RC-3	0.75	0.75	0	0	0	0	0	0	0	0	0	1	1	1	1
RC-4	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1
RC-5	1	1	0	1	1	0	0	1	0	0	0	1	1	1	1
RC-6	1	1	0	0.75	0.75	0.75	0.75	1	1	1	1	0.5	0	0	1
RC-7	0	0	0	0	0	0	0	0	0	0	0	0.75	0.75	0.75	1
RC-8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
RC-9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RC-10	1	1	0	1	0	0	1	1	0	0	0	0.5	0.5	0	1
RC-11	0	0	0	0	0	0	0	0	0	0	0.25	1	1	1	1
	7.75	**6.75**	**3**	**6.75**	**4.75**	**2.75**	**4.75**	**7**	**4**	**4**	**3.25**	**8.75**	**8.25**	**7.75**	**11**

PPR Portland Pattern Repository [PPR]
PS PatternShare [Patternshare]
SJB Sun J2EE Blueprints [CJ2P]
MDL MOUDIL [MOUDIL]
PA Patterns Almanac [PAW]
PEAA Patterns of Enterprise Application Architecture [EAAC]
EIP Enterprise Integration Patterns [EIPC]
PHSA Patterns en el Handbook of Software Architecture [PHSA]
PID Patterns in Interaction Design [PID]
UIP Tidwell's UI Patterns [Tidwell]
DF DoFactory [DoFactory]
WOP Web of Patterns Project [WOP]
BT Borland Together [Together]
EA Enterprise Architect [EA]
PB Patterns Browser [Welicki06]

Tabla 17.3 - Evaluación del catálogo.

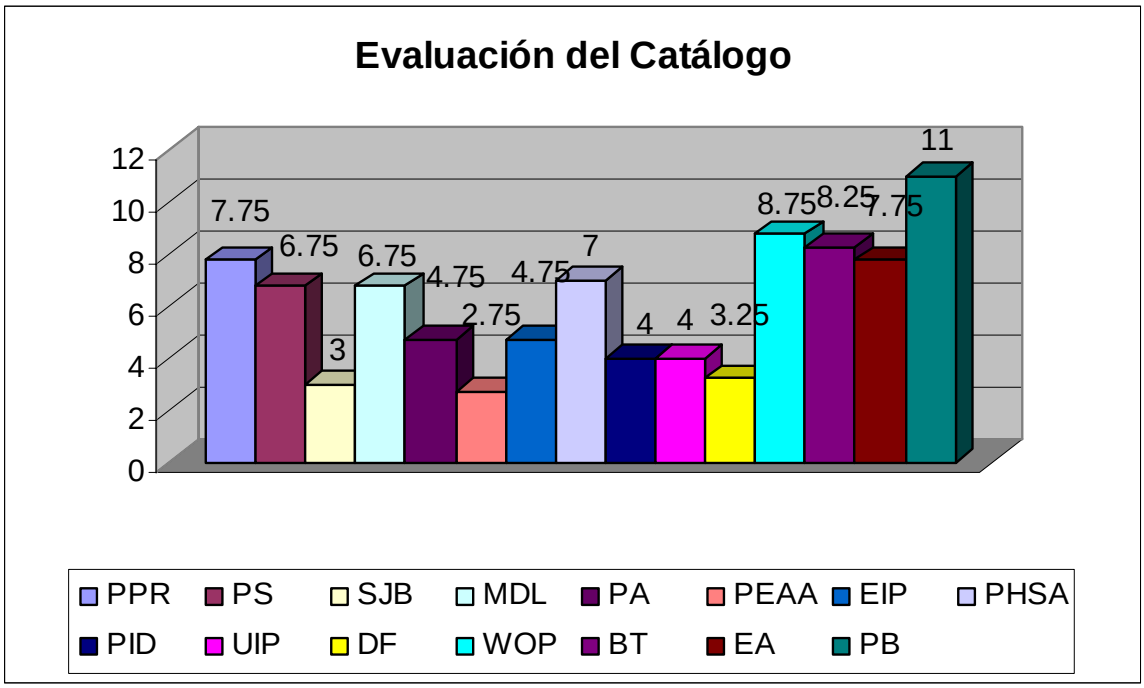


Figura 17.3 – Evaluación del catálogo.

17.2.5 Justificación de los Resultados de la Evaluación Contra Implementaciones Particulares

RC-1 **Soporte para almacenar entidades**: todos los catálogos brindan soporte para almacenar entidades

RC-2 **Soporte para almacenar relaciones:** todos los catálogos contienen algún tipo de relación entre sus elementos, salvo DF y PEAA.

RC-3 **Analizador de entidades**: sólo el catálogo EML, Web of Patterns, Together y Enterprise Architect incluyen un analizador de entidades.

RC-4 **Registro de entidades:** PB, WOP, BT y EA permiten introducir nuevas entidades fácilmente en el sistema. PPR y PS permiten a los usuarios crear nuevas páginas WIKIs, lo cual puede ser visto en forma bastante similar a la introducción de nuevas entidades. El resto sólo permite introducir entidades a sus administradores o creadores.

RC-5 **Soporte para búsquedas:** PB, PPR, PS, MDL, PHSA, WOP, BT y EA ofrecen soporte para búsquedas.

RC-6 **Capacidad para manipular especificaciones incompletas:** PB tiene capacidad de manejar entidades incompletas. Curiosamente, esta característica la comparte con los catálogos que manejan información con menor grado de estructuración y semántica (catálogos HTML y WIKI, por ejemplo). Los catálogos que gestionan información más estructurada cumplen este requisito en menor medida (en caso de cumplirlo).

RC-7 **Exponer las entidades mediante un API OO:** PB expone sus contenidos mediante un API de objetos. WOP, BT y EA exponen sus contenidos en esta forma, aunque es necesario un pequeño esfuerzo de traducción.

RC-8 **Exponer las entidades mediante servicios Web:** sólo PB expone sus contenidos a través de servicios Web.

RC-9 **Direccionamiento de entidades:** todos los catálogos tienen alguna forma de direccionamiento para referirse a las entidades que contienen.

RC-10 **El catálogo debe ser un "espacio de información vivo":** PB, PRR, PS, MDL y EIP provee mecanismos para dar soporte a la creación de comunidad y alguna forma de interacción entre los consumidores del catálogo. WOP y BT ofrecen un soporte más limitado, orientado a un subconjunto más pequeño de personas (usuarios familiarizados con la ontología y las tecnologías en el primer caso y usuarios de la herramienta en el segundo).

RC-11 **Generación de código:** PB, WOP, BT y EA permiten generar código a partir de definiciones abstractas de los patrones. DF contiene varios ejemplos de código que el usuario puede utilizar, pero ya éstos son fijos y están codificados en HTML.

17.3 Herramientas de Visualización del Catálogo

En esta sección evaluaremos al visor del catálogo de patrones y entidades creado en esta tesis respecto a otros enfoques existentes. Los criterios de evaluación que utilizaremos son los requisitos establecidos anteriormente en la sección 14.1. Los catálogos públicos que utilizaremos para realizar la comparación han sido analizados previamente en el **capítulo 10** y están incluidos en los enumerados previamente en la sección 17.3.4 de este capítulo.

17.3.1 Criterios Evaluados

Los aspectos utilizados para evaluar el catálogo son los requisitos establecidos previamente en el **capítulo 14** de esta memoria. A continuación los incluimos nuevamente para asistir al lector en el estudio de la evaluación:

RV-1 **Multiplataforma:** Poder ser utilizado en plataformas heterogéneas.

RV-2 **Accesible a través de la Web:** Estar disponible a través de la Web. De esta manera, no existe la limitación geográfica y de distribuciones.

RV-3 **Usabilidad:** el catálogo debe ser fácil de utilizar.

RV-4 **Múltiples vistas de la información:** Exponer múltiples vistas de los patrones del catálogo. Para cada patrón, por ejemplo, mostrar las vistas de conocimiento, de código fuente, etc.

RV-5 **Navegabilidad:** Permitir navegar en forma sencilla y eficiente por los patrones del catálogo. Poder explotar las relaciones entre los patrones.

RV-6 **Facilidades de búsqueda:** poder buscar patrones a través de distintas técnicas de búsqueda, como por ejemplo pattern-matching, inferencia a través de los meta-datos, etc.

RV-7 **Comunidad:** ofrecer infraestructuras a los usuarios para generar comunidad para promover la evolución del conocimiento almacenado y generación de nuevo conocimiento a partir del existente.

RV-8 **Personalización de resultados:** Los niveles de conocimiento e implementación deben poder ser ajustados por el usuario: los patrones son un punto de partida. Por ejemplo, ajustando detalles de implementación se puede llegar a soluciones diferentes a partir de un mismo patrón. Ajustando el nivel de conocimiento se puede expresar mejor una idea en un contexto determinado.

17.4.2 Resultados de la Evaluación

	PPR	PS	SJB	MDL	PA	PEAA	EIP	PHSA	PID	UIP	DF	BT	EA	PB
RV-1	1	1	1	1	1	1	1	1	1	1	1	0.5	0	1
RV-2	1	1	1	1	1	1	1	1	1	1	1	0	0	1
RV-3	0.5	0.5	1	0.75	0.75	1	1	1	1	1	1	0.5	0.5	1
RV-4	0	0	0	0	0	0	0	0	0	0	0	0.75	0.75	1
RV-5	0.5	0.5	1	0.75	0.75	1	1	0.75	1	1	1	0.75	0.75	1
RV-6	1	1	0	1	1	0	0	1	0	0	0	0.5	0.5	1
RV-7	1	1	0	0.75	0	0	0.75	0.5	0	0	0	0	0	1
RV-8	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	5	**5**	**4**	**5.25**	**4.5**	**4**	**4.75**	**5.25**	**4**	**4**	**4**	**4**	**3.5**	**8**

PPR	Portland Pattern Repository [PPR]
PS	PatternShare [Patternshare]
SJB	Sun J2EE Blueprints [CJ2P]
MDL	MOUDIL [MOUDIL]
PA	Patterns Almanac [PAW]
PEAA	Patterns of Enterprise Application Architecture [EAAC]
EIP	Enterprise Integration Patterns [EIPC]
PHSA	Patterns en el Handbook of Software Architecture [PHSA]
PID	Patterns in Interaction Design [PID]
UIP	Tidwell's UI Patterns [Tidwell]
DF	DoFactory [DoFactory]
BT	Borland Together [Together]
EA	Enterprise Architect [EA]
PB	Patterns Browser [Welicki06]

Tabla 17.4 – Evaluación del Visor del Catálogo.

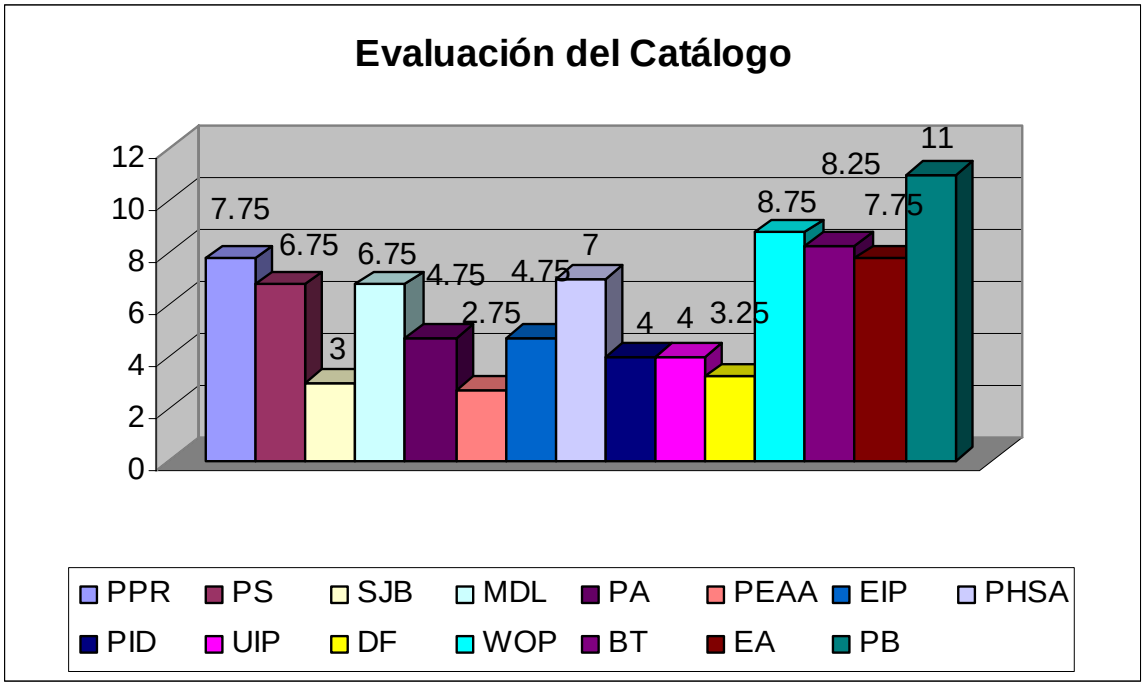


Figura 17.4 – Evaluación del Visor del Catálogo.

17.3.3 Justificación de los Resultados

RV-1 **Multiplataforma:** Todos los visores evaluados excepto BT y EA pueden ser utilizadas desde cualquier plataforma, dado que son accesibles a través de la Web. BT permite acceder desde varias plataformas a través de su cliente Java.

RV-2 **Accesible a través de la Web:** Todos los visores evaluados excepto BT y EA son accesibles desde la Web.

RV-3 **Usabilidad:** si bien este criterio es difícil de medir, hemos utilizado las heurísticas recomendadas por Nielsen para realizar la evaluación de usabilidad. PB cumple con todas las heurísticas.

RV-4 **Múltiples vistas de la información:** PB ofrece varias vistas sobre la información, permitiendo al usuario elegir de qué modo desea consumir las entidades. BT y EA permiten tener varias vistas de la información, pero dentro de un dominio acotado, que es el modelado.

RV-5 **Navegabilidad:** PB ofrece un modelo de navegabilidad sencillo, construido a partir de la experiencia recogida al estudiar los catálogos existentes. Provee varios modos de navegación permitiendo al usuario recorrer el catálogo en la forma que considere más adecuada.

RV-6 **Facilidades de búsqueda:** en este caso, se ha medido las opciones de búsqueda que ofrece cada visor. Si bien hay varios visores de catálogos que ofrecen funcionalidades de búsqueda, los más avanzados son PB, MDL y PHSA (basado en tecnología Google). El resto ofrecen interfaces más simples de búsqueda (en los casos aplicables).

RV-7 **Comunidad:** PB ofrece las mayores opciones de comunidad, permitiendo a los usuarios discutir en foros, integrarse con entornos colaborativos, etiquetar patrones, notificaciones, etc. PPR y PS ofrecen motores de edición colaborativa de alta aceptación y popularidad. Las demás herramientas de visualización ofrecen un subconjunto de las funcionalidades de PB.

RV-8 **Personalización de resultados:** PB, BT y EA permiten modificar la forma en que se consumirá una entidad a partir de opciones en sus interfaces de usuario.

17.4 Conclusiones de las Evaluaciones

En el **capítulo 10** analizamos un conjunto de herramientas (que son las que hemos utilizado como base para nuestra evaluación) y concluimos con una serie de limitaciones a modo de conclusión de ese análisis. A lo largo de los **capítulos 12**, **13** y **14** hemos establecido una serie de requisitos para cada una de las partes que conforman a nuestro modelo a partir de esas limitaciones y hemos enumerado al final de cada capítulo a modo de conclusión cómo nuestro modelo cumple esos requisitos. A partir de estos antecedentes y de las evaluaciones realizadas en este capítulo, ofrecemos las siguientes afirmaciones como conclusiones de la evaluación[264]:

- **Descripción de Patrones**: EML resuelve todos los problemas detectados en el **capítulo 10**. A partir de los resultados de la evaluación realizada en este capítulo, estamos en condiciones de afirmar que:

    1. EML permite describir a los patrones y conceptos de soporte a un alto nivel de abstracción, comprendiendo los niveles de conocimiento e implementación y abarcando todas las plantillas existentes.

    2. EML permite describir los niveles de descripción y conocimiento uniformemente a partir de un conjunto de construcciones básicas que se fundan sobre las mismas reglas sintácticas y semánticas.

    3. EML permite relajar la relación existente entre el grado de formalidad en la descripción, la rigidez y la expresividad, permitiendo describir

[264] Estas afirmaciones refuerzan las conclusiones parciales de los capítulos 12, 13 y 14

prosa en lenguaje natural y estructura y comportamiento en lenguajes formales a partir de elementos primitivos de un mismo lenguaje.

4. EML no está pensado para existir en aislamiento y por lo tanto permite integrar conocimiento de fuentes heterogéneas

5. EML no solo describe entidades, sino que también permite describir meta-información (relaciones, etiquetas, etc.) que puede ser utilizada en diversas formas.

6. Las meta-especificaciones EML no imponen ningún criterio sobre cómo deben ser utilizadas, dado que están a un alto nivel de abstracción que es totalmente independiente de la utilización que se haga de éstas.

- **Catalogación**: el catálogo de patrones y entidades resuelve todos los problemas detectados en el **capítulo 10**. A partir de los resultados de la evaluación realizada en este capítulo, estamos en condiciones de afirmar que:

1. El catálogo puede albergar cualquier tipo de entidad (patrones y conceptos de soporte), relaciones, meta-información, etc. Las entidades almacenadas en el catálogo pueden estar incompletas y ser evolucionadas luego de que se introducen en él.

2. El catálogo no es un contenedor de información pasivo y contiene un conjunto de funcionalidades que le gestionar sus contenidos y exponerlos al mundo.

3. El catálogo esta vivo: se pueden añadir y quitar entidades y expone una infraestructura para generar comunidad con el objeto de evolucionar sus contenidos para generar nuevos conocimientos a partir de los existentes.

4. El modelo de catalogación es sencillo y esta basado en las reglas de la emergencia, permitiendo modificar las relaciones entre las entidades alojadas en forma ad-hoc sin un coordinador de alto nivel, simplificando la evolución y crecimiento del catálogo.

5. El catálogo puede gestionar meta-especificaciones incompletas y evolucionarlas una vez que han sido introducidas. La clasificación, relación y anotación de las entidades pueden ser creadas en forma emergente y colaborativa.

6. El catálogo permite a sus usuarios realizar búsquedas a distintos niveles dentro del gradiente semántico.

7. Las relaciones entre los elementos del catálogo son semánticas.

- **Visualización y Navegación**: la herramienta de visualización del catálogo de patrones y entidades resuelve todos los problemas detectados en el **capítulo 10**. A partir de los resultados de la evaluación realizada en este capítulo, estamos en condiciones de afirmar que:

    1. El visor del catálogo es una aplicación Web que expone a los usuarios los contenidos del catálogo en forma usable y flexible, permitiéndoles elegir la forma en que quieren visualizar una entidad y el modo en que quieren recorrer el catálogo.

    2. El visor del catálogo expone los contenidos y funcionalidades del catálogo en forma sencilla mediante el uso de estándares y guías de usabilidad.

    3. El visor del catálogo ofrece funcionalidades para la creación de comunidad a efectos de potenciar la evolución y creación de conocimiento a partir de los contenidos del catálogo.

    4. El visor del catálogo permite al usuario utilizar múltiples modelos de navegación para recorrer sus contenidos, dándole la libertad de elegir el modelo cognitivo con el que se sienta más confortable.

    5. El visor del catálogo permite aplicar múltiples vistas sobre una misma entidad, permitiendo al usuario enfocarse en distintos aspectos de ésta según sus intereses y necesidades particulares.

Como corolario podemos decir que el modelo creado en esta tesis cubre las carencias recurrentes detectadas en los enfoques existentes (§ 10.4) en todos los aspectos que abarcan la descripción de patrones y entidades, su catalogación y posterior compartición con la comunidad de ingeniería del software.

Parte V
Conclusiones

"Un científico debe tomarse la libertad de
plantear cualquier cuestión,
de dudar de cualquier afirmación,
de corregir errores."
Julius Robert Oppenheimer

Capítulo 18
Conclusiones y Trabajo Futuro

"Vale más saber algo acerca de todo,
que saberlo todo acerca de una sola cosa"
Blas Pascal

Los patrones se han convertido en una herramienta de uso masivo y su ubicuidad se hace cada vez más evidente con el paso del tiempo. Constantemente se publican trabajos de investigación, libros, reportes corporativos y demás formas literarias que documentan lenguajes y sistemas de patrones en los dominios más diversos. Sin embargo, esta masificación no ha sido acompañada por un componente de orden. El movimiento de patrones es inherentemente anárquico y basado en elementos empíricos. La cantidad de patrones crece a un acelerado ritmo sin vistas de normalización.

En esta tesis se propone un modelo de meta-especificación y catalogación de patrones y conceptos como respuesta a las necesidades de gestión del conocimiento en este ámbito de la ingeniería del software.

El beneficio de la normalización u homogeneización en la descripción y gestión está guiado por un objetivo superior: la posibilidad de compartir conocimiento y experiencia en este ámbito e la ingeniería del software. Los modelos, tecnologías y conceptos desarrollados en esta tesis intentan ser un paso firme hacia este gran objetivo.

❖ ❖ ❖

18.1 Verificación, Contraste y Evaluación de los Objetivos

En el **capítulo 1** de esta memoria hemos planteado un grupo de objetivos específicos, los cuales han sido cumplidos durante el desarrollo de esta tesis doctoral. A continuación se enumeran dichos objetivos y se enuncia la forma en que ha sido conseguido.

- ***Crear un lenguaje de representación de patrones.*** EML es un lenguaje de meta-especificación que permite representar patrones y entidades a un alto nivel de abstracción y con flexibilidad respecto a la estructura de la entidad que se representa (debido a la utilización de estructuras que pueden ser modificadas en tiempo de diseño y ejecución). Las meta-especificaciones EML son auto-contenidas, es decir, un único fichero EML contiene toda la información necesaria para definir los niveles de conocimiento e implementación de un patrón o concepto de soporte. El DSL de EML dedicado a la descripción del comportamiento permite describir el nivel de implementación en forma totalmente independiente de los lenguajes de programación y plataformas de ejecución destino.

- ***Dotar al lenguaje de expresividad para describir conceptos.*** EML permite describir cualquier concepto. Esto incluye a los conceptos adicionales que permiten realmente comprender al patrón como por ejemplo lenguajes de patrones, categorías, tipos de patrones, autores, conceptos de orientación a objetos, etc. EML soporta la creación dinámica de nuevos tipos de conceptos que pueden ser creados en función de las necesidades de descripción de cualquier entidad. Como corolario de este punto y el anterior podemos decir que una meta-especificación EML puede describir cualquier entidad independientemente de su tipo.

- ***Dotar al lenguaje de capacidades relacionales.*** EML contiene un DSL orientado a la descripción de relaciones entre entidades (EML-RDL). Este DSL permite establecer relaciones entre entidades de distintos tipos, permitiendo crear redes de conceptos en forma emergente, idealmente en forma colaborativa. Esto hace que no sea necesario un diseño top-down de las relaciones entre entidades, permitiendo que estas emerjan y creen una red compleja y adaptativa de conceptos. Esto permite también crear redes que puedan modificarse en forma sencilla dinámicamente y es un elemento clave para poder gestionar información incompleta (por ejemplo, se pueden crear relaciones entre elementos que todavía no se han introducido en el catálogo)

- ***Dotar al lenguaje de capacidades de anotación.*** EML tiene soporte intrínseco para etiquetado[265] de entidades. Estas etiquetas pueden utilizarse tanto para búsquedas como para clasificación.

- ***Construir un catálogo de patrones.*** En el desarrollo de esta tesis se ha creado un catálogo de patrones y entidades meta-especificados utilizando EML.

- ***Crear la infraestructura de catalogación***. Se ha creado una infraestructura de catalogación que permite almacenar, anotar, buscar, enlazar y compartir un conjunto entidades. La infraestructura de catalogación incluye componentes de registro para incluir, modificar o quitar entidades del catálogo, búsquedas, anotación, compartición de entidades en diversas modalidades, colaboración, suscripciones, etc. El componente principal de la infraestructura de catalogación llamado FREP (*Flexible Runtime Execution Platform*) dota de un elevado grado de flexibilidad a nivel horizontal a nivel de arquitectónico. Esta flexibilidad se logra principalmente mediante la combinación de DLSs y AOMs.

- ***Crear una herramienta de visualización del catálogo y de los patrones y conceptos.*** El visor del catálogo permite a los usuarios navegar por el catálogo y utilizar las entidades que se encuentran alojadas en éste. La herramienta de visualización ha sido creada siguiendo estándares industriales de usabilidad e interacción, permitiendo a los usuarios interactuar con el catálogo en forma sencilla.

- ***Establecer un mecanismo de visualización de los patrones y conceptos***. La herramienta de visualización incluye un motor de vistas dinámico que permite registrar vistas y asociarlas a entidades en tiempo de diseño o ejecución. De esta forma es posible tener varias vistas asociadas a una entidad, lo cual permite al usuario elegir de qué forma desea visualizar una entidad concreta..

- ***Habilitar el trabajo colaborativo para evolucionar a los patrones.*** El visor del catálogo y la infraestructura de catalogación incluyendo foros de discusiones, soporte para shepherding, etc.

- ***Convertir al catálogo en un proveedor de servicios de información.*** El catálogo no es un repositorio pasivo de información. Además de proveer la infraestructura necesaria para incluir y gestionar entidades, puede exponerlas en varias formas. En el prototipo desarrollado, el catálogo puede exponer las entidades a través de un API orientada a objetos o de una interfaz de servicios Web.

[265] Tagging (en inglés)

El cumplimiento de estos objetivos específicos implica el cumplimiento de los objetivos generales (que han sido enumerados también en el **capítulo 1**), dado que se derivan directamente de éstos[266].

18.2 Síntesis del Modelo Propuesto

En esta tesis se propone un modelo de meta-especificación y catalogación de patrones y conceptos como respuesta a las necesidades de gestión del conocimiento en éste ámbito de la ingeniería del software. La arquitectura general de la solución propuesta (**capítulo 11**) para el problema planteado al inicio de esta tesis (**capítulo 1**) se compone de un lenguaje de meta-especificación para describir a los patrones a un alto nivel de abstracción (**capítulo 12**), un catálogo de patrones (**capítulo 13**) y una herramienta de explotación del catálogo (**capítulo 14**). Para verificar la factibilidad de la solución propuesta hemos creado un prototipo (**capítulos 15** y **16**) y lo hemos evaluado (**capítulo 17**) para demostrar que el modelo propuesto supera las dificultades recurrentes encontradas en otros enfoques existentes (**capítulo 10)**. En la lista a continuación se detallan brevemente las responsabilidades de cada uno de los elementos que conforman la solución:

- **EML (Entity Meta-Specification Languaje):** EML es un lenguaje de meta-especificación de patrones (y conceptos de soporte) basado en XML. Permite definir a estos elementos en un único fichero auto-contenido. Incluye construcciones que permiten representar en forma abstracta los niveles de descripción e implementación. Permite también describir relaciones entre entidades. El nivel de conocimiento incluye a todas las plantillas [C2PatternForm] existentes. El nivel de implementación permite describir el comportamiento del patrón en forma abstracta, independiente de los lenguajes de programación y entornos de ejecución.

- **Catálogo de Patrones:** catálogo de patrones descriptos utilizando EML. Contiene también conceptos asociados a los patrones descriptos en EML (lenguajes de patrones, categorías, fuentes de información, autores, conceptos de orientación a objetos, etc.). Tiene facilidades para poder registrar, vincular, anotar y buscar a las entidades que están incluidas en él. Expone los contenidos del catálogo a través de un API local (FREP) o de una interfaz de servicios Web

- **Visor del Catalogo**: aplicación de visualización del catálogo. Permite navegar por el catálogo de patrones en diversas formas. Permite ver a un patrón o concepto de muchas formas, permitiendo al usuario concentrarse en un aspecto específico de cada entidad. Registra los elementos más populares y contiene foros de discusión multi-hilo asociados a cada concepto, a efectos de desarrollar el sentido de comunidad y dar el sentido de "espacio vivo de información"

[266] El objetivo final es el mismo, pero están a diferentes niveles de agregación.

En la figura a continuación se muestra una visión general de la arquitectura completa de la solución. Cada uno de los elementos que se presentan en esta figura ha sido presentado en profundidad previamente en esta tesis en los **capítulos 11**, **12**, **13**, **14**. La implementación del prototipo ha sido presentada en los **capítulos 15** y **16**.

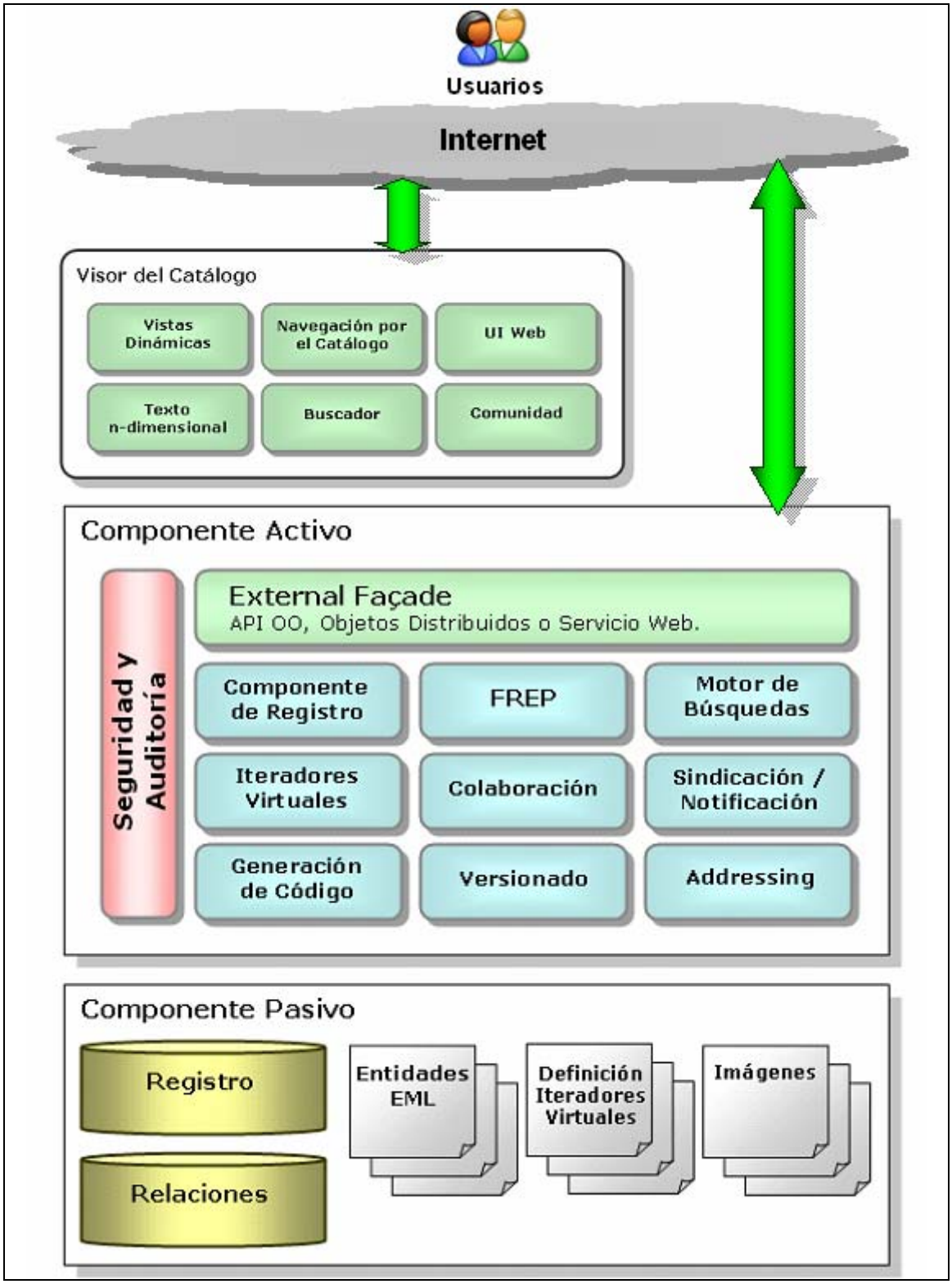


Figura 18.1 – Síntesis del modelo propuesto.

El uso conjunto de lenguajes de dominio específico (**capítulo 7**) y modelos de objetos adaptativos (**capítulo 8**) han permitido la creación de una plataforma ágil, flexible y extensible que resuelve los problemas recurrentes encontrados en otros enfoques existentes (**capítulo 10**). Esta plataforma puede describir patrones (**capítulo 3**) utilizando cualquiera de las plantillas existentes (**capítulo 4**) e inclusive crear nuevas en forma dinámica. Las entidades son descritas a un alto nivel de abstracción (**capítulo 6**). La aplicación del pensamiento emergente (**capítulo 9**) ha permitido sentar las de un modelo de gestión del conocimiento (**capítulo 5**) basado en la retroalimentación positiva que crece a partir de las interacciones entre los usuarios.

La aplicación de este modelo produce un cambio significativo en el ciclo de los patrones, haciéndolo más dinámico e interactivo, sentando las bases para la generación de conocimiento a partir de interacciones entre los miembros de una comunidad (que puede estar distribuida geográficamente). A partir de esta generación de conocimiento se promueve el refinamiento y mejora continua de los patrones almacenados en el catálogo, permitiendo así una evolución constante de los contenidos del catálogo adaptándose a cambios no previstos en su contexto.

En el nuevo ciclo de conocimiento la discusión y refinamiento no es más un paso en el medio del ciclo de vida, dado que la herramienta de visualización provee mecanismos para discutir en forma interactiva sobre cualquier entidad del catálogo. La publicación tampoco es un paso estático al final del ciclo: una vez que la meta-especificación existe y se introduce en el catálogo se "publica", estando disponible para todos los usuarios. Los autores pueden refinar sus patrones utilizando como entrada los comentarios y las discusiones sobre las entidades, producidos por usuarios del mundo entero. [Welicki06b]

Un aspecto interesante del nuevo ciclo es la distribución de la participación respecto al ciclo inicial: los primeros pasos son dados por un individuo o grupo reducido. Una vez que se realiza la publicación en el catálogo los pasos subsiguientes cuentan con la participación de toda la comunidad y además se retroalimentan, produciendo, refinando y evolucionando el conocimiento en forma emergente (figura 18.2).

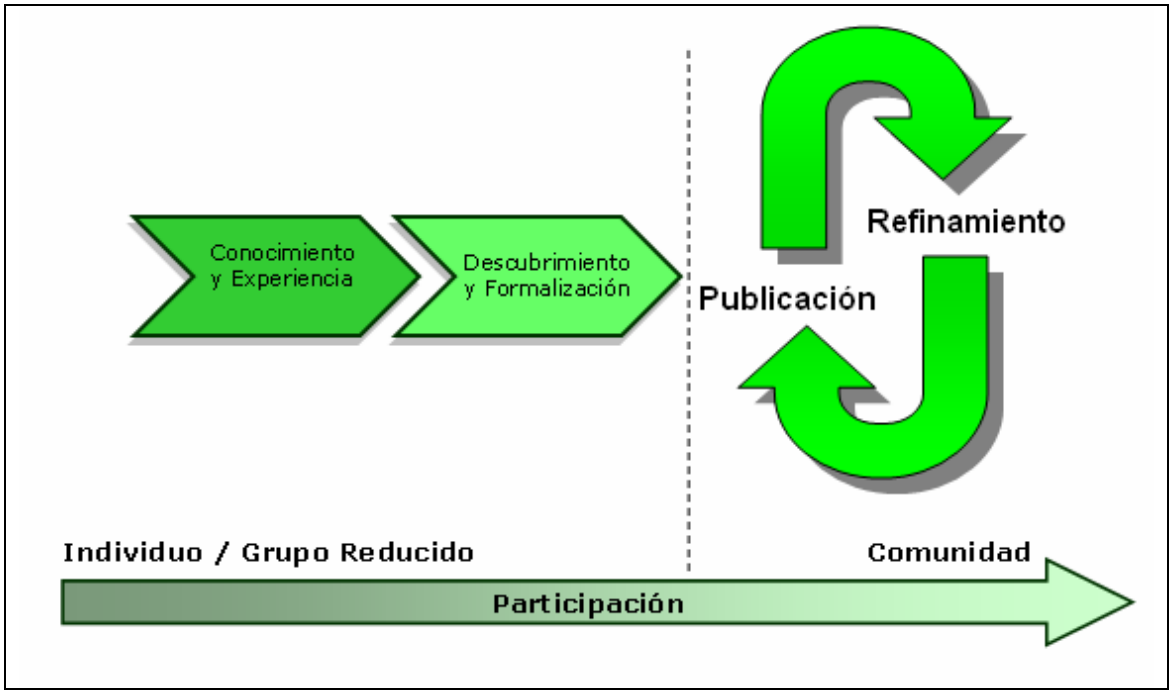


Figura 18.2 – Nuevo ciclo de vida de los patrones

18.3 Ámbitos de Aplicación

18.3.1 Ingeniería del Software

El estudio de los patrones es un área de la ingeniería del software que actualmente goza de gran interés tanto por el mundo académico como por el corporativo [Welicki05] [Welicki06]. Un conjunto de tecnologías y herramientas como los que se han planteado en este trabajo puedo ayudar a estos grupos en diferentes tareas, como por ejemplo la minería de patrones[267], estudio estructural de patrones, representación de nuevos patrones, registro de mejores prácticas, buenos diseños, capturar experiencias de diseño y arquitectura, etc.

18.3.2 RAD

RAD es el acrónimo de "*Rapid Software Development*" (desarrollo rápido de software). En [GoF95], se incluye la siguiente frase:

> *"Los patrones de diseño son soluciones estándar a problemas comunes en diseño orientado a objetos."*

De esta manera, el registro de estas soluciones estándar puede acelerar el tiempo de producción de buenos diseños. Adicionalmente, la generación de artefactos de software a partir de esas soluciones acelera aún más estos tiempos. El modelo presentado y desarrollado en esta tesis permite documentar, gestionar y compartir en forma eficiente estas soluciones.

[267] Pattern-minning

18.3.3 Asistente para Refactorización

El término "*refactorización*" se refiere a modificar un sistema de software de forma tal que no altere su comportamiento externo. En el libro "*Refactoring: improving the desing of existing code*" [Fowler99], se define este concepto de la siguiente forma:

> *"La refactorización es el proceso de cambiar un sistema de software de modo que no se altere el comportamiento externo del código pero se mejore su estructura interna. Es un modo disciplinado de limpiar el código que minimiza la posibilidad de introducir errores. En esencia, al refactorizar se mejora el diseño del código luego de que ha sido escrito."*

En el libro "*Refactoring to Patterns*" [Kerievsky04], se presenta la motivación de la realización de refactorización a patrones:

> *"Al explorar la motivación para la refactorización a patrones, encontré que era idéntica a la motivación para implementar refactorizaciones no basadas en patrones: reducir o quitar duplicidad, simplificar lo complejo y mejorar la forma en que nuestro código comunica su intención."*

El conjunto de tecnologías desarrollado ayuda a este tipo de refactorizaciones, haciendo que el catálogo de patrones sea fácilmente disponible, que sea fácil pasar de un patrón a otro y tener acceso a diferentes vistas del mismo patrón.

Durante el desarrollo del prototipo, se ha utilizado este enfoque en forma experimental. En el caso del generador de código, por ejemplo, se realizo primero un desarrollo del tipo SPAGHETTI CODE [BMMM98] y luego con la asistencia del mismo PatternsBrowser se realizo una refactorización a una estructura basada en el patrón BRIDGE [GoF95].

18.3.4 Desarrollo Basado en Patrones

El catálogo de patrones de Alexander, provee una serie de entradas generativas que pueden ser compuestas para lograr un diseño. Esto se ver reflejado en el siguiente comentario de Berczuk:

> *"El lenguaje de patrones de Alexander contiene alrededor de 250 patrones organizados de mayor a menor nivel. El objetivo de documentar los patrones que existen en arquitectura de software es llegar a un sistema similar, pero esto llevará tiempo." [Berczuk94]*

El desarrollo basado en patrones permite crear software mediante la composición de patrones. El conjunto de tecnologías desarrollado en este trabajo (EML (**capítulo 12**), el catálogo de patrones y conceptos entidad (**capítulo 13**) y PatternsBrowser (**capítulo 14**)) pueden ayudar a llevar a cabo este paradigma de desarrollo.

18.3.5 Documentación

Mediante el uso de patrones, se pueden documentar buenas prácticas y conocimiento avanzado de ingeniería del software, las cuales pueden ser registradas usando el conjunto de tecnologías desarrollado. Por ejemplo, si un equipo quisiera registrar patrones podría crear su propio lenguaje de patrones y registrarlo en el catálogo. EML permite documentar patrones y conceptos que pueden ser almacenados en el catálogo y compartidos mediante PatternsBrowser.

18.3.6 Gestión del Conocimiento

En [Joyanes03], se define a la gestión de conocimiento en la siguiente forma:

> *"Entregar a las personas los datos e información necesarios para ser eficientes en sus trabajos en cualquier lugar y momento, con cualquier dispositivo."*

Los patrones son un gran mecanismo de comunicación para almacenar y transmitir la experiencia de los ingenieros y diseñadores experimentados a los más noveles, convirtiéndose en unas de las vías para la gestión del conocimiento. Mediante su uso, podemos almacenar, compartir y transmitir conocimientos y experiencia entre los miembros de un equipo de trabajo y/o comunidad.

Como hemos mencionado anteriormente, los patrones permiten establecer un vocabulario común de ingeniería, cambiando el nivel de abstracción a colaboraciones entre clases y permitiendo comunicar experiencia sobre dichos problemas y soluciones.

El conjunto de tecnologías desarrollado ofrece la sintaxis, semántica e infraestructura para registrar, visualizar y utilizar todo tipo de patrones de software.

De esta manera, utilizando estas tecnologías podemos entregar la información referente al catálogo de patrones a las personas en cualquier lugar, momento y con cualquier dispositivo. En nuestro caso, esta información permite a los ingenieros de software hacer su trabajo en forma más eficiente.

18.3.7 Educación

Este conjunto de tecnologías tienen un amplio uso en la educación.

- **EML** permite estudiar desde un nivel alto de abstracción la estructura y comportamiento de los patrones a partir de su descomposición en elementos primitivos con dominios acotados y bien definidos.

- **El catálogo** de patrones permite agrupar patrones según criterios arbitrarios de homogeneidad, ofreciendo acceso centralizado a información que

actualmente esta distribuida en múltiples libros, revistas y sitios Web. Estos criterios pueden ser modificados ad-hoc, permitiendo crear distintas relaciones en forma colaborativa. Estas relaciones permiten recorrer los contenidos del catálogo de diversas formas, brindando la posibilidad de guiar a los estudiantes a través de un "camino" formado por un conjunto de entidades.

- **El visualizador de patrones** permite visualizar, navegar y utilizar el catálogo de patrones en forma sencilla
  - Tener acceso en forma sencilla a diferentes vistas del mismo patrón
  - Ver las relaciones existentes entre el patrón y otros patrones
  - Ver las relaciones existentes entre el patrón y otros conceptos (refactorizaciones, categorías, principios de diseño, etc.)
  - Navegar fácilmente por los diferentes lenguajes de patrones
  - Ver el código fuente resultante del patrón
  - Ver diagramas del patrón
  - Descubrir nuevas entidades mediante el modelo de "navegación por descubrimiento"

18.4 Aportaciones Originales

A continuación, se enumeran los principales beneficios y aportaciones que surgen de esta tesis:

- ✓ Un lenguaje de meta-especificación para describir patrones y conceptos asociados a un alto nivel de abstracción comprendiendo los niveles de conocimiento e implementación, abarcando todas las plantillas de patrones existentes. Permite describir también relaciones y metadatos para búsquedas y clasificación. (**capítulo 12**)

- ✓ Un catálogo de patrones y conceptos, para establecer una base de conocimiento práctico. El catálogo no es un mero contenedor de pasivo de información: contiene componentes activos que tienen como responsabilidad la gestión y compartición de sus contenidos. Contiene también mecanismos de comunidad que permite la evolución y generación de conocimientos a partir de sus contenidos. (**capítulo 13**)

- ✓ Una herramienta Web de visualización que expone a los usuarios los contenidos del catálogo en forma usable, ofreciéndoles varios mecanismos de navegación y visualización de entidades y la posibilidad de discutir sobre los contenidos (**capítulo 14**)

- ✓ Combinación de lenguajes de dominio específico (DSL) y modelos de objetos adaptativos (AOM) para resolver el problema de la meta-especificación catalogación de patrones de software en forma ágil y flexible (§ 13.4.2)

- ✓ Ciclo de vida de los patrones más dinámico e interactivo, estableciendo las bases para la evolución continua y generación de conocimiento a partir de interacciones entre los miembros de la comunidad (§ 11.6)

A continuación se enumeran otras aportaciones originales derivadas de esta tesis doctoral[268]:

- ✓ Formalización del ciclo de vida de los patrones (§ 11.5.4)
- ✓ Iteradores Virtuales (§ 13.4.4)
- ✓ Gradiente semántico (§ 12.2.5)
- ✓ Modelo de escritura n-dimensional (§ 14.5)
- ✓ Incluir un nivel de visualización en el AOM (§ 13.4.2.1)

18.5 Escrutinio Público

Los siguientes trabajos han sido presentados en diferentes congresos, publicaciones técnicas, simposios y jornadas de investigación, para difundir ante la comunidad científica los resultados que se reflejan en esta tesis[269].

1. **Meta-Specification and Cataloging of Software Patterns: Towards Knowledge Management in Software Engineering**
   León Welicki, Luis Joyanes Aguilar, Juan Manuel Cueva Lovelle
   21st ACM SIGPLAN Object Oriented Systems, Languages, and Applications Conference (OOPSLA 2006); Portland, Oregon, Estados Unidos, 22 al 26 de Octubre de 2006

2. **Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models**.
   León Welicki, Juan Manuel Cueva Lovelle, Luis Joyanes Aguilar
   11th European Conference on Pattern Languages of Programming Conference (EuroPLoP 2006); Irsee, Alemania, 5 al 9 de Julio de 2006

3. **A Model for Meta-Specification and Cataloging of Software Patterns**
   León Welicki, Oscar Sanjuán, Juan Manuel Cueva Lovelle
   12th Pattern Languages of Programming Conference (PLoP 2005); Monticello, Illionois, Estados Unidos, 6 al 10 de Septiembre de 2005

[268] Estas no son aportaciones principales, pero consideramos adecuado incluirlas en este apartado por su originalidad. Todas estas aportaciones han sido publicadas y contrastadas con la comunidad científica.
[269] Debido al carácter multidisciplinar de esta tesis los trabajos derivados se desarrollan en las áreas de ingeniería del software, patrones, gestión del conocimiento, interacción persona ordenador, inteligencia artificial, etc.

4. **Patrones y Antipatrones: una Introducción (Parte I)**
León Welicki
Revista MTJ .Net (Microsoft Developers Network - MSDN), Junio de 2005
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_2864.asp

5. **Patrones y Antipatrones: una Introducción (Parte II)**
León Welicki
Revista MTJ .Net (Microsoft Developers Network - MSDN), Julio de 2005
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3317.asp

6. **Patrones de Fabricación: Fábricas de Objetos**
León Welicki
Revista MTJ .Net (Microsoft Developers Network - MSDN), Octubre de 2005
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3624.asp

7. **El Patrón Singleton**.
León Welicki
Revista MTJ .Net (Microsoft Developer Network - MSDN), Marzo de 2006
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_4081.asp

8. **The Configuration Data Caching Pattern**
León Welicki
13th Pattern Languages of Programmin Conference (PLoP 2006); Portland, Oregon, Estados Unidos, 21 al 23 de Octubre de 2006

9. **La Falacia Operacional y los Sistemas Emergentes**
León E. Welicki, Juan Manuel Cueva Lovelle, Luis Joyanes Aguilar
IV Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento (SISOFTW 2006); Cartagena, Colombia, 23 al 26 de Agosto de 2006.

10. **Una Propuesta de Aplicación del Paradigma Emergente para un Modelo de Agregación Dinámica de Servicios Basado en SOA**
León Welicki, Juan Manuel Cueva Lovelle
I Jornadas Científico-Técnicas en Servicios Web del W3C (JSWEB 2005); Granada, España, 13 y 14 de Septiembre de 2005. Thomson, ISBN: 84-9732-455-2

11. **Una Plataforma Basada en Sistemas Multiagentes y Servicios Web para Monitorización de Aplicaciones en Entornos Heterogéneos**
León Welicki, Juan Manuel Cueva Lovelle
II Taller en Desarrollo de Sistemas Multiagente (DESMA 2005); Granada, España, 13 de Septiembre de 2005. Thomson, ISBN: 84-9732-454-4

12. **Modificación de la Intención Semántica de Relatos Utilizando Ontologías y Técnicas Evolutivas**
    Jesús Soto, León E. Welicki
    III Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB 2005). Granada, 14 al 14 de Septiembre de 2005. Thomson.

13. **Hacia la Convergencia entre la Organización Formal y Real en Equipos de Software a Través de un Modelo de Desarrollo**
    León E. Welicki, Juan Manuel Cueva Lovelle
    III Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento (SISOFTW 2005); Santo Domingo, República Dominicana, 24 al 26 de Agosto de 2005. Mc Graw-Hill, ISBN: 84-689-3411-9

14. **XText: Un Modelo para Publicación de Textos en Múltiples Dispositivos**
    León Welicki
    I Jornadas de Postgrado de Investigación en Ingeniería Informática (JPIII 2004). Salamanca, España, 7 al 8 de Mayo de 2004. Publicaciones Universidad Pontificia de Salamanca, ISBN: 84-688-6550-8

15. **Perception of Software Problems on the Internet World**
    Jorge A. Salido, León E. Welicki
    III Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento (SISOFTW 2005). Santo Domingo, República Dominicana, 24 al 26 de Agosto de 2005. Mc Graw-Hill, ISBN: 84-689-3411-9

16. **Las Personas en las Metodologías de Ingeniería del Software**
    León Welicki, Fernando Cano García
    V Jornadas de Informática y Sociedad (JIS 2004). Bilbao, España, 25 al 26 de Marzo de 2004. Publicaciones Universidad de Deusto, ISBN 84-7485-927-1

17. **El Precio de las Palabras en la Sociedad de la Información**.
    León Welicki, Sergio Berman
    V Jornadas de Informática y Sociedad (JIS 2004). Bilbao, España, 25 al 26 de Marzo de 2004, Publicaciones Universidad de Deusto, ISBN 84-7485-927-1

18. **Modificación de la Semántica de Textos Mediante Algoritmos Genéticos**.
    León Welicki
    III Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB 2004). Cordoba, España, 3 al 5 de Febrero de 2004, Publicaciones Universidad de Córdoba, ISBN 84-688-4224-9.

19. **Desarrollo Multilenguaje con Patrones de Diseño en .NET**
    León Welicki, Fernando Cano García
    II Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento (SISOFTW 2003). Lima, Perú, 20 al 23 de Agosto de 2003), ISBN: 1501032003-3819

20. **Implementando Extreme Programming en la Plataforma .NET**
    León Welicki
    II Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento (SISOFTW 2003). Lima, Perú, 20 al 23 de Agosto de 2003), ISBN: 1501032003-3819

21. **La Gestión del Conocimiento en las Empresas de Consultoría Informática**
    León Welicki
    II Congreso Internacional de la Sociedad de la Información y el Conocimiento (CISIC 2003). Madrid, España, 7 al 9 de Mayo de 2003. Mc Graw-Hill, ISBN 84-481-3966-6

22. **Un Modelo de Compartición de Conocimiento en Red para Integrar Comunidades del Tercer Mundo**
    León Welicki
    II Congreso Internacional de la Sociedad de la Información y el Conocimiento (CISIC 2003). Madrid, España, 7 al 9 de Mayo de 2003. Mc Graw-Hill, ISBN 84-481-3966-6

18.6 Líneas de Investigación Futuras

Si bien el objetivo de este trabajo (establecido en forma clara y concisa al inicio de esta tesis en el **capítulo 1**) ha sido cumplido, todavía queda mucho trabajo por hacer. El modelo puede ser refinado y extendido para soportar características adicionales. En la lista a continuación se enumeran las principales líneas de investigación y trabajo futuro:

- **Editor de EML (Colaborativo)**: diseño y construcción de un entorno d edición para crear meta-especificaciones EML de entidades. Este entorno ayudará al usuario a crear meta-especificaciones en forma sencilla y ágil a través de ayudas y mediante una interfaz gráfica. Debe permitir también la edición colaborativa de entidades y dar soporte a la creación de patrones por más de un autor, asistiendo así a procesos de construcción de conocimiento colaborativos como el shepherding [Shepherding].

- **Combinar EML-PDL con Wiki**: añadir sintaxis de wiki [PPR] [Wikipedia] a las propiedades de EML-PDL para poder crear relaciones rápidas entre entidades. Las relaciones wiki descritas en una propiedad serán sintácticas, permitiendo conectar entidades entre sí en forma rápida dentro de un bloque de texto del nivel de conocimiento (§ 11.1.2).

- **Combinar entre EML-RDL y RDF**: combinar los triples de EML-RDL con RDF [W3C] a efectos de anotar a las entidades con metadatos. Estos metadatos pueden ser almacenados en una base de datos RDF [RDFDB] y ser utilizados para realizar búsquedas complejas.

- **Evolución de EML-BDL y EML-SDL** para soportar más construcciones primitivas. Dotar de más expresividad a los lenguajes de descripción de la estructura y del comportamiento a efectos de poder describir un mayor espectro de conceptos. Una posible línea de trabajo es la búsqueda de convergencia e integración entre EML-BDL / EML-SDL y MSIL [MSNET].

- **Integración con WOP**: integrar el EML y el catálogo con la ontología ODOL [WOP]. De esta forma podemos combinar las capacidades de descripción, relación, establecimiento de redes ad-hoc y visualización avanzada de nuestro modelo combinándolas con información con alta carga semántica que permitirá realizar búsquedas más complejas sobre el catálogo. Esto posibilita la creación de un motor de inferencias que puede ayudar a un usuario a encontrar el patrón que mejor se adapte a un contexto específico.

- **Integración con CoPE**: CoPE [SS06] es un entorno de edición de patrones colaborativo basado en wiki con capacidades de visualización de mapas de patrones. El objetivo de esta línea de trabajo es aprovechar las capacidades de layout de mapas de entidades de CoPE para generar mapas de subconjuntos arbitrarios de elementos del catálogo. Por otro lado, CoPE puede beneficiarse de las capacidades de nuestro modelo como la utilización de múltiples plantillas, la combinación de múltiples lenguajes de patrones, la visualización dinámica, etc. La integración entre ambos enfoques puede realizarse mediante la interfaz de servicios Web del catálogo. El primer paso en vías de esta integración ha sido dado en el EuroPLoP 2006, donde el autor de esta tesis y el autor de CoPE han comenzado a delinear posibles líneas de colaboración.

- **Integración con herramientas de desarrollo:** crear plugins para entornos de desarrollo (como por ejemplo Visual Studio.NET [VSNET] o Eclipse [Eclipse]) que permitan navegar y utilizar los contenidos del catálogo. De esta forma los desarrolladores podrán utilizar el conocimiento almacenado en el catálogo sin la necesidad de abandonar sus entornos de trabajo y generar artefactos software en forma automática.

- **Crear mas traductores para EML**, para mejorar la expresividad de los ejemplos del catálogo. Algunos lenguajes previstos son C++, Python, Java, Perl y Smalltalk. Se prevé también la creación de un traductor a XMI para integración con herramientas de modelado y generación de diagramas en tiempo de ejecución (la visualización de estos diagramas es realizada por un visor de XMI para la Web).

Bibliografía y Referencias

1. Publicaciones (Libros, Revistas, Actas y Reportes)

[37Signals04] 37 Signals. *Defensive Design for the Web: How to Improve Error Messages, Help, Forms, and Other Crisis Points*. New Riders. 2004.

[Acebal02] Acebal, Cesar; Juan Manuel Cueva Lovelle. *A new method for software development: Extreme Programming*. Revista Novatica. Abril 2002.

[ACM01] Allur, Deepak ; John Crupi; Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education. 2001.

[AENOR00] AENOR. *Sistemas de Gestión de la Calidad: Fundamentos y Vocabulario (ISO 9000:2000)*. Madrid. Diciembre de 2000.

[AENOR00b] AENOR. *Sistemas de Gestión de la Calidad: Requisitos (ISO 9001:2000)*. Madrid. Diciembre de 2000.

[AIS77] Alexander, Christopher; Sara Ishikawa; Murray Silversteing. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1977.

[Alexander79] Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press. 1979.

[Alvarez99] Alvarez, Darío. *Sistemas Integrales Orientados a Objetos*. Publicaciones de la Universidad de Oviedo. 1999.

[Ambler03] Ambler, Scott. *The Elements of UML Style*. Cambridge University Press. 2003.

[Arsanjani01] Arsanjani, Ali. *Rule Object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction*. Actas del 8th Pattern Languages of Programs Conference (PLoP 2001). Septiembre de 2001, Illionois, USA.

[BD99] Brown, John Seely; Paul Duguid. *La vida social de la información*. Prentice Hall. 2001.

[Beck00] Beck, Kent. *Extreme Programming Explained: Embrace the Change*. Addison Wesley. 2000.

[Beck96] Beck, Kent. *Smalltalk Best Practice Patterns*. Prentice Hall. 1996.

[Berczuk94] Berczuk, Steve. *Finding Solutions through Pattern Languages*. Computer 27, Número 12. Diciembre de 1994

[BernersLee00] Berners Lee, Tim. *Weaving the Web: The original design and ultimate destiny of the World Wide Web*. Collins. 2000.

[Bertalanffy68] Von Bertalanffy, Ludwig. *Teoría General de los Sistemas*. George Braziller. 1968.

[Betancourt87] Betancourt Gálvez, Carlos *Elementos de Lógica y Reflexiones sobre Filosofía*. UTEHA. México, 1987.

[BFVY96] Budinsky; Frank; Marilyn Finnie; John Vlissides; Patsy Yu. *Automatic Code Generation from Design Patterns*. IBM Systems Journal, Vol. 35, No. 2. 1996.

[BHL01] Berners-Lee, Tim; James Hendler; Ora Lassila. *The Semantic Web*. Scientific American, Mayo 2001.

[BMMM98] Brown, William; Raphael Malveau; Hays Mc Cormick III; Tom Mowbray. *Antipatterns: Refactoring Software, Architectures and Project in Crisis*. Wiley and Sons. 1998.

[Box02] Box, Don. *Essential .NET: The Common Language Runtime*. Addison-Wesley. 2002

[Canals02] Canals, Agusti. *Gestión del conocimiento*. Gestión 2000. Madrid. 2002.

[Cooper99] Cooper, Alan. *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity*. Prentice Hall. 1999.

[Cornella03] Cornella, Alfons. *Hacia la empresa en Red*. Madrid. Gestión2000. 2003

[Cueva03] Cueva Lovelle, Juan Manuel et al. *International Conference on Web Engineering (ICWE 2003), Oviedo, Spain, July 14-18, 2003. Proceedings. Lecture Notes in Computer Science 2722*. Springer. 2003.

[Cueva98] Cueva Lovelle, Juan Manuel. *Procesadores de Lenguajes*. Editorial Servitec. 1998.

[Czarnecki00] Czarnecki, Krzysztof; Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional. 2000.

[DGD03] Djuric, Dragan; Dragan Gasevic, Vladan Devedzic. *A MDA-based Approach to the Ontology Definition Metamodel*. Proceedings of the 4th Workshop on Computational Intelligence and Information Technologies. October 2003. Serbia

[DKT05] Deng, Junhua; Elizabeth Kemp; E. G. Todd. *Managing UI pattern collections*. Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: making CHI natural. 2005.

[DOS03] Daconta, Michael; Leo Obrst; Kevin Smith. *The Semantic Web: A guide to the future of Web Services and Knowledge Management*. Willey & Sons. 2003.

[DP99] Davenport, Thomas; Larry Prusak. *Working Knowledge: How Organizations Manage What They Know*. Prentice Hall. 1999.

[Emmeche94] Emmeche, Claus. *The Garden in the Machine: the emerging science of artificial life*. Princeton University Press. 1994

[Emmott04] Emmot, Stephen. *Open Innovation: a new model for parthership with universities*. Segundas Jornadas Académicas de Microsoft Research. Barcelona, Marzo de 2004.

[Evitts00] Evitts, Paul. *A UML Pattern Language*. SAMS Publishing. 2000.

[Ford06] Ford, Neal. *No Fluff, Just Stuff 2006: An Antology*. Pragmatic Bookshelf. 2006.

[Fowler02] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional. 2002.

[Fowler03b] Fowler, Martin. *Who Needs an Architect?* IEEE Design September / October 2003, pp 11-13.

[Fowler96] Fowler, Martin. *Analysis Patterns: Reusable Object Models*. Adisson-Wesley Professional. 1996.

[Fowler99] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Adisson-Wesley Professional. 1999.

[FS99] Fowler, Martin, Scott Kendall. *UML Distilled: Applying the Standard Object Modelling Language (2nd Edition)*. Adisson-Wesley Professional. 1999.

[FY98] Foote, Foote ; Joseph W. Yoder *Metadata and Active Object-Models*. Proceedings of the Fifth Conference on Patterns Languages of Programs (PLoP '98) Monticello, Illinois. Agosto 1998.

[Gall75] Gall, John. *Systemantics: How Systems Work and Especially How They Fail*. Quadrangle. 1975.

[GHV95] Gamma, Erich; Richard Helm, John Vlissides. *Desing Patterns Applied*. Tutorial en OOPSLA (OOPSLA 95)

[GoF95] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides. *Design Patters: Elements of Reusable Object Oriented Software*. Addison Wesley. 1995.

[GS04] Greenfield, Jack; Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley. 2004.

[HC05] Hasso, Sargon; C. R. Carlson. *A Theoretically-based Process for Organizing Design Patterns*. Actas del 12th Pattern Languages of Programs Conference (PLoP 2005). Septiembre de 2005, Illionois, USA.

[Helland04] Helland, Pat. *Metropolis: Envisioning the Service Oriented-Enterprise*. Microsoft Architects Journal EMEA Edition vol. 2, 2004

[HT00] Hunt, Andy; David Thomas. *The Pragmatic Programmer: from Journeyman to Master*. Adisson Wesley. 2000.

[HW03] Hophe, Gregor; Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Adisson-Wesley. 2003.

[IEEE90] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York. 1990.

[Izquierdo02] Izquierdo Castanedo, Raul. *RDM: Arquitectura software para el modelado de dominios en sistemas informáticos*. Tesis Doctoral, Universidad de Oviedo, 2002.

[JCJO92] Jacobson, Ivar; Magnus Christerson; Patrick. Jonsson; Gunnar Övergaard, G. *Object Oriented Software Engineering: A Use Case Driven Approach*. ACM Press. 1992.

[Johnson01] Johnson, Steven. *Sistemas Emergentes, o qué tienen en común hormigas, neuronas, ciudades y software*. The Free Press. 2001.

[Johnsonbaugh97] Johnsonbaugh, Richard. *Matemáticas Discretas (cuarta edición)*. Pearson Education. 1997.

[Joyanes00] Joyanes Aguilar, Luis. *Portales de conocimiento de empresa*. Datamation nº 116 Mayo 2000. pp.20.22

[Joyanes03] Joyanes Aguilar, Luis. *Curso de Doctorado de Tecnologías y Metodologías para la construcción de Sistemas de Gestión del Conocimiento*. Universidad Pontificia de Salamanca, campus de Madrid. 2003.

[Joyanes03b] Joyanes Aguilar, Luis. *Ingeniería Web: ¿Una nueva profesión?* Revista Data TI, Número 197, Marzo de 2003

[Joyanes03c] Joyanes Aguilar, Luis; Manuel Gonzalez, Rodríguez. *De la Sociedad de la Información a la Sociedad del Conocimiento: Los frutos de 60 años de informática*. Actas del III Congreso Internacional de la Sociedad de la Información y el Conocimiento (CISIC 2003), Madrid, España, 2003

[Joyanes97] Joyanes Aguilar, Luis. *Cibersociedad: Los Retos Sociales del Siglo XXI*. Mc Graw-Hill. 1997.

[Joyanes98] Joyanes Aguilar, Luis. *El capital intelectual. La nueva riqueza de las empresas*. Datamation nº 147. Sept. Barcelona 1998 pp. 62-70.

[Joyanes99] Joyanes Aguilar, Luis. *La sociedad del conocimiento ¿Hacia una nueva utopía?* Convergencia IT vol. 1 nº1 1999 pp. 45-51

[Kappel04] Kappel, Gerti. *Web Engineering - Old Wine, New Bottles?* Proceedings of the Fourth International Conference on Web Engineering (ICWE 2004), Munich, 2004.

[Kerievsky04] Kerievsky, Josua. *Refactoring to Patterns.* Addison-Wesley Professional. 2004.

[Kimball98] Kimball, Ralp. *Meta Meta Data Data: Making a List of Data About Data and Exploring Information Cataloging Tools.* DBMS Magazine, Marzo de 1998.

[Knuth92] Knuth, Donald. *Literate Programming.* Center for the Study of Language and Information - Lecture Notes. 1992.

[KR88] Kernighan, Brian; Dennis Ritchie: *El Lenguaje de Programación C.* Prentice-Hall. 1988.

[Krug99] Krug, Steve. *Don't make me think!* Addison Wesley. 1999.

[KS93] Korth, Henry; Abraham Silberschatz. *Fundamentos de Bases de Datos (segunda edición).* Mc Graw Hill. 1993.

[Kurzweil99] Kurzweil, Raymond. *The Age of Spiritual Machines.* 1999.

[KWB03] Kleppe, Ann; Jos Warmer; Win Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise.* Addison Wesley. 2003.

[Larman99] Larman, Craig. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall. 1999.

[Lowy03] Lowy, Juval. *Programming .NET Components.* O'Reilly. 2003.

[Martin02] Martin, Robert C. *Agile Software Development: Principles, Patterns and Practices.* Prentice Hall. 2002.

[McComb03] McComb, David. *Semantic in Business Systems: The Savvy Manager's Guide.* Morgan Kaufmann Publishers. 2003.

[Microsoft03] Microsoft Corp. *Arquitectura de Aplicaciones para .NET: Diseño de aplicaciones y Servicios.* Microsoft Press. 2003.

[Microsoft03b] Microsoft Corp. *Enterprise Solution Patterns.* Microsoft Press. 2003.

[Microsoft04] Microsoft Corp. *Enterprise Development Reference Architecture.* Microsoft Press. 2004.

[MN02] Martin, Robert C; James Newkirk. *La Programación Extrema en la práctica*, Addison Wesley. 2002.

[MR04] Manns, Mary, Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley. 2004.

[Murugesan99] Murugesan, Sam et al. *Web Engineering: A New Discipline for Development of Web-based Systems*. Proceedings of the Workshop on Web Engineering (ICSE 99), ACM Press, New York, 1999.

[Nadhan04] Nadhan, Easwaran. *Service Oriented Architecture: Implementation Challenges*. Microsoft Architects Journal EMEA Edition vol. 2, 2004

[Nielsen90] Nielsen, Jakob; R. Molich. *Heuristic evaluation of user interfaces*. Proceedings of the ACM CHI'90 Conference. (Seattle, WA, 1-5 April), pp 249-256.

[Nielsen94] Nielsen, Jakob. *Enhancing the explanatory power of usability heuristics*. Proceedings of the ACM CHI'94 Conference. (Boston, MA, April 24-28), pp 152-158.

[Nielsen94b] Nielsen, Jakob. *Heuristic evaluation*. In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley & Sons, New York, NY

[Nielsen99] Nielsen, Jakob. *Usabilidad: Diseño de Sitios Web*. Prentice-Hall. 1999.

[NT94] Nonaka, Ikujiro; Hirotaka Takeuchi. *La organización creadora de conocimiento: Cómo las compañías japonesas crean la dinámica de la innovación*. Oxford University Press. 1994.

[Ortin02] Ortín Soler, Francisco. *Sistema computacional de programación flexible diseñado sobre una máquina abstracta reflectiva no restricitiva*. Tesis Doctoral, Universidad de Oviedo, 2002.

[Parra05] Parra Fuentes, Javier. *Servicios Web Reflectivos y Adaptables Dinámicamente sobre Servidores de Aplicaciones soportados por Máquinas Virtuales*. Tesis Doctoral, Universidad Pontificia de Salamanca, 2005.

[Perera04] Perera, Kevin: *Metadata-driven Application Design and Development*. Microsoft Architects Journal EMEA Edition vol. 2 pp. 19-26. 2004.

[POSA00] Schmidt, Douglas; Michael Stal; Hans Rohnert; Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. Wiley & Sons. 2000.

[POSA03] Kircher, Michael; Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. Wiley & Sons. 2003.

[POSA96] Buschmann, Frank; Regine Meunier; Hans Rohnert; Peter Sommerland; Michael Stal. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996.

[Pressman02] Pressman, Roger. *Software Engineering (5th Edition)*. Mc Graw Hill. 2002.

[RBP91] Rumbaugh, James; Michael Blaha; William Lorensen; Frederick Eddy; William Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall. 1991.

[Resnick94] Resnick, Mitchel. *Tortugas, Termitas y Atascos de Tráficos: Exploraciones en Micromundos Masivamente Paralelos*. MIT Press. 1994.

[Rising00] Rising, Linda. *The Patterns Almanac 2000*. Addison-Wesley. 2000

[SS06] Schobert, Wolfram; Till Schümmer. *Supporting Pattern Language Visualization with CoPE*. Actas del 11th European Conference on Pattern Languages of Programs - EuroPLop 2006 (Irsee, Alemania, 5 al 9 de Julio de 2006)

[ST01] Shalloway, Allan; James Trot. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley Professional. 2001.

[SW04] Sprott, David; Laurence Wilkes. *Understanding Service-Oriented Architecture*, Microsoft Architects Journal EMEA Edition vol. 1, pp 07-11, 2004

[SW05] Salido, Jorge, León Welicki. *Perception of Software Problems on the Internet World*. Actas del III Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento - SISOFTW 2005 (Santo Domingo, República Dominicana, Agosto de 2005)

[Tidwell05] Tidwell, Jenifer. *Designing Interfaces*. O'Reilly Media. 2005.

[Turing36] Turing, Alan. *On computable numbers with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society. Series 2. Vol. 43. pp. 230-245. 1936.

[Turing54] Turing, Alan. *Computing machinery and intelligence*. Mind. 59. pp. 433-460. 1950

[Vlissides98] Vlissides, John. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. 1998.

[W3CES05] World Wide Web Consortium (W3C), Oficina Española. *Guías Breves de Tecnologías W3C*. 2005

[Welicki03] Welicki, León. *Un modelo de compartición de conocimiento en red para integrar comunidades del tercer mundo*. Actas del 3er Congreso Internacional de la Sociedad de la Información y el Conocimiento (CISIC 03), Madrid, España, 2003.

[Welicki03b] Welicki, León. *La Gestión del Conocimiento en las Empresas de Servicios de Consultoría Informática*. Actas del 3er Congreso Internacional de la Sociedad de la Información y el Conocimiento (CISIC 03), Madrid, España, 2003

[Welicki03c] Welicki, León. *Implementando Extreme Programming en la Plataforma .NET*. Actas del 2do Simposio Internacional de Ingeniería del Software (SISOFTW 03), Lima, Perú, 2003

[Welicki03d] Welicki, León; Cano García, Fernando. *Desarrollo Multilenguaje con Patrones de Diseño en la Plataforma .NET*. 2do Simposio Internacional de Ingeniería del Software (SISOFTW 03), Lima, Perú, 2003

[Welicki03e] Welicki, León. *Escritura para la Web*. Trabajo de investigación para el curso de doctorado "Interacción Hombre Máquina". Universidad Pontificia de Salamanca, campus Madrid. 2003.

[Welicki04] Welicki, León. *Arquitecturas de Software Agiles, Flexibles y Robustas para Ingeniería Web*. Trabajo de investigación tutelada de Doctorado, Universidad Pontificia de Salamanca, campus de Madrid. 2004.

[Welicki04b] Welicki, León. *XText, Un modelo para la publicación de textos en multiples dispositivos*. Actas de las 1ras Jornadas de Postgrado de Investigación en Ingeniería Informática (JPIII), Mayo de 2004, Salamanca, España

[Welicki04c] Welicki, León. *Meta-especificación y Catalogación de Patrones de Software*. Tesis de Master, Universidad Pontificia de Salamanca, campus Madrid, Diciembre de 2004.

[Welicki04d] Welicki, León; Fernando Cano García: *Las Personas en las Metodologías de Ingeniería del Software*. Actas de las 5tas Jornadas de Informática y Sociedad (JIS 04), Marzo de 2004, Bilbao, España

[Welicki04e] Welicki, León; Sergio Berman. *El Precio de las Palabras en la Sociedad de la Información*. Actas de las 5tas Jornadas de Informática y Sociedad (JIS 04), Marzo de 2004, Bilbao, España.

[Welicki04g] Welicki, León. *Utilización de Algoritmos Genéticos para la modificación de la semántica de textos*. Actas del Tercer Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB 04), Febrero de 2004, Córdoba, España.

[Welicki05] Welicki, León; Oscar Sanjuán Martinez; Juan Manuel Cueva Lovelle. *A Model for Pattern Meta-Specification and Cataloging*. Actas del 12th Pattern Languages of Programs Conference (PLoP 2005). Septiembre de 2005, Illionois, USA.

[Welicki05b] Welicki, Leon; Cueva Lovelle Juan Manuel. *Una plataforma basada en sistemas multiagentes y servicios Web para monitorización de aplicaciones en entornos heterogéneos*. Actas del Segundo Taller de Sistemas Multiagentes - DESMA 2005 (Granada, España, Septiembre de 2005).

[Welicki05c] Welicki, Leon; Juan Manuel Cueva Lovelle. *Una propuesta de aplicación del paradigma emergente para un modelo de agregación dinámica de servicios basado en SOA*. Actas de las Primeras Jornadas Científico-Técnicas sobre Servicios Web del W3C - JSWEB 2005 (Granada, España, Septiembre de 2005)

[Welicki05f] Welicki, Leon; Juan Manuel Cueva Lovelle. *Hacia la convergencia entre la organización formal y real en equipos de software a través de un modelo de desarrollo*. Actas del III Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento - SISOFTW 2005 (Santo Domingo, República Dominicana, 24 al 26 de Agosto de 2005)

[Welicki06] Welicki, Leon; Juan Manuel Cueva Lovelle; Luis Joyanes Aguilar. *Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models*. Actas del 11th European Conference on Pattern Languages of Programs - EuroPLop 2006 (Irsee, Alemania, 5 al 9 de Julio de 2006)

[Welicki06b] Welicki, Leon; Luis Joyanes Aguilar; Juan Manuel Cueva Lovelle. *Meta-Specification and Cataloging of Software Patterns: Towards Knowledge Management in Software Engineering*. OOSPLA Companion- OOPSLA 2006 (Portland, USA, 22 al 26 de Octubre de 2006)

[Welicki06d] Welicki, Leon; Juan Manuel Cueva Lovelle; Luis Joyanes Aguilar. *La Falacia Operacional y los Sistemas Emergentes*. Actas del IV Simposio Internacional de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento - SISOFTW 2006 (Cartagena de Indias, Colombia, 23 al 25 de Agosto de 2006)

[Woolf97] Woolf, Bobby. *Type Object Pattern*. Proceedings of the Fourth Conference on Patterns Languages of Programs (PLoP '97) Monticello, Illinois. 1997.

[YBJ01] Yoder, Joseph; Federico Balaguer; Ralph Johnson. *Architecture and Design of Adaptive Object Models*. Intriguing Technology Presentation at the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), ACM SIGPLAN Notices, ACM Press, December 2001.

[YFRT98] Yoder, Joseph ; Brian Foote ; Dirk Rhiele ; Michael Tilman. *Metadata and Active Object Models Workshop Results Submission*. OOPSLA Addendum 1998.

[YJ02] Yoder, Joseph ; Ralph Johnson. *The Adaptive Object Model Architectural Style*. Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02) at the World Computer Congress in Montreal 2002, August 2002.

[Yourdon93] Yourdon, Edward. *Análisis Estructurado Moderno*. Prentice Hall, 1993.

[YR00] Yoder, Joseph; Reza Razavi. *Metadata and Adaptive Obejct Models*. ECOOP 2000 Workshop Reader. Lecture Notes in Computer Science vol. 1964; Springer Verlag. 2000.

2. Recursos Web

[AA01] Agile Alliance.
http://www.agilealliance.org/

[ACM] Association for Computing Machinery
http://www.acm.org

[AM01] The Agile Manifesto.
http://agilemanifesto.org/

[AOM] MetaData and Adaptive Object-Model Pages.
http://www.adaptiveobjectmodel.com/

[Bathia03] Bathia, Shushant. *An Introduction to META Modelling.*
http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=698

[BC99] Burch / Cheswick map of the Internet. 28 June 1999.
http://research.lumeta.com/ches/map/gallery/isp-ss.gif

[BID] Borland IDE : Delphi 2006.
http://www.borland.com/us/products/delphi/index.html

[BJB] Borland: JBuilder
http://www.borland.com/us/products/jbuilder/index.html

[Booch] Booch, Grady. *Handbook of Software Architecture.*
http://www.booch.com/architecture/index.jsp

[Borland] Borland Corporation
http://www.borland.com

[Bredemeyer01] Bredemeyer, Dana; Ruth Malan. *Architecture Definitions.* 2001.
http://www.bredemeyer.com/pdf_files/Definitions.pdf

[Bredemeyer02] Bredemeyer, Dana. *What It Takes to be a great Architect.* 2002.
http://www.bredemeyer.com/pdf_files/GreatArchitect.PDF

[Bredemeyer02b] Bredemeyer, Dana; Ruth Malan. *Introduction to Software Architecture: Why? What? Where? and Who?* 2002
http://www.bredemeyer.com/pdf_files/ArchitectureIntroduction.PDF

[BS03] Bañegil Palacios, Tomás; Ramón Sanguino Galván. *Gestión del conocimiento y estrategia.* Madri+d, Número 19. Octubre de 2003.
http://www.madrimasd.org/revista/revista19/tribuna/tribuna3.asp

[C2ME] C2 Wiki: Meditations on Emergente
http://c2.com/cgi/wiki?MeditationsOnEmergence

[C2Meta] C2 Wiki: MetaLevel
http://c2.com/cgi/wiki?MetaLevel

[C2MP] C2 Wiki: MetaPatterns
http://c2.com/cgi/wiki?MetaPatterns

[C2PatternForm] Portland Pattern Repository: Pattern Forms.
http://c2.com/cgi/wiki?PatternForms

[C2WDP] C2. Wiki Design Principles.
http://c2.com/cgi/wiki?WikiDesignPrinciples

[CBDI] CBDI Forum Web Site
http://www.cbdiforum.com

[CHI2003] CHI 2003 : New Horizonz
http://www.chi2003.org/index.cgi

[CJ2P] Core J2EE Patterns. Best Practices and Design Strategies.
http://www.corej2eepatterns.com/Patterns2ndEd/index.htm

[Clarin] Clarin.com
http://www.clarin.com

[Cockburn02] Cockburn, Alistair. *Learning From Agile Software Development - Part I*. 2002.
http://www.stsc.hill.af.mil/crosstalk/2002/10/cockburn.html

[CORBA] The OMG's Corba Web Site
http://www.corba.org/

[Cornella98] Cornella, Alfons. *La infoestructura: Un concepto esencial en la sociedad de la información*, Barcelona. ESADE, 1998.
http://www.infonomics.net/cornella/ainfost.pdf

[Cueva04] Cueva Lovelle, Juan Manuel. *Tecnología de Objetos: Patrones de Diseño*. 2004.
http://di002.edv.uniovi.es/~cueva/investigacion/lineas/patrones/PatronesIntroduccion.pdf

[Cueva05] Cueva Lovelle, Juan Manuel. *Ingeniería Web*.
http://di002.edv.uniovi.es/~cueva/investigacion/lineas/web/index.html

[Cutter00] Cutter Consortium. *Poor Project Management Number-One Problem of Outsourced E-Projects*. 2000.
http://www.cutter.com/research/2000/crb001107.html

[Delicious] Del.icio.us
http://www.del.icio.us

[DICT04] Dictionary.com. *Definición de GIGO*.
http://dictionary.reference.com/search?q=GIGO

[DKV00] van Deursen, Arie; Paul Klint; Joost Visser. Domain-Specific Languages: An Annotated Bibliography. 2000.
http://homepages.cwi.nl/~arie/papers/dslbib/

[DoFactory] Data & Objects Factory: Design Patterns.
http://www.dofactory.com/Patterns/Patterns.aspx

[Dreamsongs] Dreamsongs.com. Richard P. Gabriel Home Page.
http://www.dreamsongs.com/

[EA] Sparx Systems. Enterprise Architect.
http://www.sparxsystems.com/products/ea.html

[EA03] Sparx Systems: Enterprise Architect Documentation, 2003
http://www.sparxsystems.com/

[EAAC] Fowler, Martin. *A short summary of the patterns in Patterns of Enterprise Application Architecture (P of EAA).*
http://www.martinfowler.com/eaaCatalog/

[EAXL] Enterprise Architect 6.1 User Guide. XMI Limitations
http://www.sparxsystems.com/EAUserGuide/index.html?limitationxmi.htm

[Eclipse] Eclipse Project
http://www.eclipse.org

[EIPC] Enterprise Integration Patterns Catalog.
http://enterpriseintegrationpatterns.com/eaipatterns.html

[ElMundo] El mundo digital
http://www.elmundo.es

[ElPais] El Pais
http://www.elpais.es

[Estatella05] Estatella, Adolfo. La folksonomía emerge como sistema para clasificar contenidos en colaboración. El Pais. 8 de Septiembre de 2005.
http://www.elpais.es/articulo/elpcibred/20050908elpcibenr_1/Tes

[ET] ET++ Home Page
http://www.ivtools.org/ivtools/etplusplus.html

[Fincher04] Fincher, Sally. *PLML Extension : Concerns*. 2004.
http://www.cs.kent.ac.uk/people/staff/saf/patterns/concerns.html

[Flickr] Flickr.com
http://www.flickr.com

[Fowler00] Fowler, Martin. *The New Methodology*. 2000.
http://www.martinfowler.com/articles/newMethodology.html

[Fowler04] Fowler, Martin. *Inversion of Control Containers and the Dependency Injection Pattern*. Enero de 2004.
http://www.martinfowler.com/articles/injection.html

[Fowler04b] Martin Fowler. *Domain Specific Language*. Febrero 2004
http://www.martinfowler.com/bliki/DomainSpecificLanguage.html

[Fowler05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* Junio 2005.
http://www.martinfowler.com/articles/languageWorkbench.html

[Frechot03] Frechot, Joselyn. *Domain-Specific Languages - An Overview*.
http://compose.labri.fr/documentation/dsl/dsl_overview.php3

[FUH] FernUniversitaet in Hagen
http://www.fernuni-hagen.de/

[Garcia01] García Muiña, Fernando Enrique. *Análisis del capital intelectual de las organizaciones desde la teoría de recursos y capacidades y la teoría del conocimiento. Concepto y componentes*. Revista MadrI+D. Diciembre 2001 / Enero 2002.
http://www.madrimasd.org/revista/revista8/aula/aulas1.asp

[GIF] GIF89a Specification
http://www.w3.org/Graphics/GIF/spec-gif89a.txt

[Google] Google.com
http://www.google.com

[Gruber93] Gruber, Tom. *What is an Ontology?*
http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[GSA] Google Search Appliance
http://www.google.com/enterprise/

[GUID] Webopedia. *What is a GUID?*.
http://www.webopedia.com/TERM/G/GUID.html

[Hartford] The Hartford Finantial Services.
http://www.thehartford.com/

[Hibernate] Hibernate.org
http://www.hibernate.org

[Hillside] Hillside Group - Home of the Patterns Library
http://hillside.net

[Hotdraw] University of Illinois at Urbana-Champaign. *HotDraw Home Page*.
http://st-www.cs.uiuc.edu/users/brant/HotDraw/

[Hotmail] Hotmail.com
http://www.hotmail.com

[IIT] Illinois Institute of Technology
http://www.iit.edu

[Interfaces03] British HCI Group. *Interfaces*. Número 56, Otoño 2003. http://www.bcs-hci.org.uk/interfaces/interfaces56.pdf

[IRC] Irc http://www.irchelp.org/

[J2EE] Java 2 Platform Enterprise Edition http://java.sun.com/javaee/

[Java04] Sun Microsystems. *Java Techonology*. http://java.sun.com

[JO98] Johnson, Ralph ; Jeff Oakes. The User Defined Product Framework. 1998. http://st-www.cs.uiuc.edu/users/johnson/papers/udp/UDP.pdf

[Johnson98] Johnson, Ralph. *Dynamic Object Model*. 1998. http://st-www.cs.uiuc.edu/users/johnson/papers/dom/DynamicObjectModel.pdf

[Joyanes02b] Joyanes Aguilar, Luis. *Cambios Sociales ¿Realidad o Utopía?* Octubre de 2002 http://www.gestiondelconocimiento.com/leer.php?colaborador=ljoyanes&id=133

[JPG] Jpeg.org http://www.jpeg.org/

[Labra04] Labra Gayo, Emilio. Tecnologías XML y Web Semántica http://www.di.uniovi.es/~labra/cursos/ext04

[Lea03] Lea, Doug. *Pattern Writing Checklist*. 2003 (Hillside Group). http://www.hillside.net/patterns/writing/checklist.htm

[Lea97] Lea, Doug. *Christopher Alexander:An Introduction for Object-Oriented Designers* http://gee.cs.oswego.edu/dl/ca/ca/ca.html

[Maholtra98] Maholtra, Yogesh. *Knowledge Management for the New World of Business*. 1998. http://www.kmnetwork.com/whatis.htm

[Malan96] Malan Ruth, et al. *Why do we need Software Architecture*. Fusion Newsleter, April 1996. http://www.bredemeyer.com/why.htm

[Marco01] Marco, David. Advanced Meta Data Architecture. 2001. http://www.tdan.com/i013fe01.htm

[MB02] Malan, Ruth; Dana Bredemeyer. *Software Architecture: Central Concerns, Keys Decisions*. 2002. http://www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF

[McCormick98] McCormick, Hays. *Antipatterns Tutorial*. 1998. http://www.antipatterns.com/briefing/sld001.htm

[Meyer02] Meyer, Bertrand. *Achieving Interoperability*. Dr Dobbs Journal. Marzo 2002. http://www.ddj.com/dept/architect/184414864

[Meyer02b] Meyer, Bertrand. *Polyglot Programming*. Dr Dobbs Journal. Abril de 2002. http://www.ddj.com/dept/architect/184414854

[MHS05] Mernik, Marjan; Jan Heering, Anthony M. Sloane. *When and how to develop domain specific languages*. Centrum voor Wiskunde en Informatica. Reporte SEN-E0517. Holanda, Amsterdam. Diciembre 2005. http://ftp.cwi.nl/CWIreports/SEN/SEN-E0517.pdf

[Microsoft] Microsoft Corporation http://www.microsoft.com

[MJ04] MDA Journal. *Domain Specific Modeling and Model Driven Architecture*. Enero 2004. http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Modeling%20Frankel-Cook.pdf#search=%22keith%20short%20mda%20mof%22

[MN97] Morkes, John; Jakob Nielsen. *Concise, SCANNABLE, and Objective: How to write for the web*. 1997. http://www.useit.com/papers/webwriting/writing.html

[MN98] Morkes, John; Jakob Nielsen.*Applying writing guidelines to web pages*. 1998. http://www.useit.com/papers/webwriting/rewriting.html

[MOUDIL] Montreal Online Usability Digital Library (MOUDIL) http://hci.cs.concordia.ca/moudil/

[MSAN] The Official Microsoft ASP.NET 2.0 Site. http://www.asp.net

[MSDNSQL] Microsoft Developers Network. *Sql Server Reference – System Tables*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sys_00_690z.asp?frame=true

[MSDSL] Visual Studio 2005 - Domain-Specific Language Tools http://msdn.microsoft.com/vstudio/DSLTools/

[MSNET] Microsoft Corp, .NET Framework Home Page http://www.microsoft.com/net/

[MSPP] Microsoft Patterns & Practices http://msdn.microsoft.com/practices/

[MSSQL] Microsoft Coporation. Microsoft Sql Server.
http://www.microsoft.com/sql/

[MSVB] Microsoft Corportation. Microsoft Visual Basic.
http://msdn.microsoft.com/vbasic/

[MSWF] .NET Framework Windows Forms.
http://www.windowsforms.net/

[MSWF] Windows Forms
http://www.windowsforms.net

[MU] Massey University.
http://www-ist.massey.ac.nz/

[MW] Merriam-Webster on line Dictionary.
http://www.m-w.com/cgi-bin/dictionary

[NET] Microsoft Corporation. Microsoft *.NET Homepage*.
http://www.microsoft.com/net/default.mspx

[NETRemoting] Microsoft Developers Network. *.NET Remoting*.
http://msdn.microsoft.com/webservices/remoting/default.aspx

[NHibernate] NHibernate for .NET.
http://www.nhibernate.org

[Nielsen] Nielsen, Jakob. *UseIt: Jakob Nielsen on Usability and Web Design*.
http://www.useit.com

[Nielsen94c] Nielsen, Jakob. *Guerrilla HCI: Using Discount Usability Engineering to Penetrate the Intimidation Barrier*. 1994.
http://www.useit.com/papers/guerrilla_hci.html

[Nielsen94d] Nielsen, Jakob. *Ten usability Heuristics*.
http://www.useit.com/papers/heuristic/heuristic_list.html

[Nielsen96] Nielsen. Jakob: *Inverted Pyramids in Cyberspace*
http://www.useit.com/alertbox/9606.html

[Nielsen97] Nielsen, Jakob: *Be Succint! (Writing for the Web)*
http://www.useit.com/alertbox/9703b.html

[Nielsen97b] Nielsen, Jakob: *How Users Read on the Web* <en línea>
http://www.useit.com/alertbox/9710a.html

[Nielsen99b] Nielsen, Jakob: *Prioritize: Good Content Bubbles to the Top*
http://www.useit.com/alertbox/991017.html

[OASIS02] OASIS. *Technology Reports: XML Metadata Interchange (XMI)*. Mayo de 2002. http://www.oasis-open.org/cover/xmi.html

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

[OMGMDA] The Object Management Group. *Model Driven Architecture*. http://www.omg.org/mda/

[OMGMOF] The Object Management Group. *Meta Object Facility (MOF) Specification V 1.3.1*. 2001. ftp://ftp.omg.org/pub/docs/formal/01-11-02.pdf

[OMGXMI] The Object Management Group. *XML Metadata Interchange (XMI). Version 1.1*. 1999. http://www.omg.org/docs/ad/99-10-02.pdf

[OR02] Entrevista a Steven Jonson. O´Reilly Network. 22/02/2002 http://www.oreillynet.com/pub/a/network/2002/02/22/johnson.html

[Oracle] Oracle Coporation http://www.oracle.com/

[PatternShare] Microsoft Corp. Pattern Share. http://www.patternshare.org/

[PAW] Weir, Charles. *Patterns Almanac 2000 (Web)*. http://www.smallmemory.com/almanac/index.html

[PCW] Principia Cybernetica Web http://pespmc1.vub.ac.be/

[PHSA] Booch, Grady. *Patterns (Handbook of Software Architecture)*. http://www.booch.com/architecture/patterns.jsp

[PID] Welie, Martijn van. *Patterns in Interaction Design*. http://www.welie.com/

[PLML] Fincher, Sally. *PLML : Pattern Language Markup Language* http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html

[PLML03] CHI 2003 *Perspectives on HCI Patterns : Concepts and Tools (introducing PLML)*. 2003. http://www.cs.kent.ac.uk/people/staff/saf/patterns/CHI2003WorkshopReport.doc

[PLMLX] Bienhaus, Diethelm. *PLMLx - Extended Pattern Language Markup Language*. http://www.cs.kent.ac.uk/people/staff/saf/patterns/diethelm/plmlx_doc/plml_doc.dtd.html

[PLoP] Pattern Languages of Programs Conference
http://hillside.net/plop/

[PLoP05] Pattern Languages of Programs Conference 2005
http://hillside.net/plop/2005

[PPR] Portland Pattern Repository
http://c2.com/ppr/

[PSWK] The Professional Site of Jos Warmer and Anneke Kleppe. *What is the Model Driven Architecture?*
http://www.klasse.nl/mda/mda-introduction.html

[RAE] Real Academia Española
http://www.rae.es

[Rational] IBM Rational Software
http://www-306.ibm.com/software/rational/

[RDFDB] RDFDB : An RDF Database
http://www.guha.com/rdfdb/

[RJ96] Roberts, Don; Ralph Johnson. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks.*
http://st-www.cs.uiuc.edu/~droberts/evolve.html

[RSS] RDF Site Summary (RSS) 1.0 Oficial Specification
http://web.resource.org/rss/1.0/

[RssReader] RssReader.com
http://www.rssreader.com/

[SAG] Software Architecture Group (SAG) at University Illinois at Urbana-Champaign.
http://wiki.cs.uiuc.edu/SAG/

[SAGAOM] Software Architecture Group (SAG) at University Illinois at Urbana-Champaign. *Active/Dynamic/Adaptative Object Module Architectures.*
http://wiki.cs.uiuc.edu/SAG/Active%2FDynamic%2FAdaptive+Object-Model+architectures

[SAP] SAP.com
http://www.sap.com

[SBPC] Sun Microsystems. *Core J2EE Patterns.*
http://java.sun.com/blueprints/corej2eepatterns/

[SharpReader] Sharp Reader (Free RSS/Atom Agregator)
http://www.sharpreader.net/

[Shepherding] Hillside.net : Shepherding
http://hillside.net/shepherding.html

[SJFC] Sun Microsystems. *Java Foundation Classes (JFC) / Swing.*
http://java.sun.com/products/jfc/

[Slashdot] Slashdot.com
http://www.slashdot.com

[Smalltalk] Smalltalk.org
http://www.smalltalk.org/

[Spinellis01] Diomidis Spinellis. *Notable design patterns for domain specific languages*. Journal of Systems and Software, 56(1):91–99. February 2001.
http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html

[StarLogo] MIT. *StarLogo on the Web.*
http://education.mit.edu/starlogo/

[Stroupstrup] Bjarne Stroustrup's homepage
http://www.research.att.com/~bs/

[Struts] The Apache Software Foundation. *Apache Struts.*
http://struts.apache.org/

[Tabor] Tabor Communications. *Inxight Introduces Hyperbolic Tree Server.*
http://www.taborcommunications.com/dsstar/00/0822/102080.html

[Tag] Technoratti. *Tags: The real-time web, organized by you.*
http://www.technorati.com/tag/

[TagCloud] TagCloud.com
http://www.tagcloud.com/

[TCDM] The Computing Dictionary. Definición de "Meta".
http://computing-dictionary.thefreedictionary.com/Meta

[TCGN03] The Code Generation Network. *Code Generator Models.*
http://www.codegeneration.net/tiki-index.php?page=ModelsIntroduction

[Technoratti] Technoratti.com
http://www.technorati.com/

[Tidwell] Tidwell, Jenniffer. *Designing Interfaces. Patterns for Effective Interaction Design.*
http://designinginterfaces.com/

[Together] Borland Corp. *Together Technologies.*
http://www.borland.com/together/

[Tognazini00] Tognazini, Bruce. *First Principles of Interaction Design*. 2000. http://www.asktog.com/basics/firstPrinciples.html

[TON] Tech on the Net. Oracle PL/SQL System Tables. http://www.techonthenet.com/oracle/sys_tables/

[TSN] The Server Side .NET – .NET Community http://www.theserverside.net

[TSS] The Server Side Community – Java Community http://www.theserverside.com

[UC] Universidad CAECE http://www.caece.edu.ar

[UFHT] Usability First. *Usability Glossary: Hyperbolic Tree*. http://www.usabilityfirst.com/glossary/term_360.txl

[UIUC] University of Illinois at Urbana-Champaign http://www.uiuc.edu/

[UML] Object Management Group (OMG). *Unified Modeling Language (UML)* http://www.uml.org

[UO] Universidad de Oviedo http://www.uniovi.es/

[UPEX] Universidad Pontificia de Salamanca, campus Madrid. *Doctorado en Ingeniería del Software - Programa de Excelencia*. http://www.upsam.es/index.php?Mod=Estudios&Section=Mostrar&IdEstudio=5&IdOpcionMenu=701&Lang=es

[UPSA] Universidad Pontificia de Salamanca http://www.upsa.es

[UPSAM] Universidad Pontificia de Salamanca, campus Madrid http://www.upsam.es

[VSNET] Microsoft Corp. Visual Studio .NET http://msdn.microsoft.com/vstudio/

[VSTSM] Microsoft Corp. *Visual Studio 2005 Team System Modeling Strategy and FAQ*. Mayo 2005. http://msdn.microsoft.com/vstudio/DSLTools/default.aspx?pull=/library/en-us/dnvs05/html/vstsmodel.asp

[W3C] World Wide Web Consortium (W3C) http://www.w3c.org

[W3C04] World Wide Web Consortium (W3C). *Web Services Glossary* (February 2004)
http://www.w3.org/TR/ws-gloss/

[W3C04b] World Wide Web Consortium (W3C). *Web Services Architecture* (February 2004)
http://www.w3.org/TR/ws-arch

[W3C99] World Wide Web Consortium (W3C). *HTML 4.01Specification.*
http://www.w3.org/TR/REC-html40/

[W3CSWE] World Wide Web Consortium (W3C). *Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering.*
http://www.w3.org/2001/sw/BestPractices/SE/ODA/

[W3CXS] World Wide Web Consortium (W3C) *XML Schema 1.1.*
http://www.w3.org/XML/Schema

[Welicki05d] Welicki, Leon. *Patrones y Antipatrones: Una Introducción (Parte I)*, Revista MTJ.NET, Junio de 2005.
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/mtj_2864.asp

[Welicki05e] Welicki, Leon. *Patrones y Antipatrones: Una Introducción (Parte II)*, Revista MTJ.NET, Junio de 2005.
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3317.asp

[Welicki05g] Welicki, León. *Patrones de Fabricación: Fábricas de Objetos*. Revista MTJ.NET, Noviembre de 2005.
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3624.asp

[Welicki06c] Welicki, León. *El Patrón Singleton*. Revista MTJ.NET, Marzo de 2006.
http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_4081.asp

[Wijngaarden03] van Wijngaarden, Jonne. *Code Generation from a Domain Specific Language: Designing and Implementing Complex Program Transformations*. 2003. Masters Thesis Software Technology, Institute of Information and Computing Sciences, Faculty of Mathematics & Computer Science, Utrecht University.
http://www.cs.uu.nl/~visser/ftp/Wij03.pdf

[Wikipedia] Wikipedia: The free encyclopedia
http://en.wikipedia.org/wiki/Main_Page

[WOP] Web of Patterns Project
http://www-ist.massey.ac.nz/wop/

[WPW] Wikipedia - Wiki
http://en.wikipedia.org/wiki/Wiki

[WSI] Web Services Interoperability
http://www.ws-i.org/

[WSJ] Wall Street Journal on-line
http://online.wsj.com/public/us

[WWISA] Word Wide Institute of Software Architects Home Page
http://www.wwisa.org

[XDEFAQ] IBM : Rational XDE FAQ, 2004
http://www-128.ibm.com/developerworks/rational/library/4304.html

[Yahoo] Yahoo
http://www.yahoo.com

[YJ03] Yoder, Joseph ; Ralph Johnson. *Architecture and Design of Adaptive Obejct Models*. 2003.
http://www.adaptiveobjectmodel.com

[ZIP] ZIP File Format Specification
http://www.pkware.com/index.php?option=com_content&task=view&id=64&Itemid=107

Parte VI
Apéndice

"Lo bueno si breve, dos veces bueno"
Baltasar Gracián

Apéndice A
Patterns Happy

El término ***Patterns Happy*** aparece en el libro "*Refactoring to Patterns*" [Kerievsky04] y es un calificativo aplicable a los programadores que se refiere al uso indiscriminado de patrones, aún cuando su utilización no agrega ningún valor a la solución que se esta desarrollando. Un programador *Patterns Happy* aprende un patrón e "*inmediatamente encuentra un lugar para abusar de él*" [Kerievsky04]

Para clarificar mejor este concepto, incluyo un breve párrafo de "*Refactoring to Patterns*" [Kerievsky04] donde se define de esta forma este calificativo:

> *Did you ever learn a software pattern and then want to use it right away because you liked it so much? If so, you've been Patterns Happy.*

A.1 Ejemplo: Versión "Pattern Happy" del Hello World

A modo de ejemplo, en esa misma obra se incluye la siguiente pieza de código en Java:

```
interface MessageStrategy {
    public void sendMessage();
}


abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}

class MessageBody {
    Object payload;

    public Object getPayload() {
        return payload;
    }

    public void configure(Object obj) {
        payload = obj;
    }

    public void send(MessageStrategy ms) {
        ms.sendMessage();
    }
}

class DefaultFactory extends AbstractStrategyFactory {
    private DefaultFactory() {;}

    static DefaultFactory instance;
```

```
    public static AbstractStrategyFactory getInstance() {
        if (instance == null)
            instance = new DefaultFactory();
        return instance;
    }

    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println((String) obj);
            }
        };
    }
}

public class HelloWorld {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
        mb.configure("Hello World!");
        AbstractStrategyFactory asf = DefaultFactory.getInstance();
        MessageStrategy strategy = asf.createStrategy(mb);
        mb.send(strategy);
    }
}
```

Código A.1 – Implementación "Patterns Happy" del clásico "Hello World"

El código de arriba es la implementación del clásico "*Hello World*", pero con varios patrones (ABSTRACT FACTORY, STRATEGY, SINGLETON, MEMENTO). Este programa que debería ser comprensible por cualquier (de hecho, es el ejemplo arquetípico del primer programa que aprende un programador) se torna bastante complicado. La implementación de arriba ofrece sofisticadas características de flexibilidad y configuración que probablemente nadie utilizará: la solución padece de sobre-ingeniería. Este es el caso que se intenta evitar en XP[270] con el principio de *YAGNI* (*You Aren't Gonna Need It*) [Beck00].

Ser *Pattern Happy* no es un problema de los programadores principiantes. Los programadores intermedios y avanzados también pueden tener este problema. De hecho, entre sus efectos colaterales se incluyen los antipatrones GOLDEN HAMMER, WOLF TICKET, SWISS ARMY KNIFE e INTELECTUAL VIOLENCE [BMMM98].

[270] Extreme Programming (XP) [Beck00]

Apéndice B
EML 1.0

EML es el acrónimo de *Entity Meta-specification Language*. EML es un lenguaje de dominio específico textual de tipo "*high-level modularly composable*" [Czarnecki00], basado en XML, creado con el objetivo de describir todo tipo de patrones de software y conceptos de soporte (lenguajes de patrones, categorías, refactorizaciones, autores, principios de orientación a objetos, etc.).

B.1 Descripción General de EML

EML provee la infraestructura necesaria para describir los niveles de conocimiento e implementación junto con las relaciones entre patrones y conceptos de soporte. En sus primeras versiones [Welicki04c] [Welicki05] sólo estaba preparado para describir patrones utilizando plantillas fijas estáticas previamente definidas (basadas en elementos estandarizados[271]). En la versión actual [Welicki06] [Welicki06b] (que es la que se presenta en esta tesis) puede describir cualquier tipo de entidad utilizando plantillas dinámicas. También puede describir relaciones entre entidades del mismo o diferentes tipos (por ejemplo, puede expresar que "*un patrón está contenido en un lenguaje de patrones*", "*un patrón conforma un principio de orientación a objetos*", etc.). Las entidades descriptas utilizando EML deben tener información literaria (nivel de conocimiento) y pueden tener información de comportamiento (nivel de implementación).

B.1.1 Sublenguajes de EML

EML es un "*high-level modularly composable*" [Czarnecki00] que incluye cinco sub-lenguajes de dominio específico más limitados, pero más especializados:

- **Properties Definition Language (EML-PDL):** expresa la información literaria (parte fundamental del nivel de conocimiento) de una entidad a partir de composición dinámica de propiedades atómicas (por ejemplo, la plantilla de un patrón es un conjunto de propiedades).

- **Relationships Definition Language (EML-RDL):** expresa relaciones entre entidades. Las entidades pueden ser del mismo o diferentes tipos. Se

[271] En [Welicki04c] se presenta la plantilla general que se utiliza para describir a los patrones en las primeras versiones de EML (llamado entonces M4PS). Si bien permitía describir un amplio número de patrones no era flexible y no permitía describir a todas las plantillas existentes. Adicionalmente no servía para describir conceptos de soporte (dado que había sido diseñada sólo para describir patrones).

pueden expresar relaciones incompletas, donde uno de los elementos no existe o no está registrado en el catálogo.

- **Annotation Language (EML-AL):** anotación de la entidad mediante etiquetas.

- **Structure Definition Language (EML-SDL):** expresa la estructura de la entidad (participantes, relaciones y responsabilidades)

- **Behavior Definition Language (EML-BDL):** expresa el comportamiento de los participantes de la entidad (declarados usando EML-SDL) a un alto nivel de abstracción. Contiene abstracciones de construcciones comunes en lenguajes de programación.

B.1.2 Secciones de una Entidad EML

EML permite representar una entidad en un único fichero. Una representación EML es auto-contenida: esto significa que contiene toda la información necesaria para expresar un patrón o un concepto[272], incluyendo la información literaria, la información de implementación y metadatos para clasificación y búsquedas. Esta representación puede ser utilizada con múltiples propósitos (§ 12.2.6), como por ejemplo, visualizar la información literaria, generar artefactos de software (modelos, código fuente, XMI, etc.), generar documentación, ver relaciones, etc.

EML no se limita a la descripción de patrones, permitiendo describir cualquier tipo de entidad (lenguajes de patrones, refactorizaciones, categorías, conceptos de orientación a objetos, libros, autores, etc.)

Una meta-especificación EML se compone de las siguientes partes:

- **Identificación**: información general para identificar a la entidad (nombre de la entidad, tipo de entidad).

- **Etiquetas (Tags)**: etiquetas para anotar a las entidades [Technoratti]. Decora a la entidad con metadatos descriptivos que pueden ser utilizados para búsquedas o categorización.

- **Relaciones**: relaciones entre la entidad que esta siendo descripta y otras entidad (por ejemplo "ABSTRACT FACTORY está contenido en el lenguaje de patrones del GoF", "FACTORY METHOD es un patrón de diseño", "ABSTRACT FACTORY contiene a FACTORY METHOD", etc.)

- **Resumen**: un resumen de la entidad que se está describiendo.

[272] Tiene toda la información necesaria sobre la entidad que se está describiendo y relaciones con otras entidades que aumentan la información sobre ésta.

- **Plantilla (Template):** información literaria de la entidad. En el caso de los patrones, contiene la información de la plantilla [C2PatternForm] (para más detalles sobre las plantillas existentes, ver el **capítulo 4**).

- **Estructura**: Estructura de la entidad (participantes con sus relaciones y responsabilidades).

- **Implementación**: información sobre el comportamiento de la entidad (nivel de implementación). En esta sección tenemos dos partes bien diferenciadas: la implementación base, que es el modelo general de comportamiento y las implementaciones concretas, que son ejemplos específicos del comportamiento de la entidad[273].

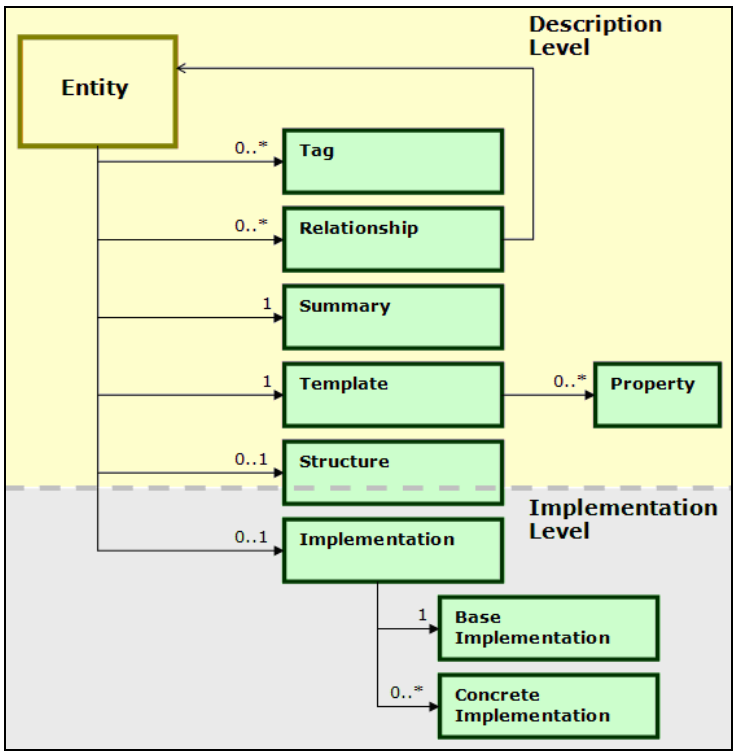


Figura B.1 – Definiendo entidades con EML (tomado de [Welicki06]).

Como puede observarse en la figura B.1, el nivel de conocimiento se compone de las secciones *Anotaciones*, *Relaciones*, *Resumen*, *Plantilla* y *Estructura*. El nivel de implementación consiste de las secciones *Estructura* e *Implementación*. En la misma figura puede observarse también que la sección *Estructura* es compartida por los niveles de conocimiento e implementación. Esto se debe a que contiene una enumeración formal de los participantes y sus relaciones (nivel de implementación) y la descripción textual de las responsabilidades de los participantes (nivel de conocimiento).

[273] Podría establecerse un paralelismo a modo de ejemplo entre la implementación base y una clase base abstracta y la implementación concreta y una clase concreta que implementa la clase base abstracta.

Las secciones *Estructura* e *Implementación* son opcionales. Algunas entidades pueden no necesitarlas. Por ejemplo, los patrones de "*Design Patterns*" [GoF95] tienen ambos niveles. Los patrones de "*Analysis Patterns*" [Fowler96] sólo tienen nivel de conocimiento. Los antipatrones [BMMM98] en algunos casos tienen ambos niveles (LAVA FLOW, THE BLOB, etc. [BMMM98]) y en otros sólo tienen el nivel de conocimiento (CORNCOB, THE FEUD, etc. [BMMM98]). Como conclusión, podemos afirmar que el nivel de conocimiento es obligatorio y que el nivel de implementación es opcional.

B.2 Secciones Generales de EML

En este apartado describiremos las secciones generales que se incluyen dentro de la definición de EML y no pertenecen a ninguno de sus sub-DSLs.

B.2.1 Identificación

La identificación de la entidad es el primer paso necesario para meta-especificar un patrón. Estos datos se incluyen en el elemento raíz del documento EML (recordar que EML está basado en XML). La identificación es obligatoria y fundamental para poder distinguir una entidad de otra. Es utilizada por la interfaz de registro para crear la dirección asociada la entidad. Los parámetros de identificación se incluyen en la etiqueta "`entity`" que es la etiqueta raíz de las meta-especficaciones EML.

Parámetros:

Type ID Name [Version]

Type	Tipo de elemento de la entidad (patrón, principio, categoría, etc.). A continuación enumeramos los principales tipos de entidades creados en el sistema: • *Pattern (patron)* • *PatternLanguage (lenguaje de patrones)* • *PatternType (tipo de patrón)* • *Category* • *DesignPrinciple (principio de diseño)* • *Author (autor)* • *AuthorGroup (grupo de autores)* • *Refactoring (refactorización)* • *Source (fuente de información)* • *AbstractionLevel (nivel de abstracción)*
Id	Identificador de la entidad en el sistema
Name	Nombre amigable de la entidad
Version	Versión de la entidad ***(opcional)***

Ejemplos:

En el primer ejemplo se muestra la identificación del patrón FACTORY METHOD. En el segundo se muestra la identificación de la versión 2 del lenguaje de patrones *Evolving Frameworks*.

```
<entity
    type="pattern"
    id="FactoryMethod"
    name="Factory Method">
...
... Resto de la entidad omitida para mayor claridad
...
</entity>
```

```
<entity
    type="patternLanguage"
    id="EvolvingFrameworks"
    name="Evolving Frameworks"
    version="2">
...
... Resto de la entidad omitida para mayor claridad
...
</entity>
```

B.2.2 Etiquetas

Los patrones se anotan (decoran) con etiquetas que pueden ser utilizadas para búsquedas o categorización. Las *etiquetas* simplemente son literales que se asocian a la entidad mediante la palabra reservada `tag`.

Ejemplos:

En el ejemplo a continuación se presenta una parte de la sección `tags` de la meta-especificación EML del patrón ABSTRACT FACTORY.

```
<tags>
  <tag>Factory</tag>
  <tag>Object Creation</tag>
  <tag>Families</tag>
  <tag>Object Family</tag>
  <tag>GoF Pattern</tag>
...
... Resto de las etiquetas omitidas para mayor claridad
...
</tags>
```

B.2.3 Resumen

El resumen es una frase corta que resume a la entidad que esta siendo descripta. El *resumen* es un literal que se asocia a la entidad mediante la palabra reservada `summary`.

Ejemplo:

Sección `summary` de de la meta-especificación EML del patrón ABSTRACT FACTORY.

```
<summary>Provide an interface for creating families of related or dependent
objects without specifying their concrete classes</summary>
```

B.3 Relaciones (EML-RDL)

EML-RDL (*Entity Metaspecification Lenguaje – Relationship Definition Language*) es un sub-lenguaje de dominio específico[274] (DSL) de EML que permite describir relaciones entre entidades EML. Puede trabajar con información incompleta, permitiendo introducir relaciones entre entidades existentes y elementos no introducidos en el catálogo. A partir de estas relaciones se puede armar una "red" de conceptos que pueden ser navegados de diversas formas. Esta red puede utilizarse también para clasificación dinámica y para alimentar un motor de inferencias.

La sintaxis de EML-RDL es muy sencilla, permitiendo establecer en forma sencilla relaciones entre entidades. Dada una meta-especificación EML de una entidad, una relación de EML-RDL especifica el tipo de la entidad destino, su dirección en el sistema (§ 13.4.1.3) y el tipo de relación que vincula a la entidad destino con la entidad actual. Las relaciones se incluyen dentro de la meta-especificación de una entidad (recordar que una meta-especificación es auto-contenida § 12.2.3) y por lo tanto, el origen está implícito (es la entidad que está siendo definida).

B.3.1 Definición de Relaciones

La relaciones se describen mediante elementos de tipo `relationship` y se agrupan dentro de la etiqueta contenedora `context`. La entradas de tipo `relationship` de la sección `context` describen las relaciones entre la entidad que esta siendo descripta y otras entidades almacenadas (o no) en el catálogo[275]. Para describir una relación se incluye el tipo de entidad de destino (`targetType`),

[274] Recordar que EML es un DSL compuesto de varios sub-DSLs

[275] EML y el catálogo saben gestionar información incompleta. Por lo tanto, la entidad relacionada no debe estar necesariamente registrada en el catálogo.

su identificador (`targetId`) y el tipo de relación que tiene con la entidad origen (`type`).

Parámetros:

TargetType TargetID Type

targetType	Tipo de elemento de la entidad destino (patrón, principio, categoría, etc.)
targetID	Identificador de la entidad destino
type	Tipo de relación

Ejemplos:

En el primer ejemplo se muestra un fragmento de la definición en EML-RDL donde se describen las relaciones del patrón ABSTRACT FACTORY. En el segundo se muestra un fragmento del EML-RDL del grupo de autores "*Gang of Four*" (en este caso contiene relaciones con cada uno de sus miembros materializadas a partir de relaciones "*isMember*" con entidades de tipo "*Author*").

```
<context>
  <relationship
             targetType="Source"
             targetID="GoFBook"
             type="isPublishedIn"/>
  <relationship
             targetType="PatternLanguage"
             targetID="GoF"
             type="isContainedIn"/>
  <relationship
             targetType="Category"
             targetID="GoF.Creational"
             type="isContainedIn"/>
  <relationship
             targetType="PatternType"
             targetID="Design Pattern"
             type="isA"/>
  <relationship
             targetType="OOPrinciple"
             targetID="DIP"
             type="conforms"/>
...
... Resto de las relaciones omitidas para mayor claridad
...
</context>
```

```
<context>
   <relationship
             targetType="Author"
             targetID="Erich Gamma"
             type="isMember"/>
   <relationship
             targetType="Author"
             targetID="Richard Helm"
             type="isMember"/>
   <relationship
             targetType="Author"
             targetID="Ralph Johnson"
             type="isMember"/>
   <relationship
             targetType="Author"
             targetID="John Vlissides"
             type="isMember"/>
</context>
```

B.4 Propiedades (EML-PDL)

EML-PDL (*Entity Metaspecification Lenguaje – Property Definition Language*) es un sub-lenguaje de dominio específico[276] (DSL) de EML que permite describir plantillas de patrones a través de la composición dinámica de elementos atómicos sencillos a los cuales llamamos "propiedades", ofreciendo un marco de trabajo ágil, flexible y extensible [Welicki04] para describir entidades (patrones y conceptos) de ingeniería del software.

EML utiliza un sistema flexible de definición de plantillas que permite la combinación de elementos existentes en cualquiera de ellas[277]. Este sistema se basa en la composición dinámica de unidades atómicas sencillas de información, a las cuales llamamos "propiedades". Una plantilla es un conjunto de propiedades vinculadas. El lenguaje de definición de plantillas es capaz de tratar con cualquier agregación de elementos sencillos que siguen unas reglas sintácticas básicas[278] (en este punto, es necesario recordar que EML esta basado en XML). El lenguaje que da soporte a este sistema, permitiendo expresar las propiedades y sus relaciones es EML-PLD.

El elemento fundamental de expresión de EML-PDL es la propiedad. Una propiedad permite describir un aspecto concreto de una entidad. A partir de la composición de un conjunto discreto y finito de propiedades se consigue una plantilla (§ 4.1) que contiene la descripción de una entidad concreta. Esta composición puede modificarse dinámicamente en cualquier momento, estableciendo de esta forma un marco para la constante evolución y mejora de las entidades que describen (§ 11.6). En este contexto podemos decir que la descripción de una entidad es un fenómeno emergente (**capítulo 9**), que se

[276] Recordar que EML es un DSL compuesto de varios sub-DSLs
[277] Este sistema ha sido creado en esta tesis. Para más detalles ver el capítulo 12.
[278] Estas reglas son las de "xml bien formado" (well-formedness) [W3C]

produce a partir de la combinación de elementos simples especializados en la descripción de un aspecto local (las propiedades).

Existen dos tipos principales de propiedades que pueden ser utilizadas para describir a las entidades: las propiedades atómicas y las propiedades compuestas. La forma de interacción entre estos tipos de propiedades se basa en los principios expuestos en el patrón COMPOSITE [GoF95].

- Las **propiedades atómicas** se refieren a elementos sencillos que expresan un único concepto o tipo de dato. Ejemplos de este tipo de propiedades incluyen cadenas, fechas, relaciones, etc.

- Las **propiedades compuestas** permiten crear propiedades compuestas a su vez de otras. Las propiedades compuestas permiten crear propiedades complejas a partir de elementos sencillos. Por ejemplo, para describir los beneficios de aplicar un patrón podemos crear una propiedad compuesta llamada beneficios que se compone de propiedades de tipo beneficio. La propiedad beneficio es a su vez una propiedad compuesta que se compone de los elementos atómicos título y discusión.

A partir de la composición dinámica de propiedades se pueden crear elementos complejos sin una estructura rígida previamente definida. La composición dinámica basada en vinculación tardía provee un alto grado de flexibilidad y extensibilidad[279], permitiendo modificar la estructura de la plantilla de la entidad en tiempo de ejecución[280].

Es importante destacar que en EML-PDL no es necesario marcar a los elementos como `property`. Las propiedades se describen a partir de los nombres de sus instancias y es el analizador quien determina a partir de metadatos con qué tipo de propiedad esta tratando.

B.4.1 Propiedades Atómicas

En esta sección enumeraremos las propiedades atómicas existentes en EML 1.0.

B.4.1.1 String

Contiene cadenas de texto. Es el tipo de propiedad predeterminado. Si el analizador EML no reconoce el tipo de la propiedad que está analizando intentará almacenarla en una propiedad de este tipo. Por lo tanto, para crear una propiedad de este tipo no hace falta poner la palabra `string` en la etiqueta con la propiedad.

[279] Esta aserción se ve reforzada por el principio de diseño sobre el que se fundan todos los patrones de [GoF95] "utilizar composición en lugar de herencia". Aunque en este caso la frase se refiere a programación orientada a objetos la idea subyacente es muy general, siendo también aplicable a nuestro problema concreto.
[280] Esta característica puede ser encontrada también en los sistemas reflectivos [Ortin03]

Parámetros:

Name Value

name	Nombre de la propiedad
value	Valor de la propiedad

Ejemplos:

En el primer ejemplo se muestra la intención del patrón ABSTRACT FACTORY. En el segundo la definición del problema en el patrón SINGLETON.

```
<intent>Provide an interface for creating families of related or dependent
objects without specifying their concrete classes</intent>
```

```
<problem>Several different client objects need to refer to the same thing
and you want to ensure that you do not have more than one of them</problem>
```

B.4.1.2 Image

Contiene referencias a imágenes. Estas propiedades permiten asociar imágenes existentes (en cualquier formato, como por ejemplo gif, jpg, bmp, etc.) a las entidades EML. Las imágenes se suelen agrupar en contenedores de tipo images.

Parámetros:

Name Description Uri

Name	Nombre amigable de la imagen
Description	Descripción textual de la imagen
Uri	Dirección donde se encuentra almacenado el fichero de la imagen

Ejemplos:

El siguiente ejemplo muestra la especificación del diagrama OMT de la estructura del patrón ABSTRACT FACTORY.

```
<image name="StructureImageOMT">
  <description>Class diagram of the structure of the pattern in OMT
notation</description>
  <url>/images/GoF/OMT/abstractFactory.gif</url>
</image>
```

B.4.1.3 Consequence

Describe una consecuencia de la aplicación del patrón. La consecuencia puede ser un beneficio o un incordio. Incluye una breve frase que resume a la consecuencia y una discusión mas detallada.

Parámetros:

Type Title [Discussion]

type	Tipo de consecuencia. El valor de esta propiedad puede ser uno de estos dos: • `benefit` (beneficio) • `liability` (incordio)
title	Titulo de la consecuencia. Es una frase breve que la resume.
discusión	Discusión detallada sobre la consecuencia. Es un bloque de texto de extensión arbitraria. ***(opcional)***

Ejemplos:

El primer ejemplo muestra un beneficio de aplicación del patrón BRIDGE. El segundo un inconveniente de la aplicación del ABSTRACT Factory.

```
<consequence type="benefit">
  <title>Decoupling interface and implementation</title>
  <discussion>An implementation is not bound permanently to an interface.
The implementation of an abstraction can be configured at run-time. It's
even possible for an object to change its implementation at run-time.
Decoupling Abstraction and Implementor also eliminates compile-time
dependencies on the implementation. Changing an implementation class doesn't
require recompiling the Abstraction class and its clients. This property is
essential when you must ensure binary compatibility between different
versions of a class library. Furthermore, this decoupling encourages
layering that can lead to a better-structured system. The high-level part of
a system only has to know about Abstraction and Implementor.</discussion>
</consequence>
```

```
<consequence type="liability">
  <title>Supporting new kinds of products is difficult.</title>
  <discussion>Extending abstract factories to produce new kinds of Products
isn't easy. That's because the AbstractFactory interface fixes the set of
products that can be created. Supporting new kinds of products requires
extending the factory interface, which involves changing the AbstractFactory
class and all of its subclasses.</discussion>
</consequence>
```

B.4.1.4 Info

Describe fuentes de información externas donde puede encontrarse información adicional sobre una entidad.

Parámetros:

Name Description Uri Type

name	Nombre de la fuente de información
description	Descripción de la fuente de información
uri	Dirección de la fuente de información
type	Tipo de fuente de información: • `Book` (libro) • `Web` (referencia web)

Ejemplos:

```
<info>
  <name>Design Patterns: Elements of Reusable Object Oriented
Software</name>
  <description>Gang of Foor book on Amazon.com. This pattern is included in
the Structural Patterns section of that book, starting its description on
page 163</description>
  <url><![CDATA[http://www.amazon.com/exec/obidos/tg/detail/-
/0201633612/qid=1099925543/sr=8-1/ref=pd_csp_1/102-1422086-
1770553?v=glance&s=books&n=507846]]></url>
  <type>Book</type>
</info>
```

```
<info>
  <name>Composite Pattern on c2 Wiki</name>
  <description>Discussion on the Composite Pattern on C2 Wiki</description>
  <url>http://c2.com/cgi/wiki?CompositePattern</url>
  <type>Web</type>
</info>
```

B.4.1.5 RelatedPattern

Describe un patrón relacionado. Añade información adicional sobre la descripción de una relación establecida con EML-BDL.

Parámetros:

Id FriendlyName Relationship

Id	Identificador del patrón relacionado
FriendlyName	Nombre real del patrón relacionado
Relationship	Descripción textual del tipo de relación

Ejemplos:

Información adicional sobre la relación con el patrón SINGLETON en el patrón ABSTRACT FACTORY.

```
<relatedPattern>
  <friendlyName>Singleton</friendlyName>
  <fullName>GoF.Creational.Singleton</fullName>
  <relationship>A concrete factory is often a singleton.</relationship>
</relatedPattern>
```

B.4.1.6 Paragraph

Permite introducir un párrafo en un bloque de texto. Estos elementos pueden agruparse en cualquier propiedad donde se trate con textos.
Parámetros:

Id

Id	Identificador del párrafo

Ejemplo:

Definición de uno de los párrafos de la motivación del patrón ABSTRACT FACTORY.

```
<paragraph id="1">Consider a user interface toolkit that supports multiple
look-and-feel standards, such as Motif and Presentation Manager. Different
look-and-feels define different appearances and behaviors for user interface
"widgets" like scroll bars, windows, and buttons. To be portable across
look-and-feel standards, an application should not hard-code its widgets for
a particular look and feel. Instantiating look-and-feel-specific classes of
widgets throughout the application makes it hard to change the look and feel
later.</paragraph>
```

B.4.2 Propiedades Compuestas

En esta sección enumeraremos las propiedades compuestas existentes en EML 1.0.

B.4.2.1 CompositeProperty

Esta es la propiedad compuesta general. Cuando el analizador se encuentra ante una composición de primitivas y ésta no se corresponde con ninguno de los nombres reservados (`Consequences`, `MoreInfo`, `Images`, `RelatedPatterns`) almacena la composición en una propiedad de este tipo. Al igual que con `StringProperty` no es necesario marcar a la propiedad con nombre

`CompositeProperty`, dado que el analizador realiza la asignación a un contenedor de este tipo en forma automática.

B.4.2.2 Consequences

Contiene las consecuencias resultantes de aplicar un patrón (aunque esta sección es de utilidad también en conceptos de soporte como refactorizaciones y principios de orientación a objetos). Es un agrupador de propiedades atómicas de tipo `Consequence`.

Ejemplos:

Consecuencias de aplicar el patrón TEMPLATE METHOD.

```
<consequences>
  <consequence type="benefit">
     <title>Template methods are a fundamental technique for code reuse.
     </title>
     <discussion>They are particularly important in class libraries, because
they are the means for factoring out common behavior in library classes.
     </discussion>
  </consequence>
  <consequence type="benefit">
     <title>Inversion of Control.</title>
     <discussion>Template methods lead to an inverted control structure
that's sometimes referred to as "the Hollywood principle," that is, "Don't
call us, we'll call you" [Swe85]. This refers to how a parent class calls
the operations of a subclass and not the other way around.
     </discussion>
  </consequence>
</consequences>
```

B.4.2.3 MoreInfo

Contiene las referencias a sitios donde puede encontrarse más información sobre una entidad. Es un agrupador de propiedades atómicas de tipo `Info`.

Ejemplos:

Sitios donde puede encontrarse más información sobre el patrón ABSTRACT FACTORY.

```
<moreInfo>
  <info>
    <name>Design Patterns: Elements of Reusable Object Oriented
Software</name>
    <description>Gang of Foor book on Amazon.com. This pattern is included
in the Creational Patterns section of that book, starting its description on
page 87</description>
    <url><![CDATA[http://www.amazon.com/exec/obidos/tg/detail/-
/0201633612/qid=1099925543/sr=8-1/ref=pd_csp_1/102-1422086-
1770553?v=glance&s=books&n=507846]]></url>
    <source>GoFBook</source>
    <type>Book</type>
  </info>
  <info>
    <name>Abstract Factory at C2 Wiki</name>
    <description>Discussion on this pattern at C2 Wiki</description>
    <url>http://c2.com/cgi/wiki?AbstractFactoryPattern</url>
    <type>Web</type>
  </info>
...
... Resto de la sección omitida para mayor claridad
...
</moreInfo>
```

B.4.2.4 Images

Contiene las imágenes asociadas a una entidad. Es un agrupador de propiedades atómicas de tipo `Image`.

Ejemplo:

Imágenes asociadas al patrón Proxy.

```
<images>
  <image name="StructureImageOMT">
    <description>Class diagram of the structure of the pattern in OMT
notation</description>
    <url>/GoF/OMT/proxy.gif</url>
  </image>
  <image name="StructureImageUML">
    <description>Class diagram of the structure of the pattern in UML
notation</description>
    <url>/GoF/UML/proxy.gif</url>
  </image>
  <image name="NonSoftwareExample">
    <description>Diagram of the non-software example</description>
    <url>/GoF/Non-Software/proxy.gif</url>
  </image>
</images>
```

B.4.2.5 Related Patterns

Contiene información adicional sobre los patrones relacionados a una entidad. Es un agrupador de propiedades atómicas de tipo `RelatedPattern`.

Ejemplo:

Patrones relacionados con el patrón FAÇADE.

```
<relatedPatterns>

  <relatedPattern>
    <friendlyName>Abstract Factory</friendlyName>
    <fullName>GoF.Creational.AbstractFactory</fullName>
    <relationship>Abstract Factory can be used with Facade to provide an
interface for creating subsystem objects in a subsystem-independent way.
Abstract Factory can also be used as an alternative to Facade to hide
platform-specific classes.</relationship>
  </relatedPattern>

  <relatedPattern>
    <friendlyName>Mediator</friendlyName>
    <fullName>GoF.Behavioral.Mediator</fullName>
    <relationship>Mediator is similar to Facade in that it abstracts
functionality of existing classes. However, Mediator's purpose is to
abstract arbitrary communication between colleague objects, often
centralizing functionality that doesn't belong in any one of them. A
mediator's colleagues are aware of and communicate with the mediator instead
of communicating with each other directly. In contrast, a facade merely
abstracts the interface to subsystem objects to make them easier to use; it
doesn't define new functionality, and subsystem classes don't know about
it.</relationship>
  </relatedPattern>

  <relatedPattern>
    <friendlyName>Singleton</friendlyName>
    <fullName>GoF.Creational.Singleton</fullName>
    <relationship>Usually only one Facade object is required. Thus Facade
objects are often Singletons.</relationship>
  </relatedPattern>
</relatedPatterns>
```

B.5 Estructura (EML-SDL)

EML-SDL (*Entity Meta-specification Language – Structure Description Language*) permite describir la estructura de la solución en un patrón en términos de grupos de entidades relacionadas. Una entidad EML contiene un conjunto de participantes que se relacionan entre si. Estos participantes junto con sus relaciones conforman la estructura del nivel de implementación.

La descripción de la estructura se realiza en tres pasos sencillos y muy esquemáticos:

4. **Definición de los participantes**: se definen los participantes en la solución al problema. Para cada participante se define el rol que ocupa, la cardinalidad y si es o no abstracto. Los participantes se refieren en forma general a entidades que pueden ser traducidas a artefactos de software (por ejemplo, clases), aunque no están limitados a esto (recordar que la semántica de utilización la pone el usuario de la meta-especificación).

5. **Definición de las relaciones entre los participantes**: una vez creados los participantes se establecen las relaciones entre ellos. Las relaciones pueden ser de creación, asociación, composición y herencia.

6. **Definición de las responsabilidades de los participantes**: descripción de las responsabilidades de los participantes. Esta descripción se realiza en prosa (es la única parte de la descripción de la estructura que no es formal).

Este modelo de descripción ha sido diseñado teniendo en cuenta el orden en que se realiza la traducción de un patrón existente a EML, a efectos de facilitar la producción de entidades EML (y codificación de patrones existentes).

B.5.1 Definición de la Estructura

La estructura del patrón se representa en la sección "`structure`" de la meta-especificación. En esta sección, existen tres partes:

- **Participantes**: lista de los roles participantes. Para cada uno se especifica su nombre, un indicador de si es o no abstracto y su cardinalidad.

- **Relaciones**: relaciones entre los roles.

- **Responsabilidades**: lista de responsabilidades de cada uno de los roles.

B.5.1.1 Participantes

El objetivo de esta sección es definir los roles que participan en la solución al problema que resuelve el patrón.

Ejemplos:

```
<participants>
  <participant
        role="roleName"
        isAbstract="true|false"
        cardinality="1|0,1|*|+"/>
  ...
</participants>
```

B.5.1.2 Relaciones

El objetivo de esta sección es describir las relaciones entre los roles participantes. El tipo de relación puede ser uno de los siguientes:

- **`Inheritance`**: relación de herencia
- **`Composition`**: relación de composición
- **`Aggregation`**: relación de agregación
- **`Association`**: relación de asociación
- **`Creation`**: relación de asociación

Ejemplos:

```
<relationships>

  <relationship
      type="inheritance"
      child="childRole"
      parent="parentRole"/>

  <relationship
      type="composition|aggregation|association|creation"
      source="sourceRole"
      target="targeRole"
      name="subject"
      cardinality="1|0,1|*|+"/>

</relationships>
```

B.5.1.3 Responsabilidades

El objetivo de esta sección es describir las responsabilidades de cada rol. Esta sección es opcional.

Ejemplos:

```
<responsibilities>

  <role name="Nombre del Rol">
    <responsibility>[ Descripción de la Responsabilidad ]</responsibility>
  </role>

</responsibilities>
```

B.5.1.4 Ejemplo

En esta sección presentaremos algunos ejemplos de la definición de la estructura.

B.5.1.4.1 Sección Vacía

El siguiente fragmento EML contiene la definición vacía que de la estructura de la solución de una entidad.

```
<structure>
  <participants>
    <participant role="roleName" isAbstract="true|false"
                            cardinality="1|0,1|*|+"/>
                ...
  </participants>

  <relationships>
    <relationship
                  type="inheritance"
                  child="childRole"
                  parent="parentRole"/>
    <relationship
                  type="composition|aggregation|association"
                  source="sourceRole" target="targetRole"
                  name="name" cardinality="1|0,1|*|+"/>
    <relationship type="creation" source="source" target="target"/>
                ...
  </relationships>

  <responsibilities>
    <role name="roleName">
      <responsibility>Describe relationship</responsibility>
         ...
     </role>
     ...
  </responsibilities>
</structure>
```

B.5.1.4.2 Ejemplo: Estructura del Patrón Composite

El siguiente ejemplo presenta la meta-especificación EML de la estructura del patrón COMPOSITE.

```
<structure>
   <participants>
       <participant role="Component" isAbstract="true"   cardinality="1"/>
       <participant role="Leaf"      isAbstract="false"  cardinality="+"/>
       <participant role="Composite" isAbstract="false"  cardinality="+"/>
       <participant role="Client"    isAbstract="false"  cardinality="0,1"/>
   </participants>

   <relationships>
      <relationship type="inheritance"
                    child="Leaf" parent="Component"/>
      <relationship type="inheritance"
                    child="Composite" parent="Component"/>
      <relationship type="composition" source="Composite" target="Component"
                    name="children" cardinality="*"/>
      <relationship type="association" source="Client" target="Component"
                    name="component" cardinality="1"/>
  </relationships>

   <responsibilities>
      <role name="Component">
         <responsibility>Declares the interface for objects in the
composition.</responsibility>
         <responsibility>Implements default behavior for the interface
common to all classes, as appropriate.</responsibility>
         <responsibility>Declares an interface for accessing and managing
its child components.</responsibility>
         <responsibility>(optional) Defines an interface for accessing a
component's parent in the recursive structure, and implements it if that's
appropriate.</responsibility>
      </role>

      <role name="Leaf">
         <responsibility>Represents leaf objects in the composition. A leaf
has no children.</responsibility>
         <responsibility>Defines behaviour for primitive objects in the
composition.</responsibility>
      </role>

      <role name="Composite">
         <responsibility>Defines behaviour for components having children.
         </responsibility>
         <responsibility>Stores child components.</responsibility>
         <responsibility>Implements child-related operations in the
Component interface.</responsibility>
      </role>

      <role name="Client">
            <responsibility>Manipulates objects in the composition through
the Component interface.
            </responsibility>
      </role>
   </responsibilities>
</structure>
```

B.6 Comportamiento (EML-BDL)

EML-BDL permite describir en forma abstracta el comportamiento de los participantes que han sido declarados previamente mediante EML-SDL (§ 12.2.4.4).

El comportamiento se describe mediante un conjunto de primitivas generales que se incluyen en esta sección. Estas primitivas son abstracciones básicas presentes en múltiples lenguajes de programación (manipulación de variables, control de flujo, bucles, invocación, etc.) y se pueden combinar para expresar algoritmos.

Los algoritmos se incluyen en métodos (que son conjuntos de primitivas), para los cuales se puede especificar una firma (parámetros de entrada y salida, tipo de retorno, si es abstracto, si es virtual, si es final o la cardinalidad). Esta descripción de algoritmos puede ser modificada y adaptada en tiempo de ejecución.

EML-BDL permite describir propiedades (junto con sus modificadores y tipos) y conjuntos de métodos (un método tiene una firma y agrupa un conjunto de primitivas atómicas que definen un algoritmo).

EML-BDL describe el comportamiento de una entidad (mediante algoritmos y propiedades) a un alto nivel de abstracción, pudiendo ser traducido posteriormente (junto con EML-SDL, § 12.2.4.4) a cualquier lenguaje de programación de alto nivel (§ 13.4.10).

EML-BDL no pretende ser un lenguaje de programación ejecutable, sino describir el comportamiento de entidades a un alto nivel de abstracción.

B.6.1 Descripción del Comportamiento

En esta sección se describe la forma de especificar la implementación del patrón. El comportamiento se describe en un contenedor de tipo `implementation`. En esta sección se define el comportamiento de los participantes a partir de las siguientes secciones:

- **Properties**: propiedades de la clase
- **Constructors**: constructores
- **Behaviour**: comportamiento

Adicionalmente, la implementación incluye dos propiedades descriptivas opcionales: el nombre (`friendlyName`) y una descripción (`description`). Estas propiedades se almacenan en el contenedor `implementationInfo`.

Ejemplo:

Esquema de definición de comportamiento en EML-BDL.

```
<implementation>
    <implementationInfo>
        <friendlyName>...</friendlyName>
        <description>...</description>
    </implementationInfo>

    <participant role="roleName">
       <properties/>
       <constructors/>
       <behaviour/>
    </participant>

</implementation>
```

B.6.1.1 Propiedades

Las propiedades de una clase se especifican mediante la palabra "member".

Scope Name Type [hasGetter] [hasSetter] [isClass]

Scope	Ambito del miembro
Name	Nombre del miembro
Type	Tipo del miembro
hasGetter	Si es su valor es **verdadero**, se incluye un getter público. Si no se incluye, se asume que su valor es falso ***(opcional)***
hasSetter	Si es su valor es **verdadero**, se incluye un setter público. Si no se incluye, se asume que su valor es falso ***(opcional)***
isClass	Si es su valor es **verdadero**, indica que el miembro es estático (de clase). Si no se incluye, se asume que su valor es falso ***(opcional)***

Ejemplos:

```
EML:
<member scope="private" name="name" type="string" hasGetter="true"/>

C#:
private string name;

public string Name
{
    get { return this.name; }
}

Java:
private String name;

public String getName()
{
    return this.name;
}
```

```
EML:
<member scope="protected" name="instance" type="Singleton" isClass="true"/>

C#, Java:
protected static Singleton instance;
```

B.6.1.2 Constructores

Los constructores se describen mediante dos secciones: `signature` (que describe la firma del constructor) e `implementation` (que contiene el comportamiento del constructor).

B.6.1.2.1 Signature

Firma del constructor. Incluye el ámbito, si es abstracto, la cardinalidad y los parámetros que recibe el método.

Parámetros

*Scope [cardinality] [parameter]**

Scope	Ambito del método Valores: public, protected, private
Cardinality	Cardinalidad del método ***(opcional)*** Valores: 1; 0,1; *; +
Parameters	Parámetros que recibe el método. Se especifican como una lista de elementos ***parameter***, en el cual se especifica el nombre y el tipo del parámetro. ***(opcional)***

Ejemplos:

```
EML:
<signature scope="public" cardinality="*"/>

C#, Java:
public NombreClase()
```

```
EML:
<signature scope="public">
  <parameters>
    <parameter name="component" type="Component"/>
  </parameters>
</signature>

C#, Java:
public NombreClase(Component component)
```

B.6.1.2.2 Implementation

Implementación del constructor utilizando las primitivas enumeradas en § B.7.

B.6.1.3 Métodos

Los métodos especifican dentro de la sección `Behaviour`. Cada método se especifica mediante un elemento `method` en dos secciones bien definidas:

- **Signature**: firma del método
- **Implementation**: implementación del método, utilizando el metalenguaje definido en la sección 6.

B.6.1.3.1 Signature

Signatura del método. Incluye el ámbito, el tipo de retorno, si es abstracto, la cardinalidad y los parámetros que recibe el método.

Parámetros:

*Scope isAbstract [isClass] [returns] [cardinality] [parameter]**

Scope	Ambito del método `Valores: public, protected, private`
isAbstract	Indica si el método es abstracto. `Valores: true, false`
isClass	Indica si el método es estático ***(opcional)*** `Valores: true, false`
Returns	Tipo de retorno del método ***(opcional)*** Valores: cualquier tipo de dato
Cardinality	Cardinalidad del método ***(opcional)*** Valores: 1; 0,1; *; +
Parameters	Parámetros que recibe el método. Se especifican como una lista de elementos ***parameter***, en el cual se especifica el nombre y el tipo del parámetro. ***(opcional)***

Ejemplos:

```
EML:
<signature scope="public" isAbstract="false" cardinality="*"/>

C#, Java:
public void NombreMetodo()
```

```
EML:
<signature scope="public" isAbstract="true" returns="MyReturnObject">
  <parameters>
    <parameter name="component" type="Component"/>
  </parameters>
</signature>

C#, Java:
abstract public MyReturnObject NombreMetodo(Component component)
```

B.6.1.3.1 Implementation

Implementación del método utilizando las primitivas enumeradas en § B.7.

B.7 Construcciones del Lenguaje de Descripción del Comportamiento

En esta sección se incluyen todas las cláusulas del metalenguaje definidas a efectos de representar el código fuente en forma abstracta. En la primer parte de esta sección se presenta la lista de comandos. En la sección siguiente se presenta la especificación en mayor detalle de cada uno de ellos. Finalmente se dan ejemplos de composiciones de estos comandos primitivos para obtener comportamientos más complejos.

Es importante recordar que EML-BDL no pretende ser un lenguaje ejecutable, sino describir el comportamiento de entidades.

B.7.1 Lista de comandos

Comando	Descripción
`addToList`	Añadir un elemento a una lista.
`arithmetic`	Funciones aritméticas básicas (sumas, restas, etc.).
`blankLine`	Agrega una línea en blanco a un bloque de código, a efectos de mejorar el layout del código resultante.
`comment`	Introduce un comentario en un bloque de código.
`concat`	Concatenación de cadenas de texto.
`createInstance`	Crea una instancia de una clase. A diferencia de la creación con el declare, el objeto contenedor de la instancia ya existe previamente, aunque no ha sido inicializado.
`declare`	Crea una variable y eventualmente la inicializa.
`foreach`	Recorre todos los elementos de un tipo en una colección.
`get`	Obtiene el valor de un elemento.
`getFromList`	Obtener un elemento a una lista.
`if`	Especificación de una bifurcación condicional, típicamente expresada mediante la sentencia ***if..then*** en los lenguajes imperativos.
`invoke`	Invoca un método de una entidad (objeto o clase).
`locked`	Bloqueo (tipo mutex) sobre un bloque de código.
`optional`	Indica un bloque de código opcional. El código que se incluye dentro de esta etiqueta puede ser excluido del resultado.
`print`	Imprime un mensaje a un dispositivo de salida.
`removeFromList`	Eliminar un elemento a una lista.
`return`	Retorna un valor al terminar la ejecución de un método.
`set`	Establece el valor de un elemento, variable o miembro de una clase.
`throw`	Dispara una excepción. La excepción a disparar puede ser una nueva excepción (creada directamente en la sentencia *throw*) o una excepción existente (obtenida, por ejemplo, en la cláusula *catch*).

B.7.2 Especificación de los comandos

A continuación se presenta de definición de cada uno de los comandos enumerados en la tabla anterior. En cada caso se presenta una frase que resume el objetivo del comando, los parámetros de entrada y ejemplos (si son aplicables).

Es importante recordar nuevamente que EML-BDL no pretende ser un lenguaje de programación ejecutable, sino un DSL que permite describir la esencia del comportamiento de las entidades a un alto nivel de abstracción.

B.7.2.1 addToList

Añadir un elemento a una lista.

Parámetros:

listMemberName value

listMemberName	Nombre de la lista
value	Valor a añadir en la lista

Ejemplo:

```
EML:
<addToList listMemberName="children" value="param:component"/>

C#, Java:
theList.Add(component);
```

B.7.2.2 arithmetic

Funciones aritméticas básicas (suma, resta, multiplicación y división)

Parámetros:

operation

operation	Operación a realizar. Valores: • `add` (suma) • `subtract` (resta) • `multiply` (multiplicación) • `divide` (división)

Ejemplos:

```
EML:
<arithmetic>
  <operand value="1 type="double"/>
  <operand operation="add" value="2" type="double">
</arithmetic>

C#, Java:
1 + 2
```

```
EML:
<arithmetic>
  <operand value="0.16" type="double"/>
  <operand operation="multiply"><invoke member="beverage" method="cost"
isTerminal="false"/></operand>
</arithmetic>

C#:
0.16 * beverage.Cost()

Java:
0.16 * beverage.cost()
```

B.7.2.3 blankLine

Agrega una línea en blanco a un bloque de código, a efectos de mejorar el layout del código resultante.

Ejemplo:

```
EML:
<blankLine/>
```

B.7.2.4 comment

Introduce un comentario en un bloque de código.

Ejemplo:

```
EML:
<comment>Este es un comentario</comment>

C#, Java:
// Este es un comentario

VB.NET:
' Este es un comentario
```

B.7.2.5 concat

Concatenación de elementos. Los elementos a concatenar se expresan en elementos `item` dentro del nodo `concat`.

Parámetros:

value [type]

value	Valor a concatenar
type	Tipo del valor a concatenar ***(opcional)***

Ejemplos:

```
EML:
<concat>
  <item type="string" value="Hello"/>
  <item type="string" value=" World"/>
</concat>

C#, Java:
"Hello World"
```

B.7.2.6 createInstance

Crea una instancia de una clase. A diferencia de la creación con el declare, el objeto contenedor de la instancia ya existe previamente, aunque no ha sido inicializado.

Parámetros:

*(member | variable) [type] [parameter]**

member	Nombre del miembro
variable	Nombre de la variable
type	Tipo de la instancia (por ejemplo, al crear una clase concreta a partir de una base abstracta) ***(opcional)***
parameters	Parámetros del constructor (0 a n) ***(opcional)***

Ejemplos:

```
EML:
<createInstance variable="anImplementor" type="ConcreteImplementorB"/>

C#, Java:
anImplementor = new ConcreteImplementorB();
```

```
EML:
<createInstance type="leaf">
  <parameters>
    <parameter name="name" value="Leaf 1" type="string"/>
  </parameters>
</createInstance>

C#, Java:
new Leaf("Leaf 1");
```

B.7.2.7 declare

Crea una variable y eventualmente la inicializa.

Parámetros:

name type ([instanceType] [parameter])*

name	Nombre de la variable
type	Tipo de la variable
instanceType	Tipo de la instancia (por ejemplo, al crear una clase concreta a partir de una base abstracta) ***(opcional)***
parameters	Parámetros del constructor ***(opcional)***

Ejemplos:

```
EML:
<declare name="product" type="Product"/>

C#, Java:
Product product;
```

```
EML:
<declare name="abstraction"
         type="Abstraction"
         instanceType="RefinedAbstraction"/>

C#, Java:
Abstraction abstraction = new RefinedAbstraction();
```

```
EML:
<declare name="root" type="Component" instanceType="Composite">
  <parameters>
    <parameter name="name" value="Tree Root" type="string"/>
  </parameters>
</declare>

C#, Java:
Component root = new Composite("Tree Root");
```

B.7.2.8 foreach

Recorre todos los elementos de un tipo en una colección.

Parámetros:

type in observers

type	Tipo de elemento sobre el que se realiza la iteración
in	Lista contenedora de los elementos
into	Variable adonde ubicar los elementos obtenidos de la lista

Ejemplos:

```
EML:
<foreach type="Observer" in="observers" into="observer">
  ...
</foreach>

C#:
foreach(Observer observer in observers)
{
    ...
}
```

```
EML:
<foreach type="Observer" in="observers" into="observer">
  <invoke variable="observer" method="update" param="this"/>
</foreach>

C#:
foreach(Observer observer in observers)
{
    observer.Update(this);
}
```

B.7.2.9 get

Obtiene el valor de un elemento.

Parámetros:

[container] (member | variable | name) into

container	Contenedor donde se aloja el miembro a invocar ***(opcional)***
member	Miembro de la clase actual. En lenguajes como Java y C# se agrega this. antes del nombre (por ejemplo, this.nombreMiembro) siempre que no sea estático
variable	Nombre de la variable
name	Nombre del elemento
into	Destinatario del valor del elemento

Ejemplo:

```
EML:
<get member="name" into="myName"/>


C#, Java:
myName = this.name;
```

```
EML:
<get container="person" member="name" into="userName"/>


C#:
userName= persona.Name;

Java:
userName= persona.getName();
```

B.7.2.10 getFromList

Recuperar un elemento a una lista

Parámetros:

listMemberName key into

listMemberName	Nombre de la lista
Key	Clave del elemento
Into	Destinatario del dato que se obtendrá de la lista

Ejemplo:

```
EML:
<getFromList listMemberName="children" key="param:id" into="#return"/>

C#:
return children[id];

Java:
return children.getItem(id);
```

B.7.2.11 if

Especificación de una bifurcación condicional, típicamente expresada mediante la sentencia ***if..then*** en los lenguajes imperativos.

La representación de esta sentencia se divide en tres partes, a saber:

- La condición
- Bloque de código a ejecutar cuando la condición es verdadera
- Bloque de código a ejecutar cuando la condición es falsa

```
EML:
<if>
  <condition/>
  <truePart/>
  <falsePart/>
</if>
```

Nota: Las secciones *truePart* y *falsePart* son bloques contenedores de otras instrucciones EML.

B.7.2.11.1 Condiciones

En las siguientes secciones se presenta la definción de las condiciones dentro de una cláusula `if` en EML-BDL. En todos los casos se incluye primero la lista de parámetros de entrara y luego un ejemplo.

B.7.2.11.1.1 IsNull

Parámetros:

Member | Variable

Ejemplo:

```
EML:
<if>
  <condition>
    <isNull variable="instance"/>
  </condition>
  <truePart>
    <createInstance variable="instance" type="Singleton"/>
  </truePart>
</if>

C#, Java:
if (instance == null)
    instance = new Singleton();
```

B.7.2.11.1.2 IsNotNull

Parámetros:

Member | Variable

Ejemplo:

```
EML:
<if>
  <condition>
    <isNotNull variable="component"/>
  </condition>
  <truePart>
    <invoke variable="component" method="operation"/>
  </truePart>
</if>

C#, Java:
if (component != null)
    component.Operation();
```

B.7.2.11.1.3 IsTrue

Parámetros:

Member | Variable

Ejemplo:

```
EML:
<if>
  <condition>
    <isTrue variable="flag"/>
  </condition>
  <truePart>
    <print message="Flag is True" device="console"/>
  </truePart>
</if>

C#:
if (flag)
    Console.Out.WriteLine(“Flag is True”);
```

B.7.2.11.1.4 IsFalse

Parámetros:

Member | Variable

Ejemplo:

```
EML:
<if>
  <condition>
    <isFalse variable="flag"/>
  </condition>
  <truePart>
    <print message="Flag is false" device="console"/>
  </truePart>
</if>

C#:
if (!flag)
    Console.Out.WriteLine("Flag is false");
```

B.7.2.11.1.5 IsEqual

Parámetros:

(LeftMember | LeftVariable) (RightMember | RightVariable | RightStringValue | RightNumValue)

Ejemplo:

```
EML:
<if>
  <condition>
    <isEqual leftVariable="dia" rightStringValue="Jueves"/>
  </condition>
  <truePart>
    <print message="Hoy es Jueves" device="console"/>
  </truePart>
</if>

C#:
if (dia == "Jueves")
    Console.Out.WriteLine("Hoy es Jueves");
```

B.7.2.11.1.6 IsNotEqual

Parámetros:

(LeftMember | LeftVariable) (RightMember | RightVariable | RightStringValue | RightNumValue)

Ejemplo:

```
EML:
<if>
  <condition>
    <isNotEqual leftVariable="dia" rightStringValue="Jueves"/>
  </condition>
  <truePart>
    <print message="Hoy no es Jueves" device="console"/>
  </truePart>
</if>

C#:
if (dia!=”Jueves”)
    Console.Out.WriteLine("Hoy no es Jueves");
```

B.7.2.12 invoke

Invoca un método de una entidad (objeto o clase).

Nota: Este comando permite la agregación de otros comandos. En uno de los ejemplos que se incluyen a continuación (el último), el comando `invoke` incluye la creación de una instancia de una clase, cuyo constructor a su vez recibe parámetros.

Parámetros:

*(member | variable) (method | methodType) [parameters]**

member	Nombre del miembro que contiene la implementación del método
variable	Nombre de la variable que contiene la implementación del método
method	Nombre del método a invocar
methodType	Tipo de método a invocar. Por ejemplo, si un método tiene cardinalidad n, al elegir su method type se invocan todas las instancias de los derivados de este método
parameters	Lista de parámetros del método ***(opcional)***

Ejemplos:

```
EML:
<invoke variable="root" method="operation"/>

C#:
root.Operation();

Java:
root.operation();
```

```
EML:
<invoke variable="root" method="add">
  <parameters>
    <parameter name="comp" value="component"/>
  </parameters>
</invoke>

C#:
root.Add(comp);

Java:
root.add(comp);
```

```
EML:
<invoke variable="comp" method="add">
  <createInstance type="leaf">
    <parameters>
      <parameter name="name" value="Leaf X.2" type="string"/>
    </parameters>
  </createInstance>
</invoke>

C#:
comp.Add(new Leaf("Leaf X.2"));

Java:
comp.add(new Leaf("Leaf X.2"));
```

B.7.2.13 locked

Bloqueo (tipo mutex) sobre un bloque de código.

Parámetros:

[on]

on	Objeto sobre el que se realiza el bloqueo

Ejemplo:

```
EML:
<locked on="padlock">
...
... Resto del código omitido para mayor claridad
...
</locked>

C#:
lock(padlock)
{
  ...
}
```

B.7.2.14 optional

Indica un bloque de código opcional. El código que se incluye dentro de esta etiqueta puede ser excluido del resultado.

value	Indica si el bloque de código que contiene en su interior será incluido o excluido de los resultados (`true`, incluido; `false`, excluido) ***(opcional)***

B.7.2.15 print

Imprime un mensaje a un dispositivo de salida.

Parámetros:

(message | value) [device]

message	Tipo a buscar
value	Lista contenedora de los elementos
device	dispositivo de salida (`console`, `debug`, `trace`) ***(opcional)***

Ejemplo:

```
EML:
<print message="Hello World!"/>

C#:
Console.Out.WriteLine(“Hello World!”);
```

B.7.2.16 removeFromList

Quita un elemento de una lista.
Parámetros:

listMemberName value

listMemberName	Nombre de la lista
value	Elemento a eliminar de la lista

Ejemplo:

```
EML:
<removeFromList listMemberName="children" value="param:component"/>

C#, Java:
theList.Remove(component);
```

B.7.2.17 return

Retorna un valor al terminar la ejecución de un método.

Parámetros:

member | *variable* | *value*

Member	Miembro de la clase a retornar
Variable	Nombre de la variable a retornar
Value	Valor a retornar

Ejemplos:

```
EML:
<return value="true"/>

C#, Java:
return true;
```

```
EML:
<return variable="instance"/>

C#, Java:
return instance;
```

```
EML:
<return member="name"/>

C#, Java:
return this.name;
```

B.7.2.18 set

Establece el valor de un elemento, variable o miembro de una clase.

Parámeters:

[container] (member | variable | name) value type

container	Contenedor del miembro al que se le desea asignar la propiedad ***(opcional)***
member	Miembro de la clase actual. En lenguajes como Java y C# se agrega `this.` antes del nombre (por ejemplo, `this.nombreMiembro`) siempre que no sea estático.
variable	Nombre de la variable
name	Nombre del elemento
value	Valor a asignar al elemento
type	Tipo del valor a asignar al elemento

Ejemplos:

```
EML:
<set member="name" value="param:theName"/>

C#, Java:
this.name = theName;
```

```
EML:
<set variable="apellido" value="Perez" type="string"/>

C#, Java:
apellido = "Perez";
```

```
EML:
<set container="myContainer" member="myMethod" value="Perez"/>

C#:
MyContainer.MyMethod();

Java:
myContainer.myMethod();
```

B.7.2.19 throw

Dispara una excepción. La excepción a disparar puede ser nueva (creada directamente en la sentencia *throw*) o existente (obtenida, por ejemplo, en la cláusula *catch*).

Parámetros:

(type [message] [innerException]) | exception

type	Tipo de la excepción
message	Mensaje de la excepción ***(opcional)***
innerException	Excepción interna ***(opcional)***
exception	Excepción que se desea disparar

Ejemplos:

```
EML:
<throwException type="Exception" message="Add is not supported"/>

C#, Java:
throw new Exception("Add is not supported");
```

```
EML:
<throwException type="Exception" message="Add is not supported"
innerException="ex"/>

C#, Java:
throw new Exception("Add is not supported", ex);
```

```
EML:
<throwException exception="myExceptionInstance"/>

C#, Java:
throw myExceptionInstance;
```

B.7.3 Algunos Ejemplos de Composición de Comandos

Las primitivas de EML-BDL pueden combinarse para expresar comportamientos complejos. En esta sección presentaremos algunos ejemplos de este tipo de forma de expresión a efectos de mostrar al lector cómo expresar comandos compuestos. En cada caso presentaremos el código EML y su correspondiente traducción a C#.

```
EML:
<return>
  <arithmetic>
     <operand value="0.32" type="double"/>
      <operand operation="add">
           <invoke member="beverage" method="cost" isTerminal="false"/>
      </operand>
  </arithmetic>
</return>

C#:
return 0.32 + beverage.Cost();
```

```
EML:
<return>
  <concat>
    <item>
          <invoke container="beverage" method="getName" isTerminal="false"/>
     </item>
     <item value=", Mocha" type="string"/>
  </concat>
</return>

C#:
return beverage.GetDescription() + ", Mocha"
```

```
EML:
<print device="console">
    <get container="originator" member="state"/>
</print>

C#:
Console.WriteLine(originator.State);
```

```
EML:
<if>
<condition>
  <isNull variable="instance"/>
</condition>
<truePart>
  <locked on="padlock">
    <if>
      <condition>
        <isNull variable="instance"/>
      </condition>
      <truePart>
        <createInstance variable="instance" type="SingletonMulti"/>
      </truePart>
    </if>
  </locked>
</truePart>
</if>

C#:
if (instance == null)
{
    lock(padlock)
    {
        if (instance == null)
        {
             instance = new SingletonMulti();
        }
    }
}
```

```
EML:
<invoke variable="comp" method="add">
  <createInstance type="leaf">
    <parameters>
      <parameter name="name" value="Leaf X.2" type="string"/>
    </parameters>
  </createInstance>
</invoke>

C#:
comp.Add(new Leaf("Leaf X.2"));

Java:
comp.add(new Leaf("Leaf X.2"));
```

B.7.4 Implementación Base e Implementaciones Concretas

EML-BDL permite definir una implementación base prototípica y partir de ésta diversas implementaciones concretas. De esta forma, se pueden dar distintas alternativas de implementación para una misma solución. De esta forma se puede determinar una estrategia general de solución, describirla en términos abstractos como implementación base y a partir de esta crear múltiples implementaciones concretas que redefinan aspectos particularmente relevantes.

B.7.4.1 Implementación Base

La implementación base es la que contiene la descripción general del comportamiento de la entidad y sirve como punto de partida para las implementaciones concretas, que redefinen este comportamiento. La implementación concreta se define dentro de un contenedor de tipo `implementation` utilizando las construcciones que hemos visto en las secciones anteriores.

En el siguiente ejemplo se muestra una implementación base muy sencilla que corresponde al patrón SINGLETON.

```
<implementation>
  <participant role="Singleton">
    <properties>
      <member scope="private"
                name="instance" type="Singleton" isClass="true"/>
    </properties>
    <constructors>
      <constructor>
        <signature scope="private"/>
      </constructor>
    </constructors>
    <behaviour>
      <method name="getInstance">
        <signature scope="public" returns="Singleton"
                    isClass="true" isAbstract="false"/>
        <implementation>
          <if>
            <condition>
              <isNull variable="instance"/>
            </condition>
            <truePart>
              <createInstance variable="instance" type="Singleton"/>
            </truePart>
          </if>
          <return variable="instance"/>
        </implementation>
      </method>
    </behaviour>
  </participant>
</implementation>
```

B.7.4.2 Implementaciones Concretas

Las implementaciones concretas redefinen la implementación base a efectos de ofrecer ejemplos concretos de implementaciones de la entidad ajustados a casos más concretos. Una entidad tiene una implementación base, pero puede tener muchas implementaciones concretas.

B.7.4.2.1 concreteImplementations

Las implementaciones concretas se definen dentro del contenedor `concreteImplemenations`. Este contenedor agrupa a todas las implementaciones concretas del patrón. Cada implementación se describe en un contenedor de tipo `implementation`. La instancia de `implementation` que está al mismo nivel que `concreteImplementations` en EML-BDL es la que contiene a la implementación base.

Ejemplo:

```
<implementation>
...
... Implementación omitida para mayor claridad
...
<implementation>

<concreteImplementations>
    <implementation name="Basic Sample">
    ...
    ... Implementación omitida para mayor claridad
    ...
    </implementation>
</concreteImplementations>
```

B.7.4.2.2 instanceOf

La palabra reservada `instanceOf` permite crear meta-especificaciones de instancias de clases o redefinir clases en una instancia concreta. Dentro de una definición `instanceOf` pueden redefinirse métodos de la definición original, inhibirlos (es decir, quitarlos) o añadir nuevos. Para añadir nuevos métodos sólo hay que incluirlos en la instancia. Para redefinirlos se utiliza la propiedad reservada `is` (explicada en la siguiente sección).

B.7.4.2.3 Redefiniendo propiedades y métodos

Existe una propiedad especial que puede ser utilizada en las firmas de las propiedades y métodos llamada `is`. Esta palabra permite indicar que una implementación de un método reemplaza a otro (el que se refiere en la propiedad `is`) en el comportamiento derivado. En el ejemplo que se muestra en la siguiente sección puede observarse como se redefine el método para obtener la instancia en el patrón SINGLETON en una de sus implementaciones concretas.

B.7.4.2.4 Ejemplo: implementaciones concretas del patrón Singleton

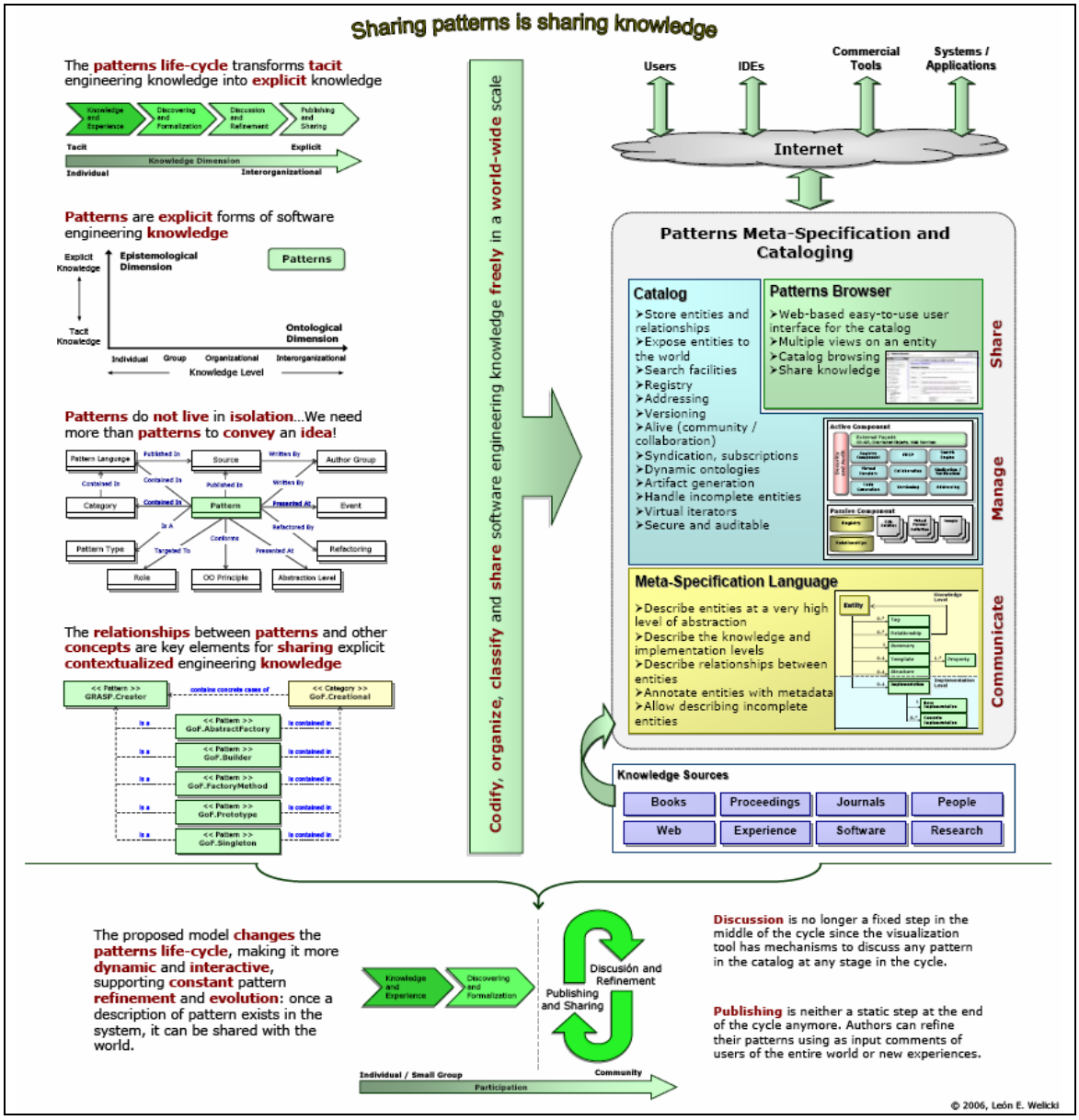
En el siguiente bloque de código se muestra la meta-especificación EML-BDL de las implementaciones concretas para el patrón SINGLETON. En este caso, tenemos dos implementaciones concretas. La primera simplemente es un reflejo de la implementación base. En cambio en la segunda se redefine completamente el comportamiento del método (notar la presencia del atributo `is` en la firma del método `getInstance`) para acceder a la instancia, implementando el idioma DOUBLE CHECK LOCK.

```
<concreteImplementations>
  <implementation name="Basic Sample">
    <implementationInfo>
      <friendlyName>Sample from the GoF book</friendlyName>
      <description>Basic implementation of the Singleton Pattern. Does not
work on multithreaded environments</description>
    </implementationInfo>

    <instanceOf role="Singleton" name="Singleton"/>
  </implementation>

  <implementation name="Double Check Lock">
    <implementationInfo>
      <friendlyName>Double Check Lock Singleton</friendlyName>
      <description>Implementation of the Singleton using the Double Check
Lock idiom (from POSA, Volume 2)</description>
    </implementationInfo>

    <instanceOf role="Singleton" name="SingletonMulti">

      <properties>
        <member scope="private" name="instance" type="SingletonMulti"
                  isClass="true" isVolatile="true" is="instance"/>
        <member scope="private" name="padlock" type="object"
                  isClass="true" instanceType="object"/>
        <property scope="private" name="padlock" type="object"
                  isClass="true" instanceType="object"
                  comment="Padlock for the double-check idiom"/>
      </properties>

      <behaviour>
        <method name="getInstance" is="getInstance">
          <signature scope="public" returns="SingletonMulti"
                        isClass="true" isAbstract="false"/>
          <implementation>
          <if>
          <condition>
            <isNull variable="instance"/>
          </condition>
          <truePart>
            <locked on="padlock">
              <if>
                <condition>
                  <isNull variable="instance"/>
                </condition>
                <truePart>
                  <createInstance variable="instance"
                                        type="SingletonMulti"/>
                </truePart>
              </if>
            </locked>
          </truePart>
          </if>
            <return variable="instance"/>
          </implementation>
        </method>
      </behaviour>
    </instanceOf>
  </implementation>
</concreteImplementations>
```

Apéndice C
Gestión del Conocimiento Mediante la Meta-Especificación y Catalogación de Patrones

En este apéndice se incluye una figura con el contenido del poster presentado en la 21st ACM SIGPLAN Object Oriented Programs, Systems and Applications Conference (OOPSLA 2006) con el objetivo de compartir y constrastar el modelo con la comunidad científica e investigadora en el ámbito de orientación a objetos y patrones. En el poster se presenta una panórámica general del modelo completo desarrollado, permitiendo visualizar y comprender (a alto nivel) en forma rápida y concreta los contenidos presentados en esta tesis. Este poster fue acompañado por un extended abstract [Welicki06b].

En este trabajo se resume la visión de gestión del conocimiento aplicada a la ingeniería del software presentada en esta tesis que surge a partir de la meta-especificación y catalogación de patrones. Para más detalles, ver [Welicki06b]


Sharing patterns is sharing knowledge
The patterns life-cycle transforms tacit engineering knowledge into explicit knowledge
Knowledge and Experience
Discovering and Formalization
Discussion and Refinement
Publishing and Sharing
Tacit
Explicit
Knowledge Dimension
Individual
Interorganizational
Patterns are explicit forms of software engineering knowledge
Explicit Knowledge
Epistemological Dimension
Patterns
Tacit Knowledge
Ontological Dimension
Individual
Group
Organizational
Interorganizational
Knowledge Level
Patterns do not live in isolation...We need more than patterns to convey an idea!
Pattern Language
Source
Author Group
Category
Pattern
Event
Pattern Type
Refactoring
Role
OO Principle
Abstraction Level
The relationships between patterns and other concepts are key elements for sharing explicit contextualized engineering knowledge
<< Pattern >> GRASP.Creator
contains concrete cases of
<< Category >> GoF.Creational
<< Pattern >> GoF.AbstractFactory
<< Pattern >> GoF.Builder
<< Pattern >> GoF.FactoryMethod
<< Pattern >> GoF.Prototype
<< Pattern >> GoF.Singleton
Codify, organize, classify and share software engineering knowledge freely in a world-wide scale
Users
IDEs
Commercial Tools
Systems / Applications
Internet
Patterns Meta-Specification and Cataloging
Catalog
➢Store entities and relationships
➢Expose entities to the world
➢Search facilities
➢Registry
➢Addressing
➢Versioning
➢Alive (community / collaboration)
➢Syndication, subscriptions
➢Dynamic ontologies
➢Artifact generation
➢Handle incomplete entities
➢Virtual iterators
➢Secure and auditable
Patterns Browser
➢Web-based easy-to-use user interface for the catalog
➢Multiple views on an entity
➢Catalog browsing
➢Share knowledge
Meta-Specification Language
➢Describe entities at a very high level of abstraction
➢Describe the knowledge and implementation levels
➢Describe relationships between entities
➢Annotate entities with metadata
➢Allow describing incomplete entities
Share
Manage
Communicate
Knowledge Sources
Books
Proceedings
Journals
People
Web
Experience
Software
Research
The proposed model changes the patterns life-cycle, making it more dynamic and interactive, supporting constant pattern refinement and evolution: once a description of pattern exists in the system, it can be shared with the world.
Knowledge and Experience
Discovering and Formalization
Discusión and Refinement
Publishing and Sharing
Individual / Small Group
Participation
Community
Discussion is no longer a fixed step in the middle of the cycle since the visualization tool has mechanisms to discuss any pattern in the catalog at any stage in the cycle.
Publishing is neither a static step at the end of the cycle anymore. Authors can refine their patterns using as input comments of users of the entire world or new experiences.
© 2006, León E. Welicki