OO-Navigator: Un Framework para Hipermedia

Extendiendo aplicaciones orientadas a objetos con funcionalidad de hipermedia

Trabajo de grado de la Licenciatura en Informática
de

Alejandra Garrido

dirigida por
Dr. Gustavo H. Rossi

Facultad de Ciencias Exactas
Universidad Nacional de La Plata
Abril, 1997

Índice

Dedicación 6
Agradecimientos 6

1. Introducción 8
1.1. Objetivo del trabajo 10
1.2. Motivación 10
1.3. Contribuciones 11
1.4. Conceptos fundamentales estudiados en el desarrollo 12
1.4.1. Hipertexto e hipermedia 12
1.4.2. Modelos formales y metodologías de diseño en hipermedia 14
1.4.2.1. Modelos formales 14
1.4.2.2. Extensiones a los modelos formales 19
1.4.2.3. Metodologías de diseño 21
1.4.3. Funcionalidad de hipermedia: un nuevo enfoque 25
1.4.4. Frameworks orientados a objetos 27
1.4.5. Patterns de diseño 31
1.4.6. Patterns y frameworks 32
1.5. Clasificación de las aplicaciones que podrán beneficiarse de este trabajo 34

2. Desarrollo del trabajo 36
2.1. Obtención del modelo esencial del dominio 38
2.1.1. Modelo unificado de Hipermedia derivado de los modelos y metodologías del dominio. 38
2.1.1.1. Niveles de definición de los componentes 38
2.1.1.2. Componentes del nivel de hipermedia 39
2.1.2. Extensión del modelo base. 47
2.2. Diseño del framework 51
2.2.1. Jerarquías principales 51
2.2.1.1. Jerarquía de Componentes de la Hipermedia 51
2.2.1.2. Jerarquías de Clase de Nodo y Clase de Link 61
2.2.1.3. Jerarquía de Estrategia de definición de Nodos-Colección 65
2.2.1.4. Jerarquía de Destino de Link 67
2.2.1.5. Jerarquía de Resolvedor de Destino de Link 68
2.2.1.6. Jerarquía de Clase de Destino de Link 72
2.2.2. Biblioteca de componentes 73

2.2.3. Patterns utilizados 78
2.2.3.1. Diseño de la relación objeto ↔ nodo ↔ interface: Pattern Observer 78
2.2.3.2. Diseño de la agregación de nodos: Pattern Composite 81
2.2.3.3. Diseño de la clasificación de nodos y vistas-de-nodos: Pattern Type-Object 82
2.2.3.4. Agregado de funcionalidad de hipermedia a una interface existente: Pattern Decorator 84
2.2.3.5. Estrategias de creación de Nodos-Colección: Pattern Strategy 85
2.2.3.6. Conexión entre los tres niveles de la arquitectura: Pattern Adapter 86
2.2.4. Patterns descubiertos 87
2.2.4.1. Navigation Strategy 87
2.2.4.2. Navigation Observer 91
2.3. Implementación del framework 95
2.3.1. Lenguaje utilizado; ventajas y desventajas del mismo; extensiones realizadas al nivel de interface 95
2.3.2. Definición de widgets adaptados a hipermedia 99
2.3.3. Herramienta auxiliar desarrollada: Browser de hipermedia 102
3. Utilización del framework 106
3.1. Cómo se instancia el framework 108
3.2. Estudio del modelo de la aplicación 108
3.3. Descubrimiento de las relaciones entre componentes de la aplicación 110
3.4. Pautas para la creación de los componentes hipermediales 112
3.4.1. Contextos 112
3.4.2. Clases de nodos 113
3.4.3. Hiper-nodos 113
3.4.4. Navegadores 114
3.4.5. Colecciones 114
3.4.6. Clases de links 114
3.4.7. Links no-clasificados 115
3.4.8. Vistas de nodo 116
3.4.9. Representaciones 116
3.4.10. Interfaces 116
3.4.11. Anclas de link 117
3.5. Utilización del Browser de Hipermedias para la instanciación del framework 118
4. Desarrollo de un ejemplo de utilización del framework 126
4.1. Modelo de la aplicación subyacente 128
4.2. Modelo de la hipermedia 129

5. Resultados del trabajo 136
5.1. Publicaciones 138
5.2. Experiencias en el uso del framework 141
5.2.1. Sistema de información académico 141
5.2.2. Aplicación en herramientas CASE 142

6. Conclusiones: Fundamentación de la utilidad del framework y posibles extensiones 144
6.1. Conclusiones 146
6.2. Fundamento de las contribuciones enunciadas 146
6.3. Extensiones 149

7. Bibliografía 150

8. Glosario de términos utilizados 158

Índice de Figuras

Figura 1: Capas del modelo de Dexter. 15
Figura 2: Vista exterior de un framework, como "caja negra" con sus hot spots..... 29
Figura 3: Vista interior de un framework con las interrelaciones entre componentes 30
Figura 4: Vistas de nodo sobre un objeto. 44
Figura 5: Vistas de nodo y representaciones 45
Figura 6: Primer y segundo nivel en la jerarquía de HypermediaComponent..... 51
Figura 7: Jerarquía completa de Node. 52
Figura 8: Jerarquía de HyperLink y su relación con los colaboradores EnpointSolver y LinkEndpoint. 58
Figura 9: Diagrama de interacción con activación del destino. 59
Figura 10: Jerarquía de NodeClass 63
Figura 11: Jerarquía de CollectionStrategy 66
Figura 12: Jerarquía definida a partir de LinkEndpoint 68
Figura 13: Jerarquía de EndpointSolver 69
Figura 14: Jerarquía de LinkEndointClass 72
Figura 15: Contextos y Vistas de Nodo 74
Figura 16: Agregación de DataSpec a dos niveles 76
Figura 17. Niveles de clase e instancia 77
Figura 18: Jerarquía de Nodo preliminar 80
Figura 19: Jerarquía de Link 80
Figura 20: Uso del pattern Composite en la jerarquía de Nodo. 81
Figure 21: Relación entre ObjectNode y NodeClass. 83
Figura 22: Pattern Navigation Strategy. 88
Figure 23: Pattern Navigation Observer. 92
Figura 24: Browser de Hipermedia 103
Figura 25: Herramienta de construcción de la interface de un nodo. 104
Figura 26: Cardinalidad de la relación objetos ↔ nodos. 109
Figura 27: Partes del Browser de Hipermedia. 118
Figura 28: Ventana para la definición de contextos. 120
Figura 29: Ventana para la definición de una clase de nodos 121
Figura 30: Ventana de descripción de links 121
Figura 31: Ventana de creación de links 122
Figura 32: Paletas para la creación de widgets en la interface de una clase de nodos (a la izquierda) y en la interface de un hiper-nodo (a su derecha). 123
Figura 33: Propiedades de un TextEditorHyperView. 124
Figura 34: Ventana de definición de hotwords. 125
Figura 35: Diagrama de clases simplificado del Modelo de Casos de Uso y Modelo de Análisis de la metodología OOSE 129
Figura 36: Diagrama del modelo de la aplicación hipermedia 133

Dedicación

Este trabajo está dedicado a mis padres, quienes me guiaron, me apoyaron y alentaron durante toda mi vida, quienes merecen todo mi esfuerzo.

Agradecimientos

Agradezco ante todo a Gustavo Rossi, quien depositó en mí toda su confianza para la realización de este trabajo y todos los que surgieron a partir de él, me dirigió brindándome todo su apoyo, compartimos largas horas de discusión y los muchos resultados alentadores obtenidos.

Francisco Vives y Pablo Zanetti trabajaron duro a la hora de terminar prototipos. Ellos construyeron gran parte de la herramienta gráfica para instanciar el framework, y desarrollaron el ejemplo del sistema académico. Su aporte fue muy valioso en este trabajo.

A mis amigos Ramiro González Maciel, quien leyó cuidadosamente esta tesis, y Fernando Das Neves, con los que compartí muchas discusiones constructivas, no sólo en este trabajo sino durante toda la carrera que aquí finaliza.

Mi abuela Lucia Viroletti, y mi amiga Sandra Marty siempre estuvieron a mi lado brindándome todo su cariño y fuerza en este camino que elegí emprender.

A todo el LIFIA, y en especial al grupo de Objetos, por el excelente ambiente de trabajo y ayuda mutua.

1. Introducción

Este capítulo conforma el marco de desarrollo del presente trabajo de grado. Las tres primeras secciones plantean el objetivo que se ha perseguido, su motivación y principales contribuciones, las que luego serán fundamentadas en los siguientes capítulos.

La cuarta sección presenta el estado del arte con respecto a los conceptos básicos sobre los que se ha trabajado, y al mismo tiempo explica qué faceta de cada uno es la que ha incidido en esta tesis. Cabe destacar que se suponen conocimientos básicos sobre el paradigma de orientación a objetos.

La quinta sección caracteriza las aplicaciones que podrán hacer uso de este trabajo a distintos niveles.

1.1. Objetivo del trabajo

Construir una arquitectura que brinde a cualquier aplicación construida bajo el paradigma de orientación a objetos, la capacidad de agregar características de navegación a su funcionalidad, en forma integrada, pero sin mezclar u obscurecer con esta nueva funcionalidad el propio modelo de la aplicación ni su comportamiento.

1.2. Motivación del trabajo

Este trabajo surgió a partir de un proyecto que involucra el estudio de ambientes de ingeniería de software, mediante la construcción de un conjunto de herramientas integradas, que además de documentar el esfuerzo de desarrollo dentro de las distintas etapas del ciclo de vida, ayuden en el proceso de pensamiento, capturando las deliberaciones de diseño y las asociaciones libres o restrictas entre las distintas etapas o componentes de cada una. Estamos hablando entonces de una herramienta CASE que permita realizar un seguimiento integrado del proyecto, rastreando los requerimientos de negocios hasta su implementación, pasando por las componentes intermedias que lo modelan, y por el razonamiento que indujo a la existencia de cada componente.

Dados los objetivos planteados en el proyecto, fue necesario el estudio de aplicaciones hipermedia, cuya navegación prometía ser la solución más apropiada para el problema de conexión de distintos componentes y etapas de cierta metodología, así como el soporte para las deliberaciones o las asociaciones libres, significativas para el mismo desarrollador. El uso de hipermedia en trabajo colaborativo ya ha sido estudiado y ha demostrado ser muy apropiado [Conklin+88, Haake+92]. Aún así, las características de navegación son generalmente incluidas con el resto de la funcionalidad de la aplicación, con el consiguiente decremento de mantenibilidad y reusabilidad.

El paradigma de orientación a objetos ha demostrado ser la manera más efectiva de lograr *reusabilidad*, pues permite definir mediante abstracción y composición, los diferentes componentes de una aplicación, que son conectados para lograr el comportamiento esperado del sistema. El estado del arte en la construcción de arquitecturas complejas y reusables en el campo de la orientación a objetos, es la construcción de *frameworks* que modelan un dominio específico. Siguiendo con esta idea, entonces, se planteó la construcción de un framework orientado a objetos (OO) modelando las características hipermediales. Aprovechando la abstracción que esta arquitectura puede proveer, se planteó su campo de aplicación no sólo para la ingeniería de software, sino para cualquier aplicación con un modelo orientado a objetos, para la cual puede resultar útil el agregado de navegación hipermedial.

1.3. Contribuciones

Las principales contribuciones de este trabajo son:

- Se ha definido un modelo unificado de conceptos de funcionalidad de hipermedia.
- Se fomenta la utilización de funcionalidad de hipermedia para realzar aplicaciones OO, mediante la posibilidad de integrar esta funcionalidad con el comportamiento propio de la aplicación, y gracias a la extensión conseguida en el concepto de relación entre componentes, con el agregado de atributos y comportamiento a la relación.
- Se han aplicado los principios de la tecnología de objetos en su concepción actual al campo de hipermedia, logrando los beneficios conocidos del paradigma OO, como reusabilidad, extensibilidad y mantenibilidad, a través de un modelo OO para las aplicaciones en el dominio.
- Se ha construido una herramienta que permite extender cualquier aplicación OO con aquella funcionalidad.
- Se han descubierto patterns de diseño usados recurrentemente en aplicaciones hipermedia y en el diseño de la arquitectura construida, sugiriendo su utilidad en la documentación de frameworks OO.

1.4. Conceptos fundamentales estudiados en el desarrollo

1.4.1. Hipertexto e hipermedia

El concepto de *hipertexto* [Nielsen93] se caracteriza por permitir la creación y representación de links o asociaciones entre porciones discretas de datos, de manera de organizar la información para ser accedida por los usuarios según su interés o necesidad. Hipertexto es simultáneamente un método de guardar y de recuperar información no lineal. Incorpora la noción de asociación de datos, permitiendo a los usuarios la posterior navegación a través de una red de información. Cuando estos datos pueden ser gráficos o sonido, así como texto o números, la estructura resultante se conoce como *hipermedia*. En otras palabras, se suele considerar hipermedia como la suma entre los conceptos de hipertexto y multimedia, definiendo la última como la colección de formas de presentar información que están relacionadas con el paso del tiempo.

Las dos características fundamentales de las aplicaciones hipertextuales e hipermediales son:

- la información es dividida en unidades pequeñas autocontenidas, que llamaremos *nodos*.
- las unidades de información se interconectan mediante *links*, formando una red. El usuario puede *navegar* por la red de hipermedia seleccionando links para viajar de un nodo a otro.

Otros elementos que siempre encontramos en este tipo de aplicaciones son:

- *anclas de links:* regiones dentro de los nodos que denotan el origen de un link.
- *estructuras de acceso:* las que permiten acceder a la información de una manera más directa o adaptada a las necesidades del usuario.
- *historia de navegación:* que provee cierta referencia en cuanto al estado de la navegación para orientar al usuario.

☞ Este trabajo no intenta ocuparse de desarrollar una completa funcionalidad multimedial, sino que está enfocado en brindar capacidades hipertextuales, y en la forma de integrarlas completamente con el comportamiento de la propia aplicación en la que se agregan, de manera que el usuario no se encuentre enfrentado a características que lo apartan de la finalidad del sistema. De todos modos, la herramienta desarrollada soporta gráficos y animaciones, y puede ser extendida para soportar sonido y otros medios.

Por otro lado, las aplicaciones hipermediales existentes pueden identificarse como pertenecientes a alguna de las siguientes dos categorías:

- aplicaciones cerradas que permiten al usuario explorar una base de información, con énfasis en estructuras de acceso adaptadas al usuario y en la representación gráfica (por ejemplo Microsoft's Art Gallery);
- subsistemas separados e independientes que permiten la navegación sobre ciertos datos, cuando el usuario explícitamente invoca el ingreso a este subsistema, y del cual debe salir para poder continuar con la aplicación que retiene el control (Microsoft' Standard Help System).

Otra categorización puede ser introducida en cuanto a la dimensión de aplicaciones que se consideran en este dominio:

- **Hiperdocumentos**

  Se consideran hiperdocumentos las páginas que pueden ser introducidas como nodos de una red hipermedial, como por ejemplo las páginas de World Wide Web o los documentos de ayuda de Microsoft Windows™.

- **Aplicaciones hipermedia**

  Son aplicaciones con las características mencionadas anteriormente.

- **Sistema de hipermedia**

  Son sistemas de información desarrollados para la creación de hiperdocumentos o aplicaciones hipermedia. ***El trabajo que aquí se presenta cae dentro de esta categoría, ya que permite al usuario crear aplicaciones hipermedia adaptadas a sus propias necesidades.***

1.4.2. *Modelos formales y metodologías de diseño en hipermedia*

En el dominio de Hipermedia se han desarrollado algunos modelos formales en el intento de capturar los conceptos básicos que caracterizan las aplicaciones en el dominio. Por otro lado encontramos varias metodologías de diseño, que plantean una serie de actividades para la construcción de aplicaciones hipermedia, y proponen su propia definición de los componentes que utilizan.

En esta sección se presenta una introducción a varios de estos trabajos, separándolos en tres grupos: *modelos formales, extensiones a los modelos formales* y *metodologías de diseño.*

De cada uno de estos trabajos se han extraído los conceptos que en su conjunto constituyen el modelo esencial de hipermedia en el que nos hemos basado. El siguiente capítulo detalla cada uno de los componentes elegidos.

1.4.2.1. Modelos formales

Estos modelos tratan de formalizar las abstracciones encontradas en el dominio. Cabe destacar que no son modelos recientes ya que fueron desarrollados en los años 90 y 91, y desde ese momento no ha aparecido una nueva formalización, al menos tan relevante como éstas. Lo que sí se han desarrollado en estos últimos años son extensiones, principalmente al modelo de Dexter, las que mencionaremos en la siguiente subsección.

- **Dexter Hypertext Reference Model** [Halasz+90, Halasz+94]

  Este se ha convertido en el modelo más importante, ya que muchas extensiones se han planteado sobre el mismo. Fue originado como resultado de las discusiones realizadas en dos workshops sobre hipertexto, y luego desarrollado por Halasz y Schwartz. Su aplicabilidad se debe a que captura tanto formal como informalmente, las abstracciones encontradas en un amplio rango de aplicaciones hipertextuales (NoteCards, Neptune, KMS, Intermedia, Augment), y separa esas abstracciones en tres capas bien definidas. La *capa de almacenamiento* describe la red de nodos y links. La *capa ejecutable* describe los mecanismos que soportan la interacción con el usuario. La *capa intra-componente* incluye el contenido y estructura de cada nodo. Esta separación en capas permite concentrarse en las funcionalidades de cada una, y así poder extenderlas sin necesidad de modificar el resto. El modelo se focaliza en la capa de almacenamiento y en las conexiones entre las capas, que pueden observarse en la Figura 1.

  A continuación se describen las capas propuestas en este modelo.

  Capa de almacenamiento

  Describe una base de datos de *componentes base* organizados en jerarquías de composición e interconectados por *links* que los relacionan. No modela la estructura interna de los componentes. Las funciones de *resolución* y *acceso* son responsables de la recuperación de componentes.

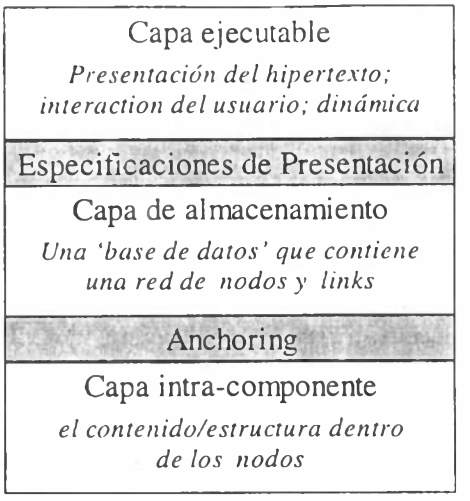


Figura 1: Capas del modelo de Dexter.

Un componente base puede ser un *átomo*, un *link* o un *compuesto*. Un átomo es el dato más primitivo en esta capa, y constituye lo que algunos sistemas llaman *nodo*. Un link representa la relación entre dos o más componentes, definido como una secuencia de dos o más especificadores de puntas de link. Un compuesto contiene otros componentes organizados en forma de DAG (DiGrafo Acíclico). Este tipo de estructuras jerárquicas son elevadas de este modo al mismo nivel que los nodos y links, como funcionalidad de hipermedia explícita, más allá de una red pura.

Cada componente tiene una identificación propia global (UID). La función de acceso recupera un componente dado su UID, y la función de resolución permite calcular el UID a partir de una especificación, en el mismo momento de activar el link *(just-in-time)*, para luego pasarle el resultado a la función de acceso.

Para permitir links de una "zona" del componente a otra, aparece el concepto de *ancla*, como un par (id, valor), donde *id* es el identificador del ancla dentro del nodo, y *valor* es una primitiva para esta capa, que sólo será resuelta por la capa intra-componente. Un *especificador de punta de link* se compone de (especificación del componente, id de ancla, dirección, especificación de presentación). La dirección puede ser: 'desde', 'hasta', 'bidireccionado' o 'ninguna', lo que significa que la componente será el origen, destino, ambas cosas o ninguna. La especificación de presentación es una primitiva que forma parte de la conexión entre la capa de almacenamiento y la capa ejecutable. No permite "dangling links", es decir, especificadores de puntas de links no completos.

Un *componente* está formado por un componente base y por información del componente, que incluye una secuencia de anclas, una especificación de presentación y un conjunto de pares (atributo, valor) arbitrarios.

Además del modelo de datos, esta capa define un pequeño conjunto de operaciones sobre el mismo.

Capa intra-componente

Describe la estructura interna de cada componente, con cualquier tipo de datos. Esta capa no está cubierta por el modelo ya que estará constituida por una aplicación particular.

Capa ejecutable

Provee el acceso, presentación y manipulación de la red. No cubierto por el modelo, aunque define algunas pautas básicas. En esta capa se *instancian* los componentes, es decir, se presentan o muestran al usuario. Las anclas se instancian como *marcas de links*.

Cada usuario que usa el sistema comienza una nueva *sesión*, dentro de la cual podrá instanciar varios componentes, y luego guardar esas instanciaciones.

Otros conceptos que presenta el modelo son:

Anchoring: Mecanismos de direccionamiento de origen y destino de links dentro de los contenidos de los componentes.

Especificaciones de presentación: Mecanismo por el cual se codifica en la capa de almacenamiento la información sobre cómo se presentará el hipertexto al usuario, como una función de la herramienta de presentación, del componente y/o del link por el que se llegó al mismo.

En síntesis, los conceptos sobresalientes de este modelo son: Componente (Componente base + información), Componente base (Atomo, Link o Compuesto), Átomo (nodo, primitiva de esta capa), Link (representa relaciones entre otros componentes), Compuesto (agregado de componentes), Ancla, Especificador, Marcas de link y Sesión.

- **Nested Context Model for Hyperdocuments (NCM)** [Casanova+91]

Sus autores lo definen como un framework conceptual para la definición, presentación y muestra de documentos. Contiene tres submodelos: el *submodelo de definición*, que contiene los conceptos básicos de nodo y link, combinados con un tipo especial de nodo llamado *nodo de contexto* (el que constituye el concepto central de este trabajo). El *submodelo de presentación* está relacionado con la interface de la aplicación con el usuario; define el concepto de presentación como una de las posibles formas de mostrar un nodo. El *submodelo de navegación* también tiene que ver con la interface, aunque presenta las primitivas de alto nivel que definen las operaciones de navegación, tanto a través de los links como jerárquicamente, a través de las estructuras formadas por los nodos de contexto.

Submodelo de definición

Define los *nodos* como fragmentos de información, y los *links* como los encargados de interconectar nodos formando una red. Los nodos se dividen en

terminales y *contextuales*. Los primeros contienen un identificador y datos, que no tienen estructura interna para el modelo. Los últimos agrupan nodos teminales o contextuales (en jerarquías o conjuntos) y links entre ellos. Es importante destacar que cada nodo puede pertenecer a varios nodos de contexto, pero los links son internos al nodo de contexto al que pertenecen sus puntas, no pudiendo conectar entonces nodos de distintos contextos.

Un nodo contextual define un *hiperdocumento*. Al permitir que un nodo pertenezca a varios hiperdocumentos y sea anidado a diferente nivel, se permite definir diferentes vistas del mismo, basándose en las distintas clases de usuarios. En el modelo, esto se traduce como distintas *perspectivas* de un nodo, donde cada perspectiva determina la secuencia de nodos contextuales a través de la cual un nodo es observado.

Un *link* se define como un par ordenado de *anclas*, cada ancla como un par (nodo, offset), donde el offset puede ser nulo, un desplazamiento dentro del nodo, o otra ancla hacia un nodo interno.

Submodelo de presentación

Define el concepto de *representación* de un nodo como la forma particular de reflejar los valores de sus atributos, de acuerdo a las necesidades de una aplicación dada. Aparece también la idea de *clase de representación* y *clase de nodo*, tal que:

- un nodo puede indicar directamente una representación por defecto en su clase;

- un link puede definir una representación de clase por defecto para cada nodo presente en sus anclas;

- un nodo de contexto puede definir una representación de clase por defecto para los nodos que contiene;

- una clase de nodo puede estar asociada con una representación de clase por defecto.

Define además la noción de *estado* como un par s = (G, P), donde G es un digrafo acíclico cuyos nodos son representaciones, y P es un camino desde un nodo a una raíz de G. Decimos que P es la *perspectiva corriente* de s, y que el primer elemento de P es la *representación corriente* de s.

Submodelo de navegación

Define las operaciones de navegación que son necesarias para recorrer los hiperdocumentos descendiendo a través de los nodos contextuales, o explorando la estructura a través de los links.

Esencialmente, los conceptos aquí propuestos son entonces: Hiperdocumento, Nodo terminal y Nodo contextual, Clase de nodo, Link, Ancla, Perspectiva, Representación, Clase de Representación, Representación por defecto y Estado de la navegación.

- **Post-Prototype Formal Specification (PPFS)** [Lange90a, Lange90b]

  El nombre de este modelo se debe a la importancia que le concede a la formalización recién después de cada paso en el refinamiento de un prototipo.

  Los componentes del modelo básico hipertextual que presenta son: nodos, links y estructuras (como conjuntos, secuencias, mapas o árboles).

  Los *nodos* se definen como fragmentos nombrados de información, divididos interiormente en *slots* de datos. Pueden tener *atributos*, para reflejar por ejemplo el tipo de nodo.

  Una red hipertextual está conformada por una colección de *links* identificados por su nombre, que parten de un origen o *ancla* y llegan a un *destino*, pudiendo ser ambos "partes" de un nodo. En realidad un link puede tener múltiples anclas y múltiples destinos. Los links también pueden tener atributos, para identificar su "tipo". Además presenta el concepto de links de segundo orden, es decir aquellos que tienen como destino otro link, aunque no justifica claramente la necesidad de los mismos. El destino de un link puede ser en realidad una función que se interpreta al momento de navegarlo, y genera el destino o una vista particular del mismo (aunque no clarifica qué significa que un mismo nodo pueda tener o no varias vistas).

  Las *estructuras* constituyen distintas formas de organizar el hipertexto, como conjuntos, secuencias, mapas o árboles. No contienen interiormente los nodos sino su identificación.

  El modelo especifica las operaciones básicas posibles sobre cada componente, y para esto utiliza los conceptos de *orientación a objetos*, encapsulando la estructura y presentando las operaciones como interface de cada componente.

  En síntesis este modelo define: Nodo, Slot de nodo, Link, Link de segundo orden, Ancla, Destino de link, Atributos de nodos y links, Estructuras, operaciones como interface de cada componente.

1.4.2.2. Extensiones a los modelos formales

Como se mencionó anteriormente, no se han desarrollado nuevos modelos formales sino extensiones a los mismos, sobre todo, y como veremos aquí, al modelo de Dexter. Se elige este modelo porque es el que captura las mejores ideas de diseño de un grupo de grandes sistemas de hipermedia de su época.

- **DHM (Devise Hypermedia Development Framework)** [Grønbaek+94a], [Grønbaek94b]

  En este trabajo se toma al modelo de Dexter como base para convertirlo en un diseño orientado a objetos. Se presenta un framework conceptual mediante la definición de las distintas clases que implementarían el modelo de Dexter.

  Los clases que define para la capa de almacenamiento del modelo de Dexter son:

  - *Hypertext*. Encapsula el conjunto de componentes.

  - *Component*. Clase abstracta que modela los componentes. Entre sus subclases se encuentran: *AtomComponent*, *LinkComponent* y *CompositeComponent*, que mapean respectivamente los conceptos de Átomo, Link y Compuesto de Dexter. El AtomComponent tendrá una subclase por cada tipo de dato que se necesite; por ejemplo TextComponent sería una componente textual. De esto se deduce que un AtomComponent sólo puede contener un tipo de dato particular.

  En DHM se ha realizado un amplio estudio respecto a componentes compuestos. El framework soporta todo lo propuesto en el modelo de Dexter, como las nociones de *compuestos calculados* y *compuestos virtuales*, aunque destaca ciertas limitaciones del modelo, amplia las posibilidades y las clasifica. Así, un *BCCompositeComponent* puede contener sólo AtomComponents; *LinkComposite* sólo puede contener *LinkComponents*; *CompositeComponent* puede contener cualquier tipo de componente, y puede ser virtual. Estas clases de compuestos mantienen sus componentes por referencia, permitiendo así que un mismo componente sea referenciado en distintos compuestos, mientras que *ContainerComposite* los mantiene por inclusión, con la ventaja de disminuir el tiempo de borrado y acceso a los componentes. Son también subclases de Composite: *TableTopComposite* y *GuidedTourComposite*. Estas clases representan los conceptos de tabletops y guided tours respectivamente, propuestos por Trigg en [Trigg88], donde un tabletop es un dispositivo para mostrar varios valores coordinados en una sola ventana, y un guided tour conecta distintos tabletops en una estructura de grafo que permite ser recorrida con comandos como Start, Next, Previous, Jump.

  Con respecto a los links, DHM soporta también *dangling links*, aquellos que no tienen un destino especificado.

  El modelo de Dexter no plantea la definición de anclas para los compuestos. DHM va más allá y propone tres tipos de anclas: *whole-component anchor*, o anclas que representan el componente todo; *marked anchors*, o anclas que de alguna manera se hacen visibles en una parte del

contenido del nodo, y *unmarked anchors*, cuya ubicación no se especifica y debe ser calculada o buscada.

Otras jerarquías de clases paralelas a éstas se plantean para la *capa de ejecución*, donde encontramos las clases *Session, Instantiation, LinkMarker.*

Este trabajo también plantea la integración de estructuras de hipermedia con terceras aplicaciones que contienen los datos. Se puntualiza que Dexter deja una puerta abierta hacia este tipo de extensiones gracias a su división en capas, donde la capa intra-componente sería la de la aplicación. Pero también expone sus debilidades: primero, no se distingue entre componentes cuyos contenidos son manejados por la aplicación o por la hipermedia, y segundo, no se especifica qué sucede con documentos de la aplicación que tengan una estructura interna, la forma de anclarlos o estructurarlos. Ante esto, DHM propone distintas posibilidades para guiar al usuario en la estructuración de los componentes:

- *componentes atómicos*: los datos pueden estar incluidos en el componente o estar referenciados por él, por medio de apuntadores a la aplicación o archivo donde se encuentran, usando anclas del tipo whole-component;

- *componentes compuestos*: se presentan distintos tipos, como *"compuestos que contienen componentes"*, ejemplificados con los TableTopComponents mencionados anteriormente, que referencian a sus componentes por medio de punteros; *"objetos de datos encapsulados"*, donde la estructura interna de los datos contenidos es totalmente visible a la hipermedia; y *"compuestos estructurados"*, que mantienen una tabla de claves y referencias a sus componentes.

- **Adding Multimedia Collections to the Dexter Model** [Garzotto+94]

Este artículo presenta un tipo de componente compuesto específico, agregado al modelo de Dexter, llamado *colección*.

Estas colecciones pueden ser consideradas como un *conjunto* de componentes, pero también pueden tener una estructura interna, estar anidadas, y tener información propia. Su propósito es *introducir un patrón de navegación*, basado en su estructura, y diferente al standard de navegación nodo-link. Una colección puede definirse particionando el conjunto de nodos en grupos semánticos consistentes, o agrupando nodos para organizar una presentación temática determinada. También permiten especificar estrategias de sincronización que se presentan con componentes multimediales.

Los miembros de una colección pueden pertenecer al mismo tipo, formando una colección homogénea, o ser de distinto tipo, a lo que se denomina heterogénea.

Las colecciones se pueden definir mediante seis métodos distintos:

- *built-in*: es el único método que no requiere otra colección como punto de partida. Está basado en los mecanismos provistos por el sistema, como tipos de nodos.

- *intensional*: basado en un predicado de selección.

- *pick-up*: donde el autor selecciona los miembros manualmente.

- *set-oriented*: su definición se realiza mediante operaciones de conjuntos;

- *link-oriented*: dada una colección y un subconjunto de links que parten de la misma, obtener la colección resultante de aplicar los links a la colección original.

- *session-based*: más conocido como la historia de nodos visitados.

La estructura interna de una colección puede definirse mediante un orden total, lo que resultaría en una secuencia, o mediante un orden parcial, para formar un árbol o reticulado.

Toda colección tiene un nodo asociado que cumple básicamente tres funciones: ayuda al lector a entender el contenido de la colección, provee información adicional para la colección, y provee mecanismos de acceso a los miembros de la colección. Como posibles mecanismos de acceso se han identificado: Indices y Tours Guiados.

El artículo discute muchos de los problemas que aparecen con el nuevo tipo de navegación introducido por las colecciones, sugiriendo una precisa especificación de las respuestas a cada operación que el usuario puede realizar.

Con respecto al uso de colecciones para la sincronización de nodos multimediales, se propone que el nodo asociado a la colección tenga información relacionada con la forma de presentar y coordinar los distintos miembros.

1.4.2.3. Metodologías de diseño

Las metodologías de diseño son procedimientos informales que intentan dar pautas y herramientas para que el desarrollador (sobre todo el no experimentado), pueda construir aplicaciones en forma documentada, correcta y mantenible. Esta es el área donde encontramos más trabajos recientes, ya que está en continuo desarrollo en el campo de Hipermedia. Presentamos aquí una síntesis de aquellas metodologías que consideramos más relevantes o que incluyen a otras.

- **HDM - Hypermedia Design Model** [Garzotto+91]

Esta podría considerarse la metodología más renombrada o referenciada, no por ser la más completa, sino porque sentó las bases de muchas que se desarrollaron a posteriori. Plantea la problemática de desarrollar aplicaciones hipermedia "in the large", cuando es necesario proveer al autor de primitivas que le permitan describir los componentes en forma más concisa, clara e independiente del sistema.

De acuerdo con HDM, una aplicación está compuesta de *Entidades*, que a su vez están formadas por una jerarquía de *Componentes*. Las Entidades pertenecen a un *Tipo*. Pueden conectarse a otras Entidades o Componentes a través de *Links* que a su vez son *Estructurales* ó *de Aplicación*. Los Componentes pueden ser instanciados por una o más *Perspectivas* en

Unidades. Las Unidades proveen una referencia contextual a la información contenida en sus *Cuerpos*. Un esquema HDM es entonces un conjunto de definiciones de tipos de Entidades y Links de Aplicación.

- **OOHDM - Object-Oriented Hypermedia Design Model** [Schwabe+95, Schwabe+96]

  Esta metodología es la que más se acerca al trabajo desarrollado en esta tesis. Por un lado, porque ha inspirado en mayor medida nuestro modelo esencial de hipermedia, y por otro lado porque la separación de actividades que plantea es muy similar a nuestra separación en niveles, y por lo tanto podría ser aplicada como metodología de desarrollo que luego se implementa directamente en nuestro framework.

  OOHDM presenta una manera sistemática de desarrollo, en la que se pueden plantear aspectos del dominio, la estructura de la aplicación hipermedia y su semántica navegacional, independientemente de lo que concierne a la implementación. Provee construcciones de diseño de alto nivel y mecanismos de abstracción basados en el paradigma OO, para acrecentar las posibilidades de reuso de diseño.

  La metodología está compuesta de cuatro actividades iterativas: *Diseño Conceptual*, *Diseño Navegacional*, *Diseño Abstracto de Interface*, e *Implementación.*

  Diseño Conceptual

  Durante el diseño conceptual se construye un modelo OO del dominio de aplicación. Para la construcción de este modelo se han incorporado los conceptos de perspectivas de atributos y subsistemas, y se ha realzado el concepto de relación entre componentes. El propósito de esta actividad es capturar la semántica del dominio, sin tener mayormente en cuenta los tipos de usuario y sus tareas.

  Diseño Navegacional

  Una aplicación en OOHDM constituye una vista navegacional sobre un dominio conceptual (el construido como primera actividad). En este paso el diseñador toma en cuenta los distintos tipos de usuario y las tareas que cada uno debe realizar utilizando la aplicación.

  Esta separación en actividades permite construir diferentes modelos de navegación sobre el mismo esquema conceptual, por cada rol o perfil de usuario. La estructura de navegación de una aplicación hipermedia se define entonces a partir de un esquema formado por nodos, links y estructuras de acceso. Los nodos se definen como "vistas" de las clases del modelo conceptual, permitiendo que un mismo nodo pueda combinar atributos de diferentes clases relacionadas en aquel esquema. Los nodos contienen atributos tipados y anclas de links, y pueden ser atómicos o compuestos. De la misma forma, los links reflejan las relaciones del modelo conceptual que serán exploradas por el usuario mediante la navegación.

Diseño Abstracto de Interface

Esta actividad comprende la construcción de un modelo abstracto de la interface que hará perceptible el esquema de navegación definido en el paso anterior. Esto implica definir los objetos de interface que el usuario percibirá, y la forma particular en la que los nodos, links y estructuras de acceso se van a mostrar, qué objetos de interface activarán la navegación, la forma en la que los elementos multimediales serán sincronizados, y las transformaciones de interface que tendrán lugar.

Nuevamente, la separación de esta etapa permite construir diferentes interfaces para el mismo modelo de navegación. OOHDM propone la utilización de un modelo formal denominado Abstract Data Views, para describir la interface de los objetos de navegación.

Implementación

En esta actividad el diseñador debe realizar un mapeo de los modelos de navegación y de interface abstracta en objetos concretos, disponibles en el ambiente elegido de implementación.

Más allá de las actividades en las que se separa esta metodología, los principales conceptos que define son: Nodo, Clase de Nodo, Link, Clase de Link, Ancla, Contexto de navegación, Estructura de Acceso.

- **Object-Oriented Design Method for Hypermedia Information Systems** [Lange94]

  Presenta una procedimiento informal de diseño para el desarrollo de sistemas de información basados en hipermedia. En este material se discute la incorporación de tecnología de hipermedia en los sistemas de información de las grandes empresas. También utiliza varios conceptos del paradigma OO en su desarrollo.

  Comienza con un *análisis OO* que puede obtenerse mediante la aplicación de una metodología OO convencional. Este análisis sirve como entrada para la construcción de un esquema con el *Modelo de Realce de Relaciones entre Objetos* (EORM). EORM fue desarrollado en el contexto de esta metodología para elevar a las relaciones entre objetos al mismo nivel que los objetos, como interacciones semánticamente ricas. El esquema finalmente logrado formará el esquema de la aplicación hipermedia.

  El principal aporte de este método es el realce de las relaciones en un modelo OO, que luego se traducen a distintas clases de links. En cuanto a sus carencias, no presenta una clara separación de pasos en la construcción de los distintos modelos, mezcla el modelo del dominio y el modelo de hipermedia, y no especifica como hacer el mapeo del comportamiento de los objetos del dominio.

- **RMM - Relationship Management Methodology** [Isakowitz+95]

  Esta es una metodología de diseño y construcción de aplicaciones de hipermedia que exhiben una estructura regular en el dominio de interés, es decir basada en clases de objetos y relaciones identificables.

  Su modelo de datos se basa en HDM: considera entidades, componentes, todos los tipos de links, unidades, etc., aunque no soporta perspectivas. Desarrolla el concepto de *estructura de acceso* que ha influenciado en mayor medida en el modelo esencial que hemos construido. Básicamente proveen construcciones para índices y tours guiados con predicados lógicos complejos.

  Además del modelo de datos, RMM presenta una secuencia de pasos a seguir en el proceso de desarrollo de una aplicación hipermedia. Estos pasos involucran: representación del dominio mediante un diagrama de Entidad/Relación, diseño de entidades, diseño navegacional, diseño de conversión del protocolo para la implementación, diseño de la interface, diseño del comportamiento en ejecución, construcción y testing.

1.4.3. Funcionalidad de hipermedia: un nuevo enfoque

El uso de hipermedia se ve afectado por dos grandes problemas: en primer lugar, no existe un modelo unificado de las características que identifican a estas aplicaciones. En segundo lugar, la tecnología de hipermedia ha tenido muy poco o nada de impacto en otros dominios de organizaciones [Isakowitz93]. Sin embargo, ***es posible observar las aplicaciones hipermediales con el objetivo de encontrar sus cualidades, y extraerlas con un nuevo enfoque: ser incluidas en otros sistemas de información, para aumentar su utilización y utilidad.*** Esto ha dado lugar a un nuevo concepto en este campo, llamado *funcionalidad de hipermedia* (FH). Los primeros artículos que motivan la generalización en hipermedia son [Bieber+92, Bieber93, Isakowitz93]. En 1994 se realizó el primer workshop específicamente en este tema [HTFI], y en [Oinas95] podemos encontrar una mejor caracterización del concepto.

Las características claves pertenecientes al conjunto que define la FH, incluyen posibilidades de:

- representar información no-lineal, permitiendo asociar distintos componentes de una aplicación;
- manejar grandes unidades de información a ser compartida por grupos que trabajan en forma colaborativa;
- mejorar la interface de la aplicación;
- definir diferentes contextos para distintos perfiles de usuario;
- mostrar las relaciones mediante browsers y mapas;
- permitir acceso rápido y fácil a la información deseada, por medio de índices y metáforas de navegación;
- guiar la búsqueda de información que mejor se acomode al perfil del usuario mediante tours guiados;
- agregar anotaciones y bookmarks;
- proveer la historia de nodos visitados.

Existen dos enfoques principales con los cuales soportar la FH, como lo define Oinas-Kukkonen [Oinas95]:

- *a nivel del sistema operativo (OHS)*, a través de una facilidad o herramienta de creación de links externa, que permita la asociación de diferentes aplicaciones en un ambiente de hipermedia abierto (es decir con links inter-aplicación), o
- *a nivel del sistema de información*, incorporando características de navegación dentro del entorno de una misma aplicación (es decir con links intra-aplicación).

En el primer enfoque encontramos aproximaciones como las de [Ashman+96], link-servers como Microcosm [Davis+94] y Chimera [Anderson+94], o el mismo WWW, donde un máquina externa permite definir links entre documentos, mediante el direccionamiento del ancla y destino de cada link como offsets dentro de cada documento.

En el segundo enfoque encontramos aproximaciones como la de [Bieber+95], que intenta suplementar con funcionalidad de hipermedia la capacidad computacional de una aplicación particular. Esta aproximación propone nuevas alternativas de visualización y manejo del conocimiento de la aplicación, navegando entre ítems de interés y anotando con comentarios las relaciones o links.

☞ **Nuestra aproximación** se encuentra precisamente en este segundo enfoque, es decir, apuntando a enriquecer sistemas de información, mejorando el acceso a sus componentes [Garrido+96a]. En particular, nos hemos focalizado en un subconjunto de los sistemas de información (SI): aquellos construidos con el ***paradigma de orientación a objetos***. Nuestro objetivo fundamental es lograr una completa integración del comportamiento de la aplicación con la FH, de manera que esta última no tenga que ser necesariamente sobresaliente, sino complementaria de lo anterior.

Uno de los mayores problemas en otras aproximaciones a este enfoque es que se mezclan los aspectos de navegación y los de interface con los de la propia aplicación, impidiendo la construcción de extensiones limpias sobre cada aspecto particular, o el reuso de ciertos componentes. La definición de una ***arquitectura en capas*** es la que mejor soluciona este problema, y si cada capa se puede construir en forma independiente de las otras, y en forma abstracta, se puede llegar al mayor grado de reuso. Y a eso apuntamos.

La separación que proponemos entre la aplicación subyacente y la capa de hipermedia es la que permitirá además ***retro-adecuar las aplicaciones existentes***, es decir enriquecer SI que ya se encuentran construidos, agregando FH sin tener que modificarlos por ello, y en esto se basa la originalidad de este trabajo.

1.4.4. Frameworks orientados a objetos

Los sistemas de software tienden a crecer con el tiempo a medida que se le agregan nuevas características y se cambian las viejas. Pero no son sólo las viejas características que lo hacen crecer, si no que se re-escribe código, y se extiende "a la defensiva", agregando nuevas versiones sin tocar las anteriores, en vez de generalizar la versión actual. El problema surge cuando el diseño no era el adecuado para los propósitos subsiguientes del sistema. Esto resulta en que la estructura de un programa se deteriore con el tiempo.

La única manera de evitar esta decadencia en la estructura de un programa que debe mantenerse es, según Opdyke y Johnson [Opdyke+91], re-escribiéndolo, de manera tal de ir abstrayendo y mejorando su estructura.

La orientación a objetos se promociona como la mejor forma de reusar y extender programas, pero la estructura de un programa OO también se deteriora a medida que se agregan nuevas características. Las jerarquías se hacen grandes y con menos sentido, se duplica código y las clases individuales se hacen grandes y difíciles de entender y manejar.

La manera de resolver esta situación es *refactorizando* código de tiempo en tiempo. Refactorizar un programa OO implica ir creando superclases e ir trasladando hacia ellas el comportamiento común a las subclases, y así obtener *clases abstractas.*

Hacia la abstracción

Al definir *clases* estamos encapsulando el comportamiento común de un conjunto de objetos. Si avanzamos un poco más en el camino a la abstracción, y definimos *jerarquías de clases* y *clases abstractas*, estamos factorizando y abstrayendo el comportamiento común entre distintas clases, subclases de la clase abstracta. Por ejemplo podríamos tener una jerarquía de gráficos o figuras.

Las características de las clases abstractas son:

- no son instanciadas; representan comportamiento común o abstracciones de las subclases;

- algunos métodos de la clase abstracta resultante pueden ser implementados, mientras que sólo falsetes o implementaciones preliminares, o templates, pueden ser provistos para otros. Aunque algunos métodos no puedan ser implementados en la clase abstracta, se especifican sus nombres y parámetros para impedir que los descendientes cambien este protocolo. Entonces, una clase abstracta crea una *interface de clase standard* para todos los descendientes, que es lo mismo que decir que los descendientes soportan los mismos *contratos* que la superclase.

- otros componentes pueden basarse en esta interface standard definida por la clase abstracta. Los componentes confían en los contratos soportados por la clase abstracta, más allá de la subclase que finalmente utilicen, gracias al concepto de *polimorfismo.*

Con la jerarquía de figuras que teníamos podríamos construir un editor gráfico de dibujos. Supongamos ahora que más adelante necesitamos construir un editor gráfico de composiciones musicales. Deberíamos ampliar la jerarquía de figuras, pero tendríamos que construir el nuevo editor desde cero, sin poder usar el editor gráfico anterior, cuando en realidad un editor gráfico debería poder adaptarse a cualquier conjunto de figuras. Acá no tiene sentido hacer una jerarquía de editores, sino que el

mismo editor debería ser lo suficientemente flexible para *adaptarse* o *parametrizarse* con el conjunto de figuras a editar.

Lo que estamos haciendo entonces es, no sólo abstraer el comportamiento común de distintas instancias de la misma clase, sino definiendo la *relación* o *interacción* que tendrán con instancias de otras clases, hablando siempre dentro de un dominio específico (como sería en el ejemplo el dominio de editores gráficos).

☞ Un *framework* en el campo de la OO es un diseño arquitectónico de un tipo particular de aplicación o dominio [Johnson+88]. Consiste de un conjunto de *clases abstractas y concretas*, y define la *interacción* entre los componentes mediante el planteo de restricciones, herencia, polimorfismo y reglas informales de composición.

Las *clases abstractas* conforman los componentes principales de la arquitectura, y por lo tanto representan el esqueleto que va a conservar toda aplicación del dominio que el framework modeliza. El flujo de control que se especifique a través de las interacciones entre componentes deberá capturar las decisiones de diseño comunes en ese dominio.

Un framework tendrá distintas *instancias* de sí mismo en el dominio para el cual fue planteado, lo que implica que su modelo debe ser lo suficientemente general para cumplir con su objetivo fundamental: el *reuso*, no sólo de código sino también de diseño. Cada instancia particular será definida por las *clases concretas* que reemplazarán a las abstractas en tiempo de ejecución, usando los mecanismos de herencia y polimorfismo. El usuario del framework podrá elegir las clases concretas que necesite entre aquellas provistas por el framework en forma de biblioteca, o crearlas por sí mismo. Esto está diciendo que el diseño debe ser *extensible,* es decir, tener una estructura abierta a nuevas subclases. Además, su interface deberá especificar cómo se relacionará con el resto del sistema.

De lo anterior podemos deducir que la construcción de frameworks OO es muy difícil. Requiere un conocimiento profundo tanto del dominio a ser modelado, como del perfil de aplicaciones que podrán hacer uso del mismo. No existen técnicas formales de desarrollo de frameworks hasta el momento, pero nuevos adelantos se están produciendo para permitir una buena documentación de los mismos.

Los beneficios de usar frameworks son considerables, y de ahí que se sigan desarrollando a pesar de su alto costo: construcción más veloz de aplicaciones, y fácil mantenimiento, debido a que todas las aplicaciones que hagan uso de un framework particular exhibirán una estructura de diseño común, y a que sólo se necesita elegir las clases concretas rellenando ese esqueleto, para instanciarlo.

Uno de los frameworks más conocidos es el Model-View-Controller (MVC) [Krasner+88], que permite implementar interfaces gráficas sofisticadas en Smalltalk-80, separando la funcionalidad de la aplicación (Modelo), la de sus Vistas, y la interacción con el mundo exterior (Controlador). Otro framework de magnitud sobre el que se han desarrollado muchos estudios es el framework Choices para sistemas operativos [Campbell+91]. El mismo compromete distintos subframeworks que modelan los distintos servicios que cualquier sistema operativo debería proveer, en forma de subsistemas.

Siguiendo con esta idea hemos construido un framework OO que permite diferentes instanciaciones en el dominio de hipermedia. Una instanciación puede crearse mediante la extensión de alguna aplicación OO ya existente, o puede construirse desde cero.

El framework permite extender una aplicación con FH a través de un mapeo directo de componentes de la aplicación a nodos y links. De esta forma se logran observar los objetos como nodos, y acceder a los relacionados mediante el recorrido de links. Más aún, el framework permite preservar la semántica de comportamiento de la aplicación, como fue especificada en su diseño, proveyendo la comunicación pertinente nodo-objeto para notificar a los últimos sobre la ocurrencia de eventos de interface. Esto no implica que la aplicación debe ser notificada de la presencia de operaciones de navegación, sino por el contrario, sólo se le avisará cuando sea necesario ejecutar las acciones que a ella le competen, logrando una extensión reusable, mantenible y menos compleja.

Revisemos con más detalle cada aspecto de la definición de "framework" para evaluar la dificultad en su construcción y documentación.

Las *clases abstractas* que definen las jerarquías son los ***lugares de articulación*** del framework, aquellos que comprometen los aspectos del dominio que pueden variar de una aplicación a otra, donde el framework debe proveer flexibilidad. Wolfgang Pree llama a estos lugares 'hot spots', y en su libro [Pree95] describe una técnica iterativa para desarrollo de frameworks encontrando estos hot-spots y modelándolos como clases abstractas. La dificultad en el desarrollo de frameworks se traduce entonces en encontrar y modelar exitosamente los hot-spots del dominio. La iteración es necesaria para asegurar y testear que los hot-spots encontrados sean los correctos, y para lograr jerarquías más sólidas mediante *refactorización*.

Los lugares de articulación arriba definidos constituyen las "ventanas" abiertas para el usuario del framework. Todo el resto de la arquitectura debería ser una "caja negra" [Johnson+88] que un usuario no necesite conocer para poder usar el framework. Sólo deberá conocer estas "ventanas" o jerarquías de clases para poder elegir el componente correcto en cada caso.

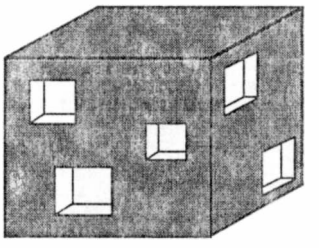


Figura 2: Vista exterior de un framework, como "caja negra" con sus hot-spots

Resulta bastante difícil encontrar el equilibrio en cuanto a cantidad de hot-spots provistos por el framework. Los factores interactuantes serían, por un lado, proveer la mayor flexibilidad posible en la arquitectura, para que pueda ser ampliamente utilizada en el dominio, y por otro lado, proveer la mayor simpleza posible para que no resulte terriblemente complicado utilizarlo.

Para lograr mayor simpleza, un paso necesario en el desarrollo de un framework consiste en construir una herramienta gráfica que permita identificar fácilmente los

componentes necesarios en una instanciación dada, sin tener que codificar para crearlos. Esto es lo que caracteriza a un framework "caja negra".

Mucho se puede aprender del diseño de un framework si éstos documentan las decisiones de diseño involucradas durante el desarrollo. Generalmente la documentación que acompaña a un framework involucra un conjunto de tarjetas CRC [Beck+89], diagramas de clase [Rumbaugh+91], tal vez diagramas de interacción [Jacobson+92], u otros documentos especificados por alguna metodología particular. De todas maneras, *lo que estos documentos muestran son las soluciones, pero nunca el por qué de las mismas, los pasos o razonamientos intermedios.*

Podemos ahondar más aún en este problema, haciendo hincapié en la segunda parte de la definición de un framework, donde dice que éste *define la interacción entre los componentes, marcando el hilo de control que seguirán todas las aplicaciones.* Este hilo de control es el que formará los huesos del esqueleto que une los lugares de articulación, la parte invariante que marcará el comportamiento común de todas las aplicaciones construidas a partir del framework.

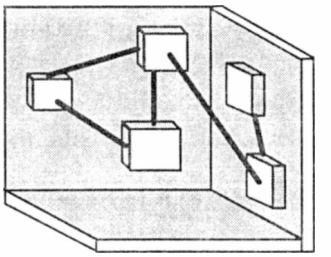


Figura 3: Vista interior de un framework con las inter-relaciones entre componentes

En la definición de estas zonas invariantes también notamos la tensión entre las dos fuerzas o factores que mencionábamos anteriormente, los que podríamos replantear como la necesidad de restringir el comportamiento de los componentes y que el framework siga siendo una "caja negra", y por otro lado no acotar demasiado el posible uso de la arquitectura por el exceso de restricciones.

Volviendo al tema de la documentación que suele acompañar el diseño de un framework, vemos que para el usuario que quiere conocer a fondo ese diseño, para aprender de él y/o para mantenerlo, resulta esencial que la documentación incluya y deje bien implícitas las interacciones definidas entre los componentes, el tipo de interacción, su alcance, restricciones que plantea, su razón de ser, y sus consecuencias.

Un tercer aspecto importante en la especificación de un framework es su relación con el resto del sistema. Un framework constituye una arquitectura que generalmente forma parte de un sistema más grande, con otras capas que podrán estar definidas por otros frameworks. Es necesario entonces hacer explícitos los contratos que el framework mantiene con su exterior.

La mejor aproximación descubierta hasta el momento que puede atacar estos problemas es la de *patterns de diseño*. Ellos están siendo utilizados en la documentación de la funcionalidad de los frameworks [Johnson92], [Beck+94] y en su especificación interna [Pree95]. En la próxima sección daremos una breve introducción a este nuevo concepto, y luego veremos cómo utilizar patterns en la documentación de frameworks.

1.4.5. *Patterns de diseño*

El concepto de *"pattern"*[1] ha sido tomado del campo de la Arquitectura, dada la semejanza que se puede encontrar entre la forma de construir casas, sus partes, manzanas, ciudades, y la forma de construir software (o al menos a la que se pretende llegar). Christopher Alexander nos habla del significado de usar patterns en la Arquitectura, con una actitud completamente nueva de construir y planificar, mediante la observación de que muchas decisiones de diseño que guían hacia soluciones con mayor calidad o estética pueden ser reusadas a distintos niveles y generalizadas lo suficiente como para encontrarlas bajo distintas circunstancias [Alexander79].

☞ En el anterior volumen [Alexander+77] de su serie de libros, él mismo nos dice que cada pattern *describe un problema* que ocurre una y otra vez en nuestro ambiente, y luego *describe el núcleo de la solución* a ese problema, de forma tal que se puede usar esa solución miles de veces, sin hacerlo dos veces de la misma forma.

Esto mismo que Alexander plantea en su área puede ser perfectamente aplicado, e incluso definido de la misma forma, en el caso de los *patterns de diseño* en el área del software. Los patterns de diseño (PsD) constituyen una excelente forma de registrar experiencias de diseño, y reusarlas exitosamente en distintas arquitecturas. Una característica importante es que los PsD no se inventan, *se descubren*. Esto significa que para que un pattern sea tal, se deben probar sus usos previos y exitosos.

Un pattern de diseño (PD) sistemáticamente nombra, abstrae, explica y evalúa un diseño relevante y recurrente en aplicaciones OO [Gamma+95]. Los patterns capturan y hacen explícitas las decisiones de expertos en el contexto específico, de manera de poder reutilizar esa experiencia. Un pattern debe determinar su propósito y consecuencias de manera tal que al momento de desarrollar una nueva aplicación, un diseñador pueda elegir entre distintas alternativas de diseño que hacen a un sistema reusable, y evitar alternativas que comprometan su mantenibilidad, o que no se adecuen al problema específico. El libro de Gamma et al. [Gamma+95] presenta un catálogo de patterns básicos, en el sentido que describen la relación entre objetos y clases que se comunican de forma de resolver un problema de diseño general, en un contexto particular. Por la simpleza con que está presentado, se ha tornado en el libro de cabecera para los desarrolladores a distintos niveles. El catálogo presenta los PsD con un formato fijo, que involucra: nombre del pattern, propósito, otros nombres con los que se le puede conocer, motivación, aplicabilidad, estructura, participantes, colaboraciones, consecuencias, implementación, código de ejemplo, usos conocidos y patterns relacionados. La inclinación de los investigadores en esta nueva área es la de mantener un formato fijo, de manera de llegar a standarizarlos y colocarlos en un repositorio a ser accedido libremente. A pesar de esto el formato varía cuando los patterns no son tan específicos, por ejemplo, cuando se aplican al dominio de la organización [Coplien95]. El formato utilizado por Coplien es: problema, contexto, fuerzas que interactúan, solución y contexto resultante.

Por otro lado, Beck y Johnson [Beck+94] definen *patterns generativos* como aquellos que se focalizan en cuándo usar el pattern, más que en la solución y sus variantes como ocurre con aquellos del catálogo. Por esto incluyen en la especificación de cada PD una sección de precondiciones y otra de restricciones que actúan sobre las

[1] Se ha elegido no traducir este término al castellano para no inducir a la confusión con otros significados de la palabra "patrón", la cual sería su traducción.

posibles soluciones al problema que se plantea. La denominación de "generativos" surge de que los patterns así planteados *determinan la arquitectura que se construye en base a ellos*, como contraposición a los patterns que se encuentran en un sistema ya construido, para su documentación.

Otra cosa importante ha destacar es que el *nombre* de un PD debe transmitir la esencia del mismo. Su elección es vital, ya que identificará al pattern en un nuevo vocabulario de más alto nivel, ha ser incorporado y compartido entre los desarrolladores, lo que permite comunicar toda una decisión de diseño con sólo un nombre.

1.4.6. Patterns y frameworks

Los PsD están siendo aplicados además a la documentación de grandes arquitecturas. Johnson comenta en [Johnson92] cómo pueden ser utilizados exitosamente en la documentación de frameworks, como un conjunto que se puede denominar *"lenguaje de patterns"*. En su trabajo, Johnson explica la dificultad de documentar frameworks, por tratarse no sólo de código, sino de diseños reusables, mezcla entre "lo concreto" y "lo abstracto", y por ser necesario explicar las decisiones que expertos han tomado para modelizar el dominio en cuestión, en forma suficientemente clara y abstracta para que los usuarios no expertos puedan utilizarlo sin llegar a profundizar en sus detalles. Como bien apunta Booch [Booch92]: "el framework más absolutamente elegante nunca será reusado a menos que el costo de entenderlo y usar sus abstracciones sea menor que el costo que perciba el programador de escribirlo desde cero".

Cabe destacar las diferencias existentes entre PsD y frameworks [Gamma+95]:

- los frameworks siempre modelan un dominio particular, mientras los PsD generalmente pueden utilizarse en distintos dominios;

- los frameworks son macroarquitecturas, que generalmente contienen varios PsD;

- los frameworks traen consigo una biblioteca de clases concretas, de manera tal de poder ser instanciados directamente. Sólo ejemplos de PsD pueden ser utilizados en el código.

Los frameworks muy raramente documentan el por qué de una decisión de diseño, mientras que los PsD *deben* hacerlo. Es por esto que muchas veces resulta difícil entender el framework que se quiere utilizar y qué clases concretas conviene elegir. Este es uno de los casos en que los PsD pueden ser utilizados en la documentación de frameworks: ***para plantear los pasos a seguir en su instanciación o uso***. Con respecto a esto, un pattern resulta más que una "receta", pues más allá de por ejemplo decir qué componente de una jerarquía abstracta conviene usar en cada caso, provee la razón de cada elección, y sus consecuencias. Estas razones involucran, en el framework que aquí se presenta, cuestiones de estilo en el diseño de aplicaciones hipermedia, dependiendo del concepto de *buena calidad* para aplicaciones en este dominio. Según C. Alexander, los patterns son los más indicados para describir lo que él llama *"calidad sin nombre"* en un diseño.

Por otro lado, en la definición de los elementos que constituyen un pattern de diseño encontramos la solución recurrente que se plantea, y en esa solución es imprescindible describir la interacción entre los componentes que el pattern propone. De modo que lo que constituye una carencia en la documentación de frameworks es directamente provisto por los PsD. Entonces, un segundo uso de patterns consiste en la ***documentación de las interacciones entre componentes que definen el "esqueleto de un framework".***

Nuestra experiencia en esta cuestión nos mostró que muchas de las decisiones que fueron tomadas en la definición de las interacciones en el framework, seguían las soluciones planteadas por patterns de aplicación general, como "Observer", "Command", "Prototype", "Template Method" [Gamma+95] y "Type Object" [Johnson+97]. Más aún, muchas otras tuvieron que adaptarse al dominio particular de Hipermedia, por lo que surgieron nuevas variaciones a los patterns allí descriptos, con sus propias fuerzas interactuantes que dan resultado a la solución, como el "Navigation Strategy" (variación del pattern "Strategy" que incluye al "Abstract Factory" y resuelve el destino de un link) y el "Navigation Observer" (variación del Observer para mantener la historia de nodos visitados) [Rossi+96a]. Veremos esto en detalle en el Capítulo 2, Sección 2.2.3 y 2.2.4.

Como tercer uso propuesto de la utilización de patterns para documentar un framework, tenemos la ***especificación de las relaciones que el framework mantiene con el resto del sistema.*** Como dijimos previamente, este aspecto también es muy importante, y la definición de contratos como la propone [Helm+90] resulta muy útil. Aún así, muchas veces un conjunto de contratos puede no resultar suficiente para entender la interacción. Es entonces cuando los contratos deberán además estar incluidos dentro de patterns que expliquen sus implicancias.

1.5. Clasificación de las aplicaciones que podrán beneficiarse de este trabajo

A pesar de que la FH puede resultar muy útil, no todos los sistemas de información permitirán una integración total de esta funcionalidad con la suya propia. Por este motivo hemos identificado tres subconjuntos de aplicaciones, de acuerdo a las *diferentes necesidades de soporte de FH* que puede resultar *factible* y *significativo* para las mismas [Garrido+96a].

Los subconjuntos se identifican como:

a) ***Aplicaciones basadas en los datos:*** éstas son aplicaciones pasivas, donde las características de hipermedia son fácilmente introducidas mediante el mapeo directo de cada concepto a nodos, y las relaciones entre conceptos a links. En el campo de orientación a objetos, esto se traduce a mapear clases a clases de nodos y relaciones entre clases a clases de links, permitiendo así una definición significativa de los componentes de la hipermedia y la completa automatización de su instanciación.

El conjunto entero de características de hipermedia puede ser agregado a este tipo de aplicaciones estáticas en las cuales ningún conflicto puede surgir con la propia funcionalidad de la aplicación.

b) ***Aplicaciones híbridas:*** tienen un aspecto de comportamiento asociado a cada ítem de información de sus componentes. En este caso, cada nodo puede ser considerado como un agente activo, que permite disparar ciertas acciones relacionadas con los datos (o con la clase de objetos) que está actualmente mapeando. Todas o muchas de las características de navegación pueden ser agregadas a estas aplicaciones, a pesar de que se debe tener en cuenta el dinamismo de las mismas. Nuestro trabajo está principalmente orientado a este tipo de aplicaciones porque son las predominantes en el campo de la orientación a objetos.

c) ***Aplicaciones basadas en el comportamiento:*** estas son aplicaciones con una interface basada en "browsers" para controlar la activación de la funcionalidad propia de la aplicación. Considere como ejemplo un sistema de información de aerolíneas, con una interface donde cada parte de la ventana permite cierto tipo de filtrado sobre el conjunto de vuelos (como aerolínea, origen, destino, costo, etc.). La navegación a través de los datos de esta aplicación parece marginal, porque la aplicación está focalizada en sus cómputos que la hacen altamente dinámica, contrariamente a la pasividad de la hipermedia. De todas formas, el framework aquí desarrollado permite al diseñador conservar las herramientas de control de la aplicación, que son importantes para su manejo de transacciones y a las cuales el usuario está acostumbrado, y ofrece una facilidad de navegación a los resultados de las computaciones o a cada entidad involucrada, además de la posibilidad de realizar anotaciones.

2. Desarrollo del trabajo

Este capítulo contiene todo el desarrollo realizado en este trabajo de investigación. En la primera sección se detalla cada concepto que resultó modelado en este trabajo, planteando su razón de ser a partir de la/las metodologías o modelos formales que lo proponen o involucran. Luego se describen los conceptos que se agregaron como extensión, para cubrir la FH no manejada en ningún modelo.

La segunda sección presenta el diseño de la arquitectura desarrollada, en términos de las jerarquías de clases principales en el modelo de objetos, la biblioteca de componentes concretos que acompañan a las clases abstractas del framework para su directa instanciación, y los patterns de diseño utilizados.

Una tercera sección involucra cuestiones de implementación del framework en términos del lenguaje utilizado y las extensiones realizadas.

2.1. Obtención del modelo esencial del dominio

En esta sección se propone un modelo de Hipermedia en el intento de unificar los conceptos que aparecen en los modelos y metodologías descriptos en el capítulo anterior. Aquí se especifica el significado con el que se considera cada componente del modelo, comparándolo con la acepción otorgada por los trabajos anteriores. En una segunda parte se extiende el modelo unificado con características que soporten FH.

La Sección 2.2 retoma cada uno de los conceptos aquí presentados, y describe en detalle los componentes del diseño OO que los modelan en la arquitectura desarrollada.

2.1.1. Modelo unificado de Hipermedia derivado de los modelos y metodologías del dominio

2.1.1.1. Niveles de definición de los componentes

Como mencionamos en la Sección 1.4.3. "Funcionalidad de hipermedia", mezclar los aspectos de interface y navegación dentro de la aplicación a ser extendida derivaría en un código obscuro, inmantenible y poco o nada reusable. Uno de los principios de la reusabilidad de grandes arquitecturas es la separación de las mismas en capas independientes y cohesivas en sí mismas. Es por esto que hemos definido tres niveles de abstracción para nuestra arquitectura: el *nivel de objetos*, el *nivel de hipermedia* y el *nivel visual.*

Una separación en capas muy similar es la definida en el modelo de Dexter. Además OOHDM propone actividades de desarrollo que bien podrían ser implementadas con cada nivel de esta arquitectura.

A continuación describimos cada nivel, comparándolo con ambos modelos.

- *Nivel de objetos.*
  Es el nivel de la aplicación subyacente, es decir que estará formado por los componentes involucrados en el modelo de esta aplicación. Esta capa será la responsable de proveer la información a ser mostrada, las relaciones a ser navegadas, y el comportamiento a ser extraído por el nivel de hipermedia.

  Este nivel podría compararse con la *capa intra-componente* del modelo de Dexter, ya que ésta incluye el contenido y estructura de cada nodo. La diferencia reside en que Dexter no considera un diseño de objetos sino que se maneja con una base de datos, con toda la pérdida de semántica que esto implica, y con la inexistencia de comportamiento asociado.

  Si el usuario del framework estuviera creando una aplicación hipermedia desde cero, debería modelar previamente el dominio, diseñando este nivel de la misma forma que propone OOHDM en su actividad de *diseño del modelo conceptual.*

- *Nivel de hipermedia.*

  En este nivel fueron definidos los componentes del framework desarrollado, que detallaremos en la Sección 2.1.1.2. Estos componentes modelan e implementan en la forma más general posible los conceptos que definen una aplicación hipermedia. El usuario los instanciará en el momento de definir la visión navegacional sobre el nivel de objetos.

  La *capa de almacenamiento* de Dexter describe de la misma forma los componentes del modelo de hipermedia. Así también, OOHDM propone una segunda actividad, *diseño navegacional*, en la que el diseñador define las clases de componentes navegacionales que mapean las del modelo conceptual, lo que se traduciría a instanciar las clases que nosotros definimos en este nivel.

- *Nivel visual.*

  Aquí se manejan los aspectos de interface, es decir, se especifica la apariencia visual de cada componente del nivel anterior, y se controla la interacción con el mundo exterior.

  El OO-Navigator utiliza otro framework para este nivel: el "Model View Controller" (MVC) [Krasner+88], el cual se ha extendido para soportar objetos de interface con capacidad de navegación, o "hypermedia-aware widgets". A pesar de que el framework de hipermedia es independiente de este nivel y podría ser usado con otra metáfora de interface, elegimos el MVC por estar siendo referenciado como el mejor modelo de interface actual, y ser prevaleciente en los sistemas OO. Detalles del nivel visual pueden encontrarse en la Sección 2.3.

  De la misma forma Dexter propone una *capa ejecutable* encargada de la interacción con el usuario. La metodología OOHDM también separa los aspectos de interface en una tercera actividad llamada *diseño abstracto de interface*.

El framework no sólo involucra los componentes del segundo nivel sino también la comunicación necesaria entre él y los restantes niveles. A continuación detallamos los componentes del segundo nivel.

2.1.1.2. Componentes del nivel de hipermedia

En esta sección se identifican los conceptos que forman *la base de nuestro modelo unificado de FH*. Por cada concepto se especifica la acepción considerada, modelo o metodología en la que fue inspirado, y contraste con las otras acepciones cuando fuera necesario. Así definiremos un *vocabulario común* en el que nos basaremos para el resto del trabajo.

Para presentar los distintos conceptos se utilizará como ejemplo un sistema de información para una biblioteca, que modela los conceptos usuales de Libro, Autor, Lector, Préstamo, Reserva, etc.

- **Hipermedia**

  Consideramos hipermedia como la extensión multimedial al hipertexto. Definimos hipertexto como una red de nodos y links que permite una estructuración no-lineal de la información, y la posibilidad de navegar a través de la red con o sin un orden predeterminado, según la necesidad del usuario. Todos los modelos coinciden en esta definición.

- **Nodo**

  Los nodos son piezas de información autocontenida. En esta definición coinciden todos los modelos aunque algunos le den distinta denominación (*componentes base* para Dexter, *entidades* en HDM).

  Los nodos pueden categorizarse de acuerdo a su estructura interna en:

  - *Nodos Atómicos:*

    Son los componentes más primitivos de esta capa. Este mismo concepto lo encontramos en el modelo de Dexter, en el NCM (con el nombre de *terminales*), y son implementados en DHM como *AtomComponents*. En PPFS, HDM y OOHDM se considera una subdivisión interior de los nodos en *slots* o *unidades* o *atributos* respectivamente. A nivel de diseño nosotros no consideramos esta subdivisión al tratar a los nodos atómicos como una unidad, aunque luego es necesario desmembrarlos para implementar sus vistas e interfaces, de la misma forma que OOHDM.

    En el ejemplo de la biblioteca podríamos considerar tener un nodo atómico representando la Biblioteca, un nodo por cada Lector y uno por cada Autor.

  - *Nodos Compuestos:*

    Están formados por otros nodos arreglados en una *jerarquía de composición*, o "parte-de". Esto introduce una semántica alternativa de organización de la información y de navegación a través de la misma.

    Retornando al ejemplo, los nodos Libro serían nodos compuestos, que muestran su propia información y agregan a todos sus capítulos.

    Dexter define con el mismo nombre a los nodos que contienen a otros organizados en forma de DAG, aunque no le otorga una semántica de composición sino que sólo los introduce para elevar estas estructuras jerárquicas al mismo nivel de los nodos y links, y proveer esta organización más allá de una red pura. La necesidad de este tipo de nodos fue reconocida desde los primeros grandes sistemas de hipertexto, como NoteCards, donde se proveía de *FileBoxes* para una mejor estructuración de los nodos y por lo tanto minimizar la desorientación del lector.

    HDM define a este tipo de nodos como componentes básicos de su modelo, con el nombre de *entidades*. Esto implica que no permite crear lo que nosotros denominamos *nodos atómicos* si no es como pertenecientes a una entidad. Consideramos que esta es una restricción que no refleja la realidad.

- *Colecciones:*

  Son agrupaciones de nodos con la semántica de conjuntos. Nuestro modelo los diferencia de los nodos compuestos explícitamente, y los eleva al mismo nivel, ya que los considera con alto potencial no sólo para la organización de los nodos, particionándolos en grupos semánticos consistentes, sino para agrupar resultados de consultas a la hipermedia, y sobre todo para la definición de nuevas metáforas de navegación (como sería el concepto de "contexto de navegación" que propone OOHDM). Los miembros de una colección pueden pertenecer al mismo tipo, formando una colección homogénea, o ser de distinto tipo, a la que se denomina heterogénea. Además las colecciones pueden anidarse a distinto nivel.

  El modelo que define el concepto de *Colección* de una forma muy similar a la planteada aquí es el de Garzotto *et al.* En él se proponen seis métodos distintos de definición de los miembros de una colección, como se describen en el capítulo anterior.

  Toda colección tiene un *nodo asociado* que ayuda al lector a entender el contenido de la colección, proveyendo información adicional para la misma. Gazotto propone que el nodo asociado contenga también el mecanismo de acceso específico a los miembros de la colección, como por ejemplo por Indice o Tour Guiado. Esto implica que este nodo contenga los links necesarios hacia los componentes (desde él hacia todos en el caso de Indice, o desde él hacia el primero del Tour). Contrariamente, nosotros consideramos que la forma de acceder a los componentes de una colección o de navegar por los mismos, se debe mantener como información separada de la colección en sí. Es decir que los links necesarios para recorrer la colección se mantendrán en un objeto aparte, que llamamos *estructura de acceso* (ver página 45).

  En el trabajo [Zellweger89] se realiza una separación semejante: denomina *"entries"* a los ítems que proveen el contenido de un camino, mientras que el *"path"* o camino provee la secuencia de acceso. Comparando con lo que aquí se presenta, la colección tendría las "entries", y la estructura de acceso conoce el "path".

  El trabajo de Garzotto *et al.* también discute el uso de colecciones para especificar estrategias de sincronización que se presentan con componentes multimediales. Para esto se propone que el nodo asociado a la colección tenga información relacionada con la forma de presentar y coordinar los distintos miembros. Esto es obviamente muy prometedor, pero requiere todo un estudio aparte no cubierto en este trabajo.

  La metodología OOHDM propone el concepto de *contextos de navegación* con un propósito muy similar al que aquí se definen las colecciones: para organización del conjunto de nodos, y para la incorporación de nuevas metáforas de navegación. Aunque el framework de hipermedia no provee directamente todo lo que OOHDM propone para la definición de los contextos de navegación, las colecciones serían el punto de partida para implementarlos.

  Para la biblioteca sería útil definir, por ejemplo, una colección con todos los libros de cierta época o movimiento, otra colección con los autores de esos libros, una colección de lectores morosos, etc.

- **Link**

Los links representan la *relación* entre dos o más nodos. Conforman los arcos de la red hipermedial. La característica distintiva de un link es que puede ser *"navegado"*, es decir, ser activado para pasar de un nodo a otro/s que están relacionados mediante ese link. Es así como el usuario irá formando el recorrido deseado con o sin un orden predefinido. Este concepto, así como el de nodo, es compartido por todos los modelos.

En el ejemplo de la biblioteca se podrían definir links entre el libro y su autor, y viceversa, entre un artículo y los otros a los que referencia, entre un tema y los materiales bibliográficos que lo cubran, etc.

Los links son accedidos a través de *anclas* contenidas en la información que el nodo presenta, o a través de una *estructura de acceso*. Hablaremos de esos otros componentes en sus ítems correspondientes. Por otro lado un link llega a un *destino* que en nuestro modelo será un nodo completo o un conjunto de nodos. Una posible extensión sería agregar anclas en los destinos para poder arribar a una parte de un nodo y no al nodo entero (como propone Dexter), aunque esa característica no fue considerada dentro del modelo base por no ser generalmente usada. Lange también presenta la idea de *links de segundo orden*, es decir aquellos que tienen como destino otro link, aunque no justifica claramente la necesidad de los mismos y por consiguiente no lo consideramos un concepto relevante.

Cuando un link es activado, éste debe producir la navegación hacia el nodo destino. Muchos de los modelos sólo incluyen la definición estática del destino de un link. En este trabajo hemos considerado otras variantes:

- *cálculo del destino en forma dinámica*: el destino se define mediante un algoritmo a ser evaluado o resuelto en el momento de la activación del link. La necesidad de calcular el destino dinámicamente ha sido reconocida en varios trabajos recientes [Agosti+95, Ashman+96] que exponen como razón principal la noción de "asimilación instantánea" de cambios en los datos subyacentes.

- *dangling links:* se denomina de esta forma a aquellos links que no tienen un destino especificado. En la extensión al modelo de Dexter propuesta por DHM se discute la importancia de otorgar la flexibilidad de especificar en forma tardía el destino de un link. En [Kelly+96] también se presenta esta necesidad en un ambiente CASE.

Los links pueden tener atributos (como expresa OOHDM). Un ejemplo sería el significado de la relación.

- **Clase de nodo y clase de link**

Simulando la forma en que las clases factorizan la estructura y comportamiento de sus instancias en el paradigma OO, se definen los conceptos de *clase de nodo* y *clase de link*. Una clase de nodo define las propiedades comunes de cierto grupo de nodos que contienen los mismos atributos (aunque cada uno tendrá un valor distinto para los mismos), se muestran de la misma forma y se relacionan con otros nodos de igual manera. Ejemplos de clases de nodos serían "Libro", "Lector", "Autor". Este concepto es el mismo que se define bajo el mismo nombre en OOHDM y con el nombre de *"Entity Type"* en HDM. Aunque otros modelos no lo definan explícitamente, la mayoría lo asume como un concepto válido.

Cabe destacar que no hacemos una jerarquía de clases de nodos según abstracciones comunes entre ellas. La jerarquía se mantiene en el nivel de objetos.

Una clase de link agrupa aquellos cuyo origen pertenece a la misma clase de nodos, al igual que su destino. Si consideramos atributos en los links, aquellos pertenecientes a la misma clase mantienen los mismos atributos. Este mismo concepto ha sido definido en OOHDM, y con el nombre de *"Application Link Type"* en HDM. Ejemplos serían "Prestado" entre "Libro" y "Lector", o "Escribió" entre "Autor" y "Libro".

Haake y Streitz han reconocido en su artículo [Haake+94] la importancia de la clasificación de nodos y links con el propósito de comunicar los objetivos más efectivamente, expresando mayor semántica. Esto se debe por un lado, a que el autor puede organizar mejor la información a presentar, y por otro lado a que el lector puede mantener la línea de lectura o el contexto (como lo traduce OOHDM), sabiendo qué clase de nodo se encuentra visitando dentro de esa organización de la red hipermedial. De todos modos también consideran la importancia de la flexibilidad en la distribución del conocimiento para los lectores, y concordando con ellos es que no restringimos a que todos los nodos o links pertenezcan a una clase, sino sólo cuando es conveniente.

- **Ancla de link**

Para permitir que los links tengan como origen una "zona" del nodo y no el nodo entero, aparece el concepto de *ancla de link*. Este concepto surgió desde los primeros sistemas hipermediales, permitiendo clickear sobre una palabra *(hotword)* resaltada de alguna forma para indicar la existencia de un link. En nuestra interpretación, un ancla está conectada con sólo un link, pero el mismo link puede ser activado por distintas anclas, (que tal vez parten del mismo concepto pero en distintas representaciones del mismo).

DHM propone tres tipos de anclas: *whole-component anchor*, o anclas que representan el componente todo; *marked anchors*, o anclas que de alguna manera se hacen visibles en una parte del contenido del nodo, y *unmarked anchors*, cuya ubicación no se especifica y debe ser calculada o buscada. Este trabajo considera válido ofrecer las distintas posibilidades de definición de anclas, aunque no se ha subclasificado el concepto de ancla por tal motivo. Esto se debe a que el objeto de interface al que el ancla se adhiere es independiente y pertenece a la capa de interface. Este objeto es el que puede aparecer como un botón aparte, o dentro de cierta información, ya sea definido en forma fija o calculado en el momento de abrir la interface del nodo, cubriendo las tres posibilidades.

- **Destino de link**

El destino del link puede ser singular o múltiple. En el primer caso el destino contiene al nodo en cuestión, más cierta información referida a la forma en que va a ser presentado de acuerdo al link por el que se llega a él (veremos esto más adelante con el concepto de vista de nodo). En el segundo caso en lugar de un nodo, el destino contiene a un conjunto de nodos.

La forma de mostrar un destino múltiple coincide en la mayoría de los sistemas en proveer una estructura de acceso intermedia para seleccionar un nodo particular,

aunque los modelos no lo especifican por considerarse una decisión de implementación.

El modelo de Dexter también propone la especificación de anclas en el destino, que nosotros no hemos considerado en el modelo base, aunque sería una posible extensión.

- **Contexto**

El concepto de *contexto* que hemos adoptado es básicamente el definido en el NCM: permite organizar la información adaptándola a diferentes perfiles de usuarios o lectores. A diferencia de aquel modelo, definimos los contextos en forma global a toda la hipermedia, planteando desde un comienzo los distintos tipos de usuarios para los que la aplicación será desarrollada.

En NCM se definen *"nodos contextuales"* que agrupan nodos teminales o contextuales (en jerarquías o conjuntos) y links entre ellos, definiendo un *"hiperdocumento"*. Es importante destacar que cada nodo puede pertenecer a varios nodos de contexto, pero los links son internos al nodo de contexto al que pertenecen sus puntas, no pudiendo conectar entonces nodos de distintos contextos. Esta es una restricción que no consideramos necesaria.

OOHDM define *"contextos de navegación"* en forma muy similar a la planteada por NCM. El concepto de contexto que ese modelo usa no tiene como propósito principal orientarse a distintos perfiles de usuario como aquí se presenta, sino organizar la navegación en estructuras que pertenecen a un mismo tema, o tienen que ver con lo mismo. Para poder implementar este tipo de construcción nosotros proponemos las *colecciones*, como fue planteado en el inciso homónimo.

- **Vista de nodo**

Una vez planteados los contextos, el nodo necesita conocer cómo presentarse en un contexto determinado, y esa información es mantenida por una de sus *vistas*. Una vista de nodo puede graficarse como una de las "caras" con las que el nodo va a mostrarse, si vemos al nodo como un prisma (ver Figura 4). Esta cara muestra ciertos atributos del nodo y permite navegar ciertas relaciones mostrando algunas de sus anclas, según el perfil de usuario para el cual fue definida.

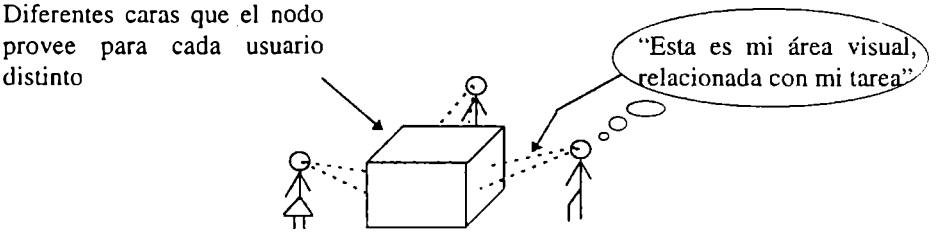


Figura 4: Vistas de nodo sobre un objeto

A medida que se produce la navegación, los nodos se irán mostrando con la vista que corresponde al *contexto actual*. NCM define *perspectivas* de nodos refiriéndose a la secuencia de nodos contextuales a través de la cual un nodo es observado. Al permitir que un nodo pertenezca a varios hiperdocumentos y sea anidado a diferente nivel, se

permite definir diferentes vistas del mismo, basándose en las distintas clases de usuarios. Esa "secuencia" por la que se llega a una vista en nuestro caso es el contexto actual.

- **Representación de nodo**

Así como una vista de nodo define qué atributos del mismo van a ser presentados en un contexto determinado, una *representación* del nodo, en una vista particular, define cómo van a mostrarse esos atributos. De esta forma un mismo lector podrá observar la misma información con diferentes medios (texto, gráfico, imagen, etc.).

NCM define el concepto de Representación de la misma forma, y HDM y OOHDM lo proponen con el nombre de *"Perspectiva"*.

La representación con la que un nodo puede elegir mostrarse la consideramos dependiente de la forma en la que se navegó hacia él, y por lo tanto será información que el link contenga en su destino. Es decir que el destino del link está formado por el nodo y la representación con la que debe mostrarse, en el contexto actual. De la misma forma que NCM, consideramos que un nodo puede indicar directamente una representación por defecto en su clase, de forma de utilizarla en caso de no estar especificada en el destino del link.

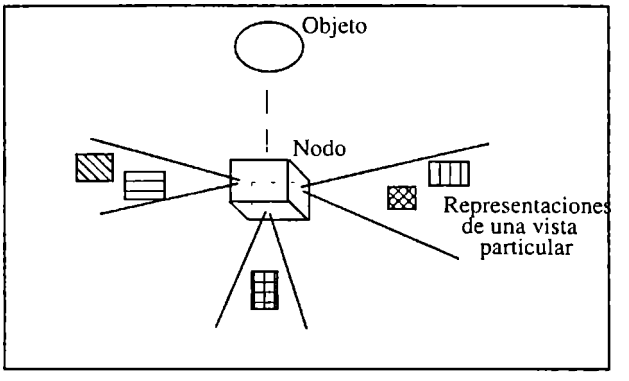


Figura 5: Vistas de nodo y representaciones

- **Estructura de acceso**

Las *estructuras de acceso* proveen distintas formas de organizar una colección de nodos y permiten búsquedas más rápidas, con el propósito de reducir el problema de "information overhead" que la navegación hipermedial puede producir. Cabe destacar la separación que planteamos entre la colección en sí y la estructura de acceso que la muestra. Esta separación no se hace implícita en ninguno de los modelos estudiados.

Nuestro modelo propone tres diferentes tipos de estructuras de acceso: Indice, Secuencia y Tour Guiado. Un *índice* presenta en forma de tabla cierto atributo identificatorio de una colección de nodos y permite la selección de un nodo particular para provocar la navegación hacia él. Un índice puede ser definido como parte de un nodo o como estructura separada. Un *tour guiado* define una secuencia ordenada sobre una colección de nodos, y permite recorrerlos mediante links "anterior" y "siguiente" según esa ordenación. Las estructuras de acceso son discutidas en detalle en [Zellweger89, Trigg88], e implementados en DHM.

Las estructuras de acceso se caracterizan por tres atributos, de acuerdo con el modelo presentado en OOHDM: el conjunto de nodos "target", el conjunto de selectores o atributos de los nodos que serán mostrados en la estructura, y un predicado lógico sobre los nodos "target" que permite la definición de estructuras de acceso condicionales, como se propone en la metodología RM. En ese trabajo se plantean dos propósitos del uso de estas estructuras condicionales: primero, para tener un mayor control sobre los nodos accesibles desde la estructura, y segundo, para soportar el cómputo dinámico de los mismos.

Ejemplos de estas estructuras serían los índices que muestran los préstamos y reservas de un lector de la biblioteca, permitiendo la navegación hacia los libros involucrados, o un tour guiado para navegar a través de los libros de un mismo autor.

- **Historia de navegación**

La mayor parte de las aplicaciones hipermedia actuales permiten registrar, de una forma perceptible al usuario, el estado de la navegación. Esto se lleva a cabo en la *historia de navegación*. A medida que el usuario recorre el espacio hipermedial, la historia se va modificando automáticamente, capturando los nodos o links que se van visitando.

2.1.2. *Extensión del modelo base*

El propósito principal de la extensión realizada sobre el modelo base, es el de soportar la conexión con una aplicación subyacente a ser extendida con FH. Detallaremos entonces los conceptos agregados o extendidos y la conexión necesaria entre el nivel de objetos y el nivel de hipermedia.

De los modelos estudiados, el DHM es el único que plantea la integración de estructuras de hipermedia con terceras aplicaciones que contienen los datos. En [Grønbaek+94a] se puntualiza que Dexter deja una puerta abierta hacia este tipo de extensiones gracias a su división en capas, donde la capa intra-componente sería la de la aplicación. Pero también expone sus debilidades: primero, no se distingue entre componentes cuyos contenidos son manejados por la aplicación o por la hipermedia, y segundo no se especifica qué sucede con documentos de la aplicación que tengan una estructura interna, la forma de anclarlos o estructurarlos. Ante esto, DHM ofrece al usuario la posibilidad de contener apuntadores al archivo donde se encuentran los datos, en vez de contener los datos directamente, y mantener tablas de claves de atributos y referencias a sus valores, para la estructuración de los componentes. Es evidente que esta aproximación no mejora mucho la situación; el hecho de poder mantener punteros a bases de datos no implica que se está extendiendo una *aplicación*, con toda la semántica y comportamiento que esto último implica.

Por lo tanto, no existe a nuestro conocimiento ningún modelo que plantee la completa integración de un sistema de información OO con FH, permitiendo retro-adecuar las aplicaciones existentes. Es por eso que se ha desarrollado en este trabajo la extensión que seguidamente se presenta.

- **Nodos como observadores de objetos**

  Para permitir una extensión de aplicaciones orientadas a objetos con FH, es necesario permitir que los nodos sean 'vistas' de los objetos de la aplicación, y que los links muestren relaciones entre objetos, o mapeen objetos asociativos. Esto significa que existirá un tipo de nodo que actúe como un "template" vacío, que sólo sepa como ser navegado, pero que requiera ser conectado con objetos de la aplicación al ser instanciado. Esto es lo que permitirá además preservar el comportamiento de la aplicación, proporcionando caminos de comunicación para notificar a los objetos sobre la ocurrencia de eventos externos.

  Se ha realizado entonces una categorización de acuerdo a cuán estrecha sea la relación entre los nodos y los componentes de la aplicación:

  - *Mapeo directo*

    El diseñador debería seleccionar qué componentes de la aplicación subyacente quiere observar como nodos en la hipermedia, es decir, de cuáles componentes va a mostrar información o va a activar servicios, y además va a 'linkear' en la aplicación hipermedia resultante. Una explicación más extensa de cómo se realiza este mapeo puede encontrarse en el Capítulo 3. En el ejemplo de la biblioteca podríamos pensar en observar como nodos a los objetos lectores, a los objetos autores, y a los objetos libros, si estos formaban parte del diseño de la aplicación.

OOHDM realiza un mapeo semejante al que proponemos aquí, en la actividad de traducir del 'Modelo Conceptual' al 'Modelo Navegacional'.

Un nodo atómico no está restringido a mapear sólo un objeto de la aplicación, sino que puede observar aspectos de varios objetos íntimamente relacionados, o que no tendría sentido mostrar separadamente. Este sería el caso de que por ejemplo la biblioteca contara con distintas salas donde se ubican los libros. Supongamos que no nos interesa mostrar las salas en nodos independientes, pero en el nodo que mapea un libro queremos mostrar el nombre de la sala y eventualmente su ubicación junto con los otros datos del libro. Lo importante para destacar es que el nodo no deja de ser atómico por el hecho de mostrar información que obtiene de distintos objetos; lo que para la aplicación eran objetos separados no implica que en la hipermedia tenga sentido mostrarlos en distintos nodos. OOHDM plantea este mismo tipo de mapeo.

De la misma manera que permitimos formar *nodos atómicos* observando objetos de la aplicación, extendemos el concepto a los *nodos compuestos*. Estos observarán toda una estructura de agregación, capturando esa organización jerárquica del modelo subyacente. Los nodos compuestos han sido además especializados en *Nodo Compuesto Dependiente*, en los cuales la existencia de los nodos agregados depende de la del nodo padre.

- *Mapeo de interface*

Cuando la aplicación subyacente contiene su propia interface, con la cual el usuario se siente cómodo, y desde la cual se controla el comportamiento de la aplicación, se provee un mapeo más 'suave', que permite observar a nivel hipermedial la interface del objeto, sin destruir la conexión existente entre el objeto y su interface, y permite que ambos controlen, como usualmente se produce, la muestra de la información y el disparo del comportamiento. Para poder crear links desde esa interface, se agrega sobre la misma una capa que sólo contiene anclas de links; esa capa del nivel hipermedial la llamamos *nodo navegador*. Estas anclas se muestran bajo demanda, cuando el usuario explícitamente desea navegar.

Supongamos que nuestra aplicación de biblioteca tuviera interfaces predefinidas para el préstamo y reserva de libros; tendrían posiblemente el aspecto de fichas para ingreso de los datos. Podríamos agregar FH de la siguiente manera: una vez que hemos arribado mediante navegación al libro deseado, las acciones de préstamo o reserva podrían dispararse mediante botones u opciones del menú, abriendo la ficha respectiva para ser llenada con los datos del lector únicamente (los del libro se llenarán automáticamente con los datos del nodo actual).

- *Sin mapeo. Nodos del nivel hipermedial*

No todos los nodos necesitan estrictamente mapear uno o varios objetos de la aplicación. Por el contrario, pueden surgir nodos en el nivel de hipermedia, cuya información sólo tiene sentido en este nivel, como podría ocurrir con un nodo de presentación o carátula. En la mayoría de los casos estos nodos mostrarán datos multimediales, pues este tipo de información no aparece generalmente en el nivel de objetos de la aplicación.

Otro tipo de nodos que pertenece a este nivel es el de las colecciones, pues estas son agregaciones que se definen sólo a nivel hipermedial, para una mejor estructuración u organización de la navegación.

- **Links como observadores de objetos**

De la misma forma en que los nodos observan objetos del primer nivel, los links observan las relaciones entre esos objetos. Más aún, en el paradigma OO muchas veces se diseñan *objetos asociativos*, que tienen como finalidad mantener la información específica de la relación entre los objetos que conectan. Los links pueden mapear estos objetos, y obtener además el significado o atributos de los mismos, manteniendo una referencia hacia ellos.

Un ejemplo de objeto asociativo podría ser un Préstamo, que relaciona un Libro con un Lector, y además guarda información como el período de préstamo.

- **Acciones asociadas a la navegación**

Hemos agregado como atributo del link un bloque de acciones a ser ejecutadas en el momento de la activación del mismo, o navegación. Lo llamamos *poscondiciones*. Podríamos considerar como ejemplo el disparo de una animación en el nodo destino.

En el caso de links que mapean objetos asociativos, así como pueden obtener su significado de estos objetos, podrían obtener el bloque de poscondiciones.

- **Clases de nodos y clases de links como observadores de clases de objetos**

Ya comentamos anteriormente el propósito de clasificar los nodos y links. Con el agregado de la semántica de la aplicación subyacente, podemos considerar a las clases de nodos como "observadores" de clases de objetos de la aplicación. De esta forma, el diseñador podrá formar un modelo de clases de nodos y links mapeando clases de la aplicación, y esto conformará su modelo de hipermedia. Esto es exactamente lo que se realiza en la actividad de definición del Modelo Navegacional de OOHDM.

Tanto los nodos atómicos como los compuestos, que mapean objetos de la aplicación subyacente, tendrán una "clase de nodos" que los define. No la tendrán los nodos con mapeo de interface ni los del nivel hipermedial. Por el otro lado, no existirá clase de nodo que no observe una clase de la aplicación.

Se ha definido además una clasificación en *Clase de Nodo Atómico* y *Clase de Nodo Compuesto*. En este momento podemos notar por qué no hace falta una jerarquía de herencia entre clases de nodos con aspectos en común: ya estarán mapeando clases de la aplicación que sí se mantienen en una jerarquía de herencia teniendo en cuenta esos aspectos.

En [Bieber+95], un trabajo muy relacionado con el nuestro, se discute la valiosa facilidad que las clases de nodos y links proveen como base para la automatización en la creación de nodos y links individuales. Veremos más adelante que la implementación realizada en el framework permite justamente esta habilidad. El diseñador no tendrá que crear los nodos y links uno a uno, sino que al definir las clases de los mismos, *se crean automáticamente* mapeando cada uno de los objetos del primer nivel, instancias de la clase/s observada/s. Además se provee la

flexibilidad de *seleccionar* las instancias de la clase observada que se quieren mapear a nodos, mediante un predicado.

- **Creación dinámica de nodos y tardía de relaciones**

  El hecho de que el nivel de hipermedia esté observando el nivel de objetos de la aplicación, y más aún, en un ambiente dinámico donde se irán creando instancias en todo momento, implica que por cada nuevo objeto que sea instancia de una clase mapeada se debería crear el nodo respectivo, si corresponde. De la misma forma se deberían crear las relaciones pertinentes.

  Si profundizamos más en este sentido, enfatizando la completa integración de la aplicación con la FH, surge pensar que la navegación de un link podría provocar la creación de su destino. Veámoslo en un ejemplo: supongamos un ambiente CASE con posibilidad de diseñar componentes mediante tarjetas CRC [Beck+89], e implementarlos en algún lenguaje. Supongamos ahora que estamos diseñando un componente en el nodo que contiene su tarjeta CRC, y queremos navegar hacia el nodo que contiene su implementación, pero el componente aún no ha sido implementado. Es decir que el nodo respectivo no existe, pero sabemos a que clase de nodo va a pertenecer (a aquella que muestra la herramienta para implementar un componente). Se podría pensar entonces que navegar ese link desde el nodo de diseño hacia el de implementación provoque la creación de este último, y el o los objetos asociados en la aplicación. Esto se realiza definiendo en el destino del link que el mismo debe ser creado, con la clase de nodo que se especifica.

  Esto implica que ambos niveles están integrados de tal forma que cada uno puede ser el iniciador de la creación dinámica de componentes del otro.

- **Historia de links navegados**

  Todos los modelos sólo discuten mantener una historia de nodos visitados. Aquí también proponemos la posibilidad de registrar los links navegados, o ambas cosas, de manera que de ahora en más hablaremos de historia de componentes hipermediales en general.

2.2. Diseño del framework

Para documentar el diseño del framework se ha elegido presentar primeramente las jerarquías principales que forman su 'esqueleto', es decir, las abstracciones que lo delinean. En una segunda parte se detallan otras clases concretas que forman su biblioteca de componentes y que serán o podrán ser usadas al momento de instanciar el framework para crear una aplicación hipermedia. Cabe recordar que las clases aquí definidas forman parte del *nivel de hipermedia* de la arquitectura[2].

2.2.1. *Jerarquías principales*

Describiremos aquí los componentes abstractos principales del framework y las jerarquías que éstos forman.

2.2.1.1. Jerarquía de Componentes de la Hipermedia

- **Componente de hipermedia (HypermediaComponent)**

  Los componentes básicos que forman una hipermedia son los nodos, links, y las estructuras de acceso. Como consideramos a estos conjuntos con el mismo nivel de importancia, hemos definimos la clase abstracta HypermediaComponent, que los agrupa bajo el mismo nombre. El comportamiento factorizado en esta clase es el de activarse (`activate`) y el de registrarse (`record`), los que pueden ser englobados en una sola operación "plantilla" o *template method*, llamada `navigate`. Las dos operaciones mencionadas serán redefinidas en cada subclase, pero básicamente implican lo siguiente:

  - activarse: mostrar la información asociada en el caso de Nodo, o activar el destino en el caso de Link.

  - registrarse: avisar a la hipermedia que ha sido visitado, para que ésta decida en su historia de navegación si corresponde guardarlo para futuro acceso.

  El conocimiento factorizado en esta clase es la hipermedia a la que el componente pertenece, para poder llevar a cabo la registración.

  En la siguiente figura se muestra el primer nivel de esta jerarquía.

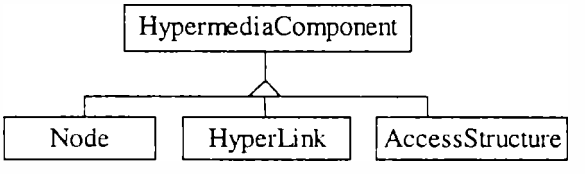


Figura 6. Primer y segundo nivel en la jerarquía de HypermediaComponent

[2] Se ha utilizado un font distinto para nombrar las clases (BookAntigua), mientras que las instancias se denotan con el font del documento (y a veces en minúscula para remarcar que son instancias). Los mensajes se escriben con font `Courier`.

- **Nodo (Node)**

Un nodo representa una pieza de información autocontenida. Los nodos en una hipermedia son activados a través de links y se muestran en una determinada interface. A su vez desde un nodo se puede navegar hacia otros por intermedio de la activación de las anclas de links que posee.

Consideramos que todo nodo tiene un nombre con el que identifica la información que posee. Este nombre será luego mostrado como su título en la interface.

Tanto sea que el nodo esté conectado con un objeto de la aplicación o no, debe mantener las anclas de links que permiten la navegación hipermedial por sí mismo, para no ensuciar u obscurecer el nivel de objetos. Es por esto que todo nodo mantiene las anclas de links que parten de él y que pueden ser accedidas desde cualquiera de sus vistas.

La jerarquía encabezada por esta clase está basada en la estructura interna de los nodos, pero en forma totalmente independiente del tipo de información que muestren o el comportamiento que disparen. Contrariamente, esa estructura se refiere a la categorización propuesta en la Sección 2.1.1.2, descripción de "Nodo", y la extensión desarrollada en 2.1.2, "Nodos como observadores de objetos".

La siguiente figura muestra la jerarquía completa partiendo de la clase abstracta Node.

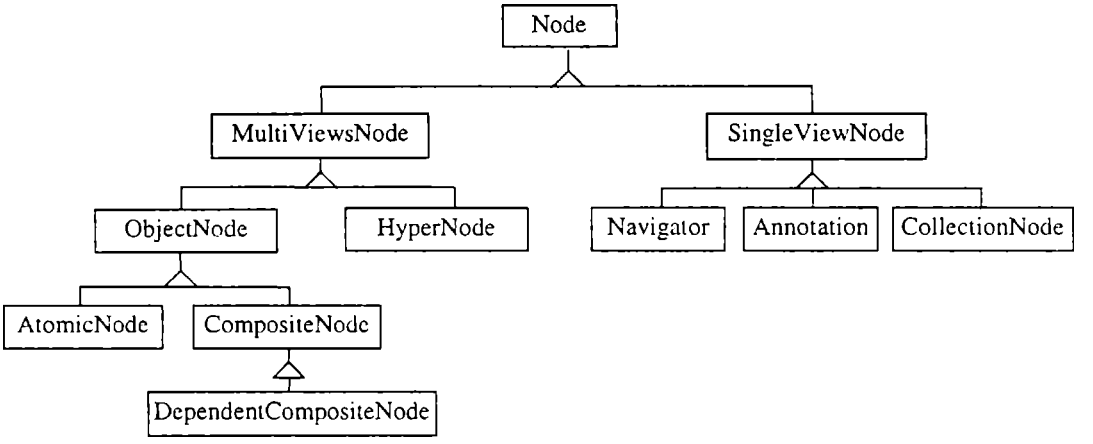


Figura 7. Jerarquía completa de Node

Como mencionamos anteriormente, un nodo puede mostrar distinta información o activar distinto comportamiento dependiendo del perfil del lector. Para lograr esto se permite que un nodo tenga distintas "vistas", una para cada contexto definido. De todos modos existen cierto tipo de nodos que sólo tendrán una "vista" posible como los "nodos-colección" (aunque de todos modos pueden tener distintas representaciones); éstos agrupan un conjunto de nodos (que a su vez podrán tener o no distintas vistas), hacia los que provee una estructura de acceso. Los datos que hacen al nodo-colección son idénticos para cualquier lector. Debido a esto definimos dos subclases de Node: una que agrega nodos con múltiples vistas, y otra que agrega los que sólo poseen una vista.

Las responsabilidades principales que esta clase factoriza son:

- mantener su nombre.

- mantener las anclas de links que parten de él.

- **Nodo de múltiples vistas (MultiViewsNode)**

Representa los nodos que muestran un "cara" distinta según el contexto, o perfil de usuario actual. Es decir que estos nodos constituyen una agregación de "vistas de nodo".

Para permitir factorizar datos comunes a todas las vistas sin tener que definir una subjerarquía para esto (lo que provocaría una explosión de clases), el nodo mismo provee acceso a los datos comunes. En el caso de nodos que mapean objetos, estos datos serán aspectos de esos objetos. De otra manera, serán contenedores de los valores. Los datos pueden tener anclas de links internas y un menú de acciones asociado.

Las responsabilidades principales que esta clase abstracta factoriza son:

- crear vistas de nodo, borrarlas y retornarlas;

- activar la vista de nodo correspondiente al contexto o perfil de usuario actual cuando el nodo es activado;

- agregar y manejar los datos a mostrar (retornarlos, cambiarles el valor, agregarles anclas de links, agregarles el menú asociado, borrarlos), en el nodo o en una vista particular; (esta responsabilidad es redefinida en las subclases);

- agregar y borrar anclas de una vista de nodo particular.

- **Nodo asociado a objetos de la aplicación (ObjectNode)**

Estos nodos son "espejos" de objetos de la aplicación. Son ellos los que permiten la incorporación de FH dentro de una aplicación existente, ya que permiten obtener los datos, mapear el comportamiento y las relaciones de los objetos que observan. Estos nodos se hacen dependientes de los objetos observados para permitir la actualización inmediata en ambos sentidos ante un posible cambio (el mecanismo de "dependencia" se explica en detalle en la Sección 2.3).

Todos los objectNodes responden a un "patrón" determinado por la clase del objeto que observan. Este patrón o tipo les dice los aspectos (mensajes) definidos en aquella clase que deberá utilizar para obtener y modificar los datos a mostrar y para disparar el comportamiento. Los colaboradores de los objectNodes que hacen de "patrón" o "tipo" de los mismos están representados por la clase NodeClass. Los objectNodes tienen que mantener una referencia explícita a su nodeClass. Como se dijo anteriormente, subclasificar ObjectNode por cada clase de cada aplicación a mapear no tendría sentido. Es por esto que se implementó un colaborador para la misma que simule a su clase dependiente del nivel de objetos. Este diseño responde al pattern Type-Object (ver Sección 2.2.3).

Como se dijo previamente, estos nodos también permiten el disparo de acciones a ser atendidas por los objetos de la aplicación. Este comportamiento del objeto a observar pueden ser especificado en el nodo mismo (con lo que podrá ser disparado desde todas las vistas) o en una vista particular.

La clase abstracta ObjectNode es subclasificada en AtomicNode y CompositeNode, que en síntesis observan un entidad atómica a nivel de hipermedia, o toda una jerarquía de composición de objetos, respectivamente.

Las responsabilidades abstraídas en la clase ObjectNode son:

- mantener la clase de nodo a la que responde según la clase del objeto observado;

- mantener las acciones que podrán ser disparadas desde el nodo, con el correspondiente aspecto o mensaje a enviar al objeto para delegarle su atención;

- redefinir el comportamiento para agregar datos a ser vistos en los nodos, adjuntando el objeto y el aspecto que responderá con ese dato.

- permitir agregar tanto acciones como datos a ser vistos en una vista del nodo particular.

- **Nodo atómico (AtomicNode)**

Un nodo atómico representa una entidad conceptual completa, que ya no puede ser desagregada o subdividida en nodos más pequeños. A pesar de esto, lo que se considera "atómico" en el nivel de hipermedia, no tiene por qué serlo en el nivel de objetos, ya que las consideraciones a tener en cuenta en sendos niveles son distintas. La atomicidad en el nivel de hipermedia estará dada por la información que tenga sentido mostrar en una misma ventana. Puede existir el caso de que esta información esté dividida en el modelo base entre varios objetos fuertemente relacionados, pero que no tenga sentido mostrarlos en la interface como entidades independientes. Es por esto que se permite que un nodo atómico mapee más de un objeto de la aplicación. Uno de ellos debe ser, sin embargo, el dominante o principal sobre los otros, de donde la entidad toma su nombre. Con este objeto el nodo se inicializa, y los otros se agregan posteriormente.

Cabe destacar que esta es una clase concreta, es decir que será instanciada en la creación de la aplicación.

Las instancias de esta clase tienen como responsabilidad, además de las heredadas:

- conocer y mantener los objetos que mapea.

- **Nodo compuesto (CompositeNode)**

Un nodo compuesto representa una entidad compleja, es decir una agregación de partes, capturando una organización jerárquica de la información, no basada en links sino en composición. Esta relación es extraída de la aplicación subyacente, mapeando un *objeto compuesto*, y todos los que representan sus partes. Los objetos "parte" ya deben estar mapeados en otros nodos, atómicos o compuestos, para ser luego agregados al nodo compuesto padre.

Los nodos "parte" de un nodo compuesto podrán ser agrupados de acuerdo a su clase de nodo, bajo cierto nombre. El nodo compuesto deberá proveer acceso no navegacional (no basado en link) para cada conjunto de partes.

Las responsabilidades que se suman entonces a la clase concreta CompositeNode son:

- mantener el objeto compuesto que se mapea de la aplicación;
- mantener los nodos que representan sus partes;
- proveer acceso a cada grupo de partes, y mostrarlo en el momento de abrirse;
- borrarse, sin borrar sus partes.

♦ **Nodo compuesto dependiente (DependentCompositeNode)**

Las instancias de esta clase difieren de las instancias de CompositeNode en que su existencia depende de la existencia de sus partes.

La responsabilidad que redefine es aquella de borrar un sub-nodo, eliminándose a sí misma de la hipermedia si se queda sin ningún sub-nodo.

♦ **Nodo de hipermedia (HyperNode)**

Los nodos del nivel hipermedial no están restringidos a reflejar únicamente objetos de una aplicación subyacente. Más aún, la hipermedia toda puede que no se base en una aplicación. Es por eso que se ha modelado la clase HyperNode para representar los nodos que no observan objetos de una aplicación sino que surgen en el nivel hipermedial, como sería el caso de un nodo de presentación. En un hyperNode se pueden agregan datos que son mantenidos por el mismo, aunque no podrá tener comportamiento.

Generalmente estos nodos mostrarán datos multimediales, puesto que este tipo de información raramente es mantenido por el modelo de objetos.

La responsabilidad redefinida para las instancias de HyperNode es:

- agregar datos y mantenerlos.

♦ **Nodo de vista única (SingleViewNode)**

Representa los nodos que muestran los mismos datos en todo contexto. Luego veremos el propósito específico de esta restricción cuando se explique cada una de sus subclases (CollectionNode, Navigator y Annotation).

Las responsabilidades principales abstraídas en la clase abstracta SingleViewNode son:

- crear y mantener las representaciones o modelos de la interface con las que puede mostrarse en su vista (responsabilidad redefinida en las subclases);
- activarse, abriendo la interface.

♦ **Nodo colección (CollectionNode)**

Como comentamos en la Sección 2.1.1.2., un nodo-colección representa un conjunto de nodos que comparten cierta característica o semántica, de manera tal que tenga sentido considerar al conjunto como una entidad en sí misma, proveyendo un acceso a sus elementos constituyentes.

Se proveen distintas formas de elegir o definir los nodos que pertenecerán al conjunto (basándonos como se explicó anteriormente en el modelo de Garzotto [Garzotto+94]):

- seleccionando de una clase de nodos aquellos nodos que cumplen cierto predicado (built-in+intensional);

- eligiendo los nodos manualmente (pick-up);

- dada una colección y una clase de link cuyos links parten de nodos de la colección, obtener la colección resultante de aplicar los links a la colección original (link-oriented);

- eligiendo la historia de nodos visitados (session-based);

- haciendo unión o intersección de colecciones ya definidas (set-oriented).

Para poder permitir estas posibles estrategias de definición del conjunto de nodos se ha definido una jerarquía aparte, la de CollectionStrategy, cada subclase soportando una de las anteriores formas de creación del conjunto.

Además del conjunto de nodos, un collectionNode debe conocer la estructura de acceso que proveerá a su componentes. Para eso se deberá indicar la estructura deseada, que estará dada por las subclases de AccessStructure.

Los nodos colección pueden tener información asociada, referente a su identidad como conjunto, para la que podrán proveerse distintas representaciones.

Las responsabilidades definidas para las instancias de CollectionNode son:

- proveer distintas estrategias para elegir el conjunto de nodos;

- mantener la estructura de acceso a los componentes;

- mantener la información que hace a la colección.

♦ **Nodo navegador (Navigator)**

Esta clase representa a los nodos que surgen de mapear interfaces gráficas definidas con el modelo de objetos. Una interface gráfica que se mapea a un navigator no será modificada en cuanto a su contenido o disposición de elementos, pero este mapeo implica que podrá ser considerada como un nodo más del espacio hipermedial, con todo lo que esto implica, a saber: que se puedan definir links hacia ella, que sobre ella puedan definirse anclas de links, que pueda ser incorporada a la historia de nodos visitados, etc. De aquí surge que estos nodos no tienen datos ni acciones asociadas, sólo guardan las anclas de links para que aquella interface y su modelo queden libres de información hipermedial y puedan por lo tanto seguir siendo usados fuera de la hipermedia. Otra característica de estos nodos es que sólo tendrán una representación posible.

En el momento de crear un nodo de esta clase se especifica la interface a la que se asocia. Automáticamente se crea entonces una representación para la ventana, que incorpora decoradores o "wrappers" (ver Sección 2.2.3.4) por sobre los widgets que muestran listas y los que muestran textos. Por el momento sólo este tipo de widgets pueden ser utilizados para la definición de anclas. En el caso de las listas, sobre éstas sólo se pueden definir "clases de links", es decir que cada elemento de la lista podrá ser utilizado como ancla para activar un link hacia un nodo-objeto, dependiendo del valor del ancla. Estos nodo-objeto consistentemente pertenecerán a sólo una clase de nodos, ya que resultaría incomprensible o confuso que de cada elemento de una misma lista se navegara a nodos que no tienen nada que ver entre sí. La clase de link

para la cual la lista servirá de índice deberá ser especificada en el momento de creación del nodo. En el caso de los textos, sobre ellos pueden definirse tanto links clasificados como no-clasificados.

Los nodos navigators están pensados para interfaces que se muestran sobre un único modelo, el cual debe especificarse también al momento de creación del nodo. Ejemplos típicos serían los browsers desde los que se dispara el comportamiento de una aplicación. Estos browsers muestran en general listas de datos sincronizadas, que a medida que se van seleccionando elementos en una de ellas provocan la actualización de las restantes, hasta llegar a un texto o información de la combinación de selecciones. Esto implica que podrá haber una primera lista fija en el browser, pero todas las demás y los textos irán cambiando su valor. En el caso del widget textual, sobre el que se podrán definir anclas de links clasificados o no, se tendrá un conjunto distinto de anclas según el valor textual que se esté mostrando. Supongamos por ejemplo un browser de vuelos entre ciudades para distintas aerolíneas y preferencias. El texto que muestra la información completa del vuelo según la combinación de selecciones y preferencias podrá tener para un vuelo particular una anotación sobre un cambio temporal de la ruta de vuelo con cierta escala agregada o quitada. Resulta obvio que el ancla hacia esa anotación no debería aparecer en la información de otro vuelo.

De lo anterior resulta que los navigators mantienen las anclas con una organización particular, en la que se asocia a cada widget de lista una "clase de ancla" (ver Sección 2.2.2 bajo el nombre de AnchorClass), y a cada widget textual un conjunto de anclas agrupadas por valor posible del widget.

Las responsabilidades asignadas a esta clase se pueden resumir entonces en:

- crear una representación para la ventana que agregue "hypermedia-aware wrappers" o decoradores que agregan navegación;

- mantener las anclas por widget y por valor posible de texto en el caso de widgets textuales.

♦ **Nodo anotación (Annotation)**

Un nodo anotación es un nodo con único dato de texto. Podemos decir que es un nodo dependiente, no en el sentido de Smalltalk sino porque no existe por sí sólo; su existencia depende de aquel nodo donde se creó la hotword anotada que lo activa. Podemos llamar a este último nodo su "dueño".

Un nodo anotación no aparece entonces en el conjunto de nodos de la hipermedia; el que aparece es su dueño, que mantiene una referencia hacia él por intermedio del ancla que lo activa. A pesar de esto se lo considera un nodo porque puede constituir el origen de un link. La diferencia con el resto de los tipos de nodos es que no puede constituir el destino de un link más que de aquel que nace en su nodo dueño.

La clase Annotation redefine el comportamiento para abrirse en forma modal (al estilo diálogo).

- **Link (HyperLink)**

Los links son los componentes esenciales de una hipermedia junto con los nodos, y al mismo nivel de importancia que ellos. Son los elementos que hacen factible la navegación, característica principal de una aplicación hipermedia.

Los links relacionan dos o más nodos, el primero llamado *origen del link* y el o los segundos llamados *destinos* o punta de link. Se considera que un link puede tener sólo un nodo origen, pero pueden existir varias anclas dentro del nodo para activar dicho link.

El link es el responsable de activar o abrir el nodo destino cuando él mismo es activado. En el momento de activarse, el link puede ejecutar ciertas acciones que tengan que ver con la navegación o con alguna poscondición al partir del nodo origen. Estas acciones pueden especificarse en un bloque o como un mensaje a ser enviado al nodo origen.

Para poder dar cierta semántica a los links, éstos tienen un atributo que define su significado. Este significado está redefinido en las subclases de HyperLink, como veremos posteriormente.

Volviendo al destino del link, existen distintas variantes para su activación. Además de ser simple o múltiple, el destino de un link puede ser fijo, cuando se especifica explícitamente el/los nodos a los que arriba, o computarse dinámicamente al momento de la activación, entre otros. Para evitar tener que testear por el tipo de destino, se separó la estrategia de obtenerlo en una jerarquía aparte de "resolvedores de destino" (EndpointSolver), con subclases específicas para cada forma distinta de activación. Por otro lado, la cardinalidad también requirió una jerarquía distinta de "destinos de link" (LinkEndpoint), subclasificada en "destino simple" (SingleEndpoint) y "destino múltiple" (MultipleEndpoint). En la Figura 8 vemos la jerarquía que comienza en HyperLink, y su relación con EndpointSolver y LinkEndpoint, las que serán explicadas más abajo.

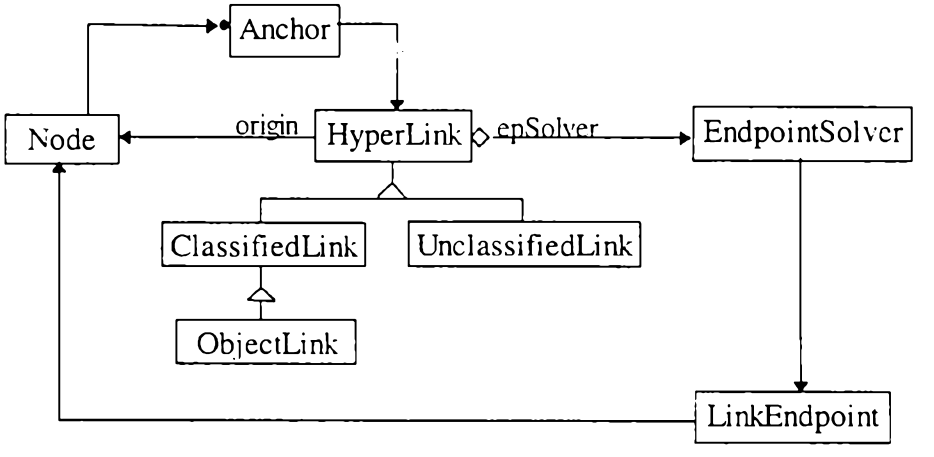


Figura 8. Jerarquía de HyperLink y su relación con los colaboradores EnpointSolver y LinkEndpoint.

La Figura 9 explica la forma de activación del destino de un link en un diagrama de interacción (para detalles sobre la notación utilizada referirse a [Buschmann+96]).

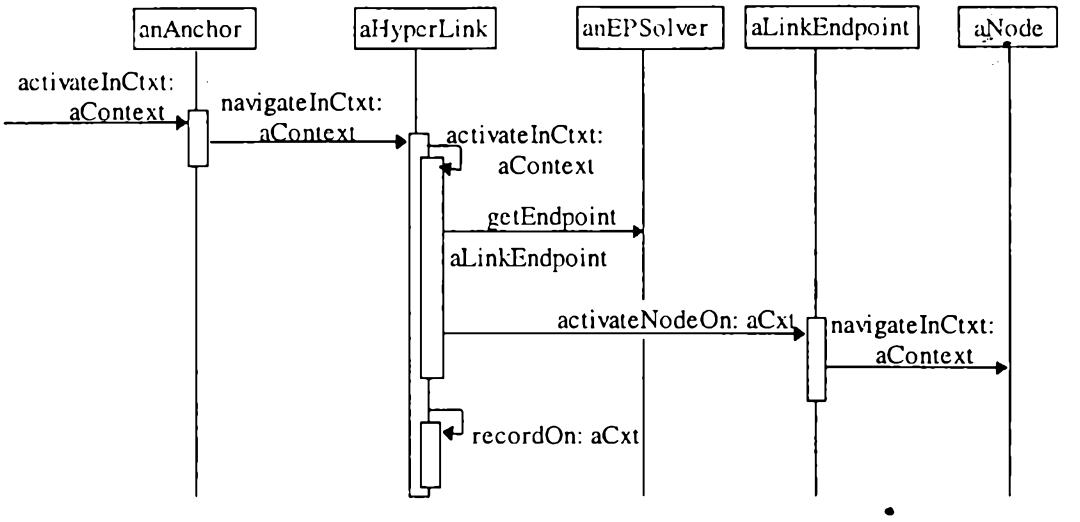


Figura 9. Diagrama de interacción con activación del destino.

La clase HyperLink es una clase abstracta que factoriza las características arriba mencionadas para sus subclases. Al igual que ciertos nodos "observan" objetos de la aplicación, ciertos links también observan y mapean las *relaciones* entre esos objetos. Esto provoca la subclasificación de esta clase en links observadores o "clasificados" y links "no clasificados". Los links clasificados obtienen tal denominación del hecho de poseer una "clase de link" que define sus características, de la misma forma que la clase de nodo lo hacia para los objectNodes.

En resumen, las responsabilidades de esta clase son:

- mantener su origen;
- mantener su destino y activarlo cuando el link es activado;
- mantener su significado;
- mantener las acciones involucradas en la navegación, o poscondiciones.

♦ **Link que responde a una clasificación (ClassifiedLink)**

En el caso de los nodos, estos pueden o no mapear directamente objetos que se encuentran en la aplicación subyacente, como comentamos anteriormente. En el caso de los links existe una diferencia en cuanto a qué pueden mapear u observar de la aplicación. Los classifiedLinks son los que observan *relaciones* entre objetos de la aplicación. Estas relaciones no son fáciles de hallar si no se tiene un modelo donde queden explícitas; en el Capítulo 3, Sección 3.3, se plantean pautas para la identificación de relaciones.

Por otro lado los links pueden observar directamente objetos, denominados *asociativos*, que representan una relación entre 2 o más objetos. Estos links están representados por la subclase ObjectLink (especialización de ClassifiedLink) y los detallaremos en el próximo inciso.

Un grupo de classifiedLinks que observan una relación entre las mismas clases de objetos (es decir por ejemplo, todos los links que observan una relación entre objetos de la clase A con objetos de la clase B) responderán a un mismo "patrón" determinado por esas clases. Este patrón o tipo determina además el significado, las poscondiciones, la forma de obtener el destino, etc. para todos sus classifiedLinks.

Los colaboradores de los classifiedLinks que hacen de "tipo" de los mismos son instancias de la clase LinkClass. Como se podrá observar esta relación entre ClassifiedLink y LinkClass es exactamente la misma que entre ObjectNode y NodeClass. Por esta razón todo classifiedLink debe conocer directamente a su linkClass. La clase de link determina por ejemplo el cómputo a realizar para obtener el destino del link en el momento de activación, preguntando al objeto del origen por el objeto destino de la relación en la aplicación, y obteniendo el objectNode que lo mapea.

El hecho de que un link sea clasificado queda totalmente determinado por el origen, que deberá ser un nodo-objeto. El destino de un classifiedLink podrá ser otro nodo-objeto (si la relación se mapea de la aplicación subyacente), o un nodo fijo, si se desea agregar una relación hacia un nodo de hipermedia o navegador, en el nivel hipermedial.

Las responsabilidades de los links clasificados son entonces:

- mantener la clase de link a la pertenece según su origen;

- responder a su significado y poscondiciones dependiendo de la clase de link a la que pertenece.

- **Link asociado a objetos de la aplicación (ObjectLink)**

Esta clase es la que representa los links que observan objetos asociativos de la aplicación. Este tipo de links pueden ser comparados directamente con los objectNodes en cuanto a que están relacionados con el componente de la aplicación que mapean y del cual obtienen su identidad, información y comportamiento. Esto se traduce para un objectLink en obtener del objeto asociado su significado y opcionalmente las acciones resultado de la navegación.

Las responsabilidades de un objectLink son:

- conocer el objeto asociado;

- interactuar con el objeto asociado para responder a su comportamiento.

- **Link que no responde a una clasificación (UnclassifiedLink)**

Una hipermedia flexible y más utilizable es aquella que permite al usuario definir links con significado especial para él mismo según su interés, sin necesidad de tener que restringirse a las clases de links previamente definidas por el diseñador. Los links no clasificados son aquellos que no responden al "patrón" determinado por una clase de link.

Cada vez que el usuario cree un ancla de link desde la interface sobre un texto o dibujo, estará provocando la instanciación de la clase UnclassifiedLink. El destino de ese link será fijo, es decir, cualquier nodo no clasificado (instancia de HyperNode, Navigator, CollectionNode o Annotation).

Al momento de construir un hyperNode o un navigator, se pueden crear links no clasificados, con la posibilidad de que el destino sea computado hacia uno o varios objectNodes. Por ejemplo, si se define un link sobre una lista que se muestra en un navegador, aquella lista pasará a actuar como un índice. Al tratar de navegar desde un ítem de la lista, se activará el objectNode correspondiente según la clase de nodos

que se haya definido como destino y según el bloque que especifique como seleccionar el nodo usando el ancla activada.

Por el contrario durante la navegación un usuario no podrá definir links hacia objectNodes, pues no corresponde que mientras navega defina el algoritmo de cómputo de un link.

Las responsabilidades de esta clase son entonces:

- mantener su significado y poscondiciones;

- crear su destino: fijo, hacia un nodo no-clasificado, ó computado, hacia un objectNode, simple o múltiple.

- **Estructura de Acceso (AccessStructure)**

Esta clase representa la forma de acceder a un conjunto de nodos. La razón de ser de esta abstracción se discutió previamente citando varias referencias.

Las estructuras de acceso se caracterizan por tres atributos, de acuerdo con el modelo presentado en OOHDM: el conjunto de nodos "target", el conjunto de selectores o atributos de los nodos que serán mostrados en la estructura, y un predicado lógico sobre los nodos "target" que permite la definición de estructuras de acceso condicionales, como se propone en la metodología RM.

- **Estructura de Acceso Indexada (Index)**

Esta estructura provee un acceso directo a sus componentes. Presenta un listado donde por cada entrada se muestra la concatenación de datos determinada por los selectores sobre un nodo target que cumplen el predicado de selección.

- **Estructura de Acceso Secuencial (Sequencial)**

Esta estructura provee un acceso secuencial a sus componentes, por medio de botones que permiten ir hacia adelante ('next') o hacia atrás ('previous') en la secuencia.

2.2.1.2. Jerarquía de Clase de Nodo y Clase de Link

- **Clase de nodo** (NodeClass)

Una instancia de NodeClass, como se dijo anteriormente, simula ser una clase que contiene la estructura y comportamiento de sus instancias, pero esta estructura y comportamiento serán las que se observan de la aplicación subyacente que se está extendiendo.

La incorporación de esta clase tuvo como propósito principal evitar la subclasificación de ObjectNode por cada clase de la aplicación que se quisiera observar. Esta subclasificación era necesaria principalmente por dos razones:

- especificar los aspectos a ser observados sobre un grupo de objetos de la misma clase (y por lo tanto que comparten esos aspectos), para un cierto grupo de nodos:

- permitir la automatización en la creación de nodos, y así evitar que por cada objeto de la aplicación se creara su nodo observador "a mano".

A pesar de tener estas necesidades, subclasificar ObjectNode era una muy mala solución, dada la terrible explosión de clases que esto provocaría por cada nueva aplicación (una sola aplicación hipermedia generalmente involucra muchas clases de nodo, y la cantidad se multiplica en cada aplicación diferente). Además, esta subclasificación estaría en manos del usuario del framework, que en cada distinta instanciación del mismo necesitaría crear nuevas clases. Esto implicaría que el usuario debe conocer detalladamente la implementación del framework para poder extenderlo, lo que lo convierte en un "framework caja blanca" [Johnson+88].

Otro problema es que la clase ObjectNode es subclasificada por otro criterio, como vimos anteriormente, lo que confundiría las distintas dimensiones de subclasificación.

Bajo estas circunstancias, la solución fue crear una clase distinta, que llamamos NodeClass, y hacer que sus instancias actuaran simulando clases para cierto grupo de nodos. Este diseño sigue el propuesto por el "Type-Object" pattern (ver Sección 2.2.3.3).

Las ventajas que se lograron por sobre la solución de subclasificar fueron:

- el usuario sólo necesita especificar las clases de la aplicación a ser observadas, y los aspectos a mapear de cada una; por cada clase a observar se crea una instancia de NodeClass, y automáticamente todas las instancias de ObjectNode (en realidad de sus subclases), necesarias para mapear cada uno de los objetos de aquella clase. Tanto la nodeClass como sus objectNodes mantendrán, a distinto nivel de especificación, los aspectos mapeados.

- por cada objeto de la aplicación que es creado dinámicamente durante la ejecución de la hipermedia, se crea automáticamente el nodo asociado si corresponde; esto es posible porque la clase de nodo observa la *creación* de cada nueva instancia de su clase asociada;

- la implementación del framework no necesita ser conocida por el usuario que sólo quiere usarlo, ya que no necesita crear nuevas clases; sólo instanciará las existentes. Esto se denomina "black-box framework" que obviamente es preferido sobre un "white-box framework" por su mayor facilidad de utilización (ver Sección 1.4.4).

Una clase de nodo puede mapear más de una clase de la aplicación. De esta manera se permite flexibilizar el diseño de la hipermedia con respecto al diseño de la aplicación, que no había sido pensado para ser "navegado". La navegación implica una serie de consideraciones en cuanto a eficiencia, presentación de la información en forma inteligible, cuidado de no provocar desorientación, etc., que podrán ser tenidas en cuenta al crear las distintas clases de nodo.

Otra flexibilización tenida en cuenta es la posibilidad de especificar un predicado de selección de los objetos a ser mapeados. Así, si ciertos objetos de la clase observada no necesitan ser mapeados a nodos, podrán filtrarse mediante dicho predicado que se especificará al crear la clase de nodo.

De la misma manera que se subclasifica ObjectNode en AtomicNode y CompositeNode, se debe subclasificar NodeClass en AtomicNodeClass y CompositeNodeClass. Esta subclasificación es necesaria ya que los nodos se crean de distinta forma para una subclase y la otra. Entonces NodeClass es en realidad una clase abstracta.

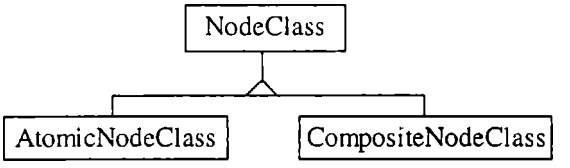


Figura 10. Jerarquía de NodeClass

El hecho de que los nodos que mapean objetos de la aplicación sean compuestos de distintas vistas requiere que una clase de nodo también se componga de la misma forma, dado que los nodos son "clones" o copias de ella. De esta forma aparecen las clases de vista de nodo (NodeViewClass) que actúan como clases de nodo pero para las vistas de los mismos. Una clase de nodo es entonces una agregación de clases de vistas de nodo.

Resumiendo lo dicho, las responsabilidades de esta clase son:

- conocer la/s clase/s de la aplicación que observa;

- conocer toda la información necesaria para crear los nodos que observen objetos de la clase de la aplicación relacionada (hipermedia a la que pertenece, aspectos a ser observados, clases de vistas de nodo, anclas de links);

- crear los nodos necesarios por cada objeto de la aplicación perteneciente a la clase relacionada;

- mantener los nodos que siguen su patrón, para permitir la modificación de ellos ante cambios o agregación de datos;

- mantener las clases de vistas de nodo, y delegarles a ellas la creación de las vistas para cada nuevo nodo que debe ser creado.

- **Clase de nodo atómico (AtomicNodeClass)**

Las instancias de esta clase cumplirán el rol de "patrones" para grupos de instancias de AtomicNode. De esta manera, una atomicNodeClass tiene la responsabilidad de crear, mantener y modificar sus atomicNodes ante cualquier cambio en su definición.

Así como dijimos que un nodo atómico podía observar más de un objeto de la aplicación, un atomicNodeClass podrá observar más de una clase de objetos. Deberá existir una clase principal asociada, de donde obtiene su identidad (se puede chequear que esto es totalmente compatible con AtomicNode). Las demás clases a observar se podrán ir agregando, indicando la forma en que cada una se relaciona con la clase principal. Esto lo hace mediante un predicado que determine cómo las instancias de la clase asociada se unirán con instancias de la clase principal para formar cada nodo. Este predicado será chequeado cada vez que se cree un objeto en la aplicación de alguna de estas clases, con lo que se creará un nuevo nodo (si el objeto pertenece a la clase relacionada principal) o el objeto pasará a estar observado por un nodo existente, si cumple el predicado con alguno de ellos.

Las responsabilidades de esta clase serán entonces:

- agregar clases asociadas, retornarlas y borrarlas;

- crear un nodo nuevo o agregar un objeto a observar en un nodo existente cuando se ha creado un nuevo objeto en la aplicación subyacente.

- **Clase de nodo compuesto (CompositeNodeClass)**

Como ya se dijo, un nodo compuesto representa una entidad compleja, formada por partes, capturando una organización jerárquica de la información, no basada en links sino en composición. El patrón que siguen estos nodos vendrá dado por la instancia correspondiente de la clase CompositeNodeClass.

Como DependentCompositeNode sólo difiere de su superclase en el borrado de una de sus partes, proceso en el cual la clase del nodo no interviene, se decidió que CompositeNodeClass fuera también el patrón para las instancias de aquella clase.

Cada compositeNodeClass tiene una clase principal asociada, que en este caso será aquella cuyos objetos son agregaciones. Las clases que constituyen las partes serán mapeadas previamente a otras nodeClasses para luego agregarlas como sub-nodeClasses. Esto lo hace mediante un predicado que determine cómo las instancias de la clase "parte" se unirán con instancias de la clase principal para formar cada nodo. Este predicado será chequeado cada vez que se cree un objeto en la aplicación de alguna de estas clases.

Las responsabilidades de esta clase serán entonces:

- agregar clases de nodos como partes, retornarlas y borrarlas;

- crear un nodo nuevo o agregar subnodos a uno de sus nodos, cuando el subnodo se ha creado.

- **Clase de Link (LinkClass)**

Idénticas razones por las que fue necesario introducir la clase NodeClass motivaron la definición de la clase LinkClass. Es decir, la misma relación existente entre NodeClass y ObjectNode se presenta entre LinkClass y ClassifiedLink. La diferencia existente es que LinkClass no observa una clase del nivel de objetos sino una relación entre clases, representada mediante el origen, destino, significado y poscondiciones, que respetarán todos los links pertenecientes a esta clase. El origen de una linkClass es siempre una nodeClass y de esta manera un link será creado a partir de cada nodo de la nodeClass origen.

Una clase de link que mapea una relación del nivel de objetos tendrá necesariamente como destino otra clase de nodos, y la especificación de cuáles nodos resultan relacionados estará dada mediante un predicado que involucra a los objetos representados en los nodos origen y destino. De esta manera, una linkClass creará un classifiedLink por cada nodo origen automáticamente al ser especificada, pero recién en el momento en que el link es activado se computa el destino según aquel predicado. Así como la especificación del destino de un link se mantiene en una instancia de LinkEndpoint en colaboración con un endpointSolver, una clase de link colabora con una 'clase de destino de link' (LinkEndpointClass) para mantener la especificación del destino para todos los links clasificados que le pertenezcan. En el

caso en que el destino sea fijo, este se mantiene directamente en un LinkEndpoint. Por último, en el caso en que el destino deba crearse en el momento de la activación, lo que se mantiene es una instancia de EndpointFactory con la especificación de lo que se debe crear.

En el nivel de hipermedia puede surgir la necesidad de definir una relación entre una clase de nodos y un nodo no clasificado en particular. Por ejemplo de la clase de nodo 'Salas' de un museo, al hyperNode que muestra el mapa de ubicación de las salas. Este tipo de relación, a pesar de que no surge del nivel de objetos, también es representada por una linkClass que, como siempre, crea un classifiedLink desde cada nodo origen. El destino ya no será computado sino fijo en el nodo pertinente.

Resumiendo lo anterior, las responsabilidades resultantes de esta clase son:

- conocer la clase de nodos origen;
- crear y mantener el destino de los links;
- mantener el significado de la relación;
- mantener las poscondiciones de navegar los links que le pertenecen, y activarlas;
- crear y mantener los links que pertenecen a sí misma, cloneando sus datos.

♦ **Clase de links que mapea una clase de la aplicación (ObjectLinkClass)**

Esta clase puede ser directamente comparada con la clase NodeClass, ya que sus instancias serán observadores de clases de la aplicación. En este caso serán clases de objetos asociativos que representan una relación entre dos o más objetos de la aplicación subyacente.

Las instancias de ObjectLinkClass simularán ser clases de las instancias de ObjectLink. De esta forma serán las responsables de crear un objectLink por cada objeto de la clase asociativa que mapean.

Las responsabilidades que se adicionan a esta clase son:

- conocer la clase asociativa de la aplicación que representa;
- observar la creación de instancias de la clase que representa;
- mantener el bloque que selecciona las instancias de la clase asociativa que se desea sean mapeadas a links;
- mantener los mensajes que los objectLinks deberán enviar a las instancias de la clase asociativa para obtener su significado y poscondiciones.

2.2.1.3. Jerarquía de Estrategia de definición de Nodos-Colección

♦ **Estrategia de definición de colecciones (CollectionStrategy)**

Cuando hablamos de la clase CollectionNode dijimos que el framework proveía distintas estrategias para la definición de sus instancias (basándonos en el modelo planteado en [Garzotto+94]). Esto se hace posible en forma elegante y exenta de estructuras de decisión mediante la definición de una jerarquía de estrategias que encapsulan el algoritmo de creación. y lo desacoplan de la clase CollectionNode. La raíz de esta jerarquía de clases es CollectionStrategy (clase abstracta).

El diseño de esta la relación entre CollectionNode y CollectionStrategy sigue la estructura planteada por el Pattern Strategy (ver Sección 2.2.3.5).

En la siguiente figura podemos observar toda la jerarquía definida a partir de CollectionStrategy.

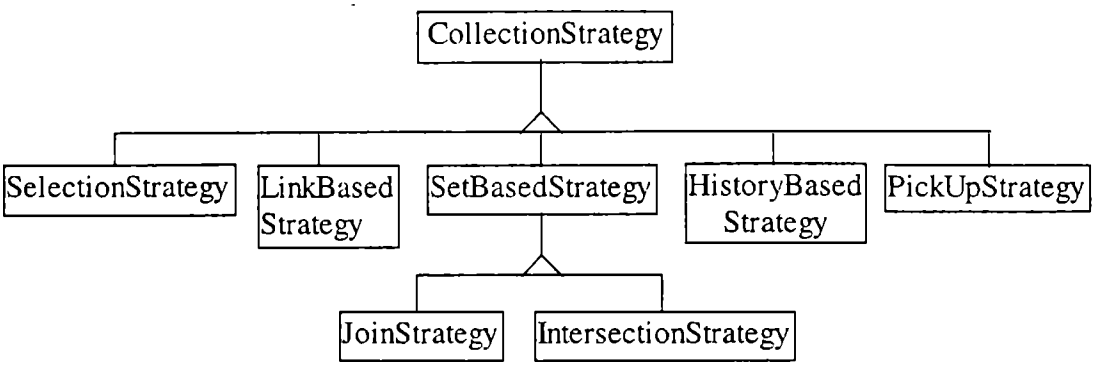


Figura 11. Jerarquía de CollectionStrategy

Las principales responsabilidades de las subclases de CollectionStrategy son:

- mantener la configuración de la colección (cada subclase define su propio mensaje de creación);

- evaluar la especificación del nodo colección retornando cuando sea requerido, los nodos pertenecientes al collectionNode.

- **Estrategia de selección para la definición de colecciones (SelectionStrategy)**

  Esta estrategia se utiliza cuando se desea crear un nodo-colección a partir de una clase de nodos, seleccionando aquellos nodos que cumplen cierto predicado. Esta estrategia conjuga las que Garzotto denomina "built-in" + "intensional".

- **Estrategia por aplicación de links para la definición de una colección (LinkBasedStrategy)**

  Dado un nodo-colección inicial y una clase de links, esta estrategia consiste en obtener el nodo-colección resultante de aplicar los links tipados por aquella clase de links, a cada elemento de la colección original. De más está decir que la clase de links deberá tener como origen la clase de nodos a la que pertenecen los elementos del nodo-colección inicial. En el modelo de Gazotto esta estrategia se denomina "link-oriented".

- **Estrategia basada en operaciones de conjuntos para la definición de una colección (SetBasedStrategy)**

  Esta clase es en realidad abstracta y representa las operaciones de conjuntos que se pueden utilizar para la definición de un CollectionNode en base a otros ("set-oriented strategy" en el modelo de Garzotto).

- **Estrategia de unión de colecciones para la definición de una nueva colección (JoinStrategy)**

  Esta especialización de estrategia basada en operaciones de conjuntos permite definir un nuevo nodo-colección en base a la unión de otros dos nodos-colección anteriores.

- **Estrategia de intersección de colecciones para la definición de una nueva colección (IntersectionStrategy)**

  Esta especialización de estrategia basada en operaciones de conjuntos permite definir un nuevo nodo-colección como la intersección de otros dos nodos-colección anteriores.

- **Estrategia basada en la historia de nodos visitados para la definición de una colección (HistoryBasedStrategy)**

  Como su nombre lo dice, esta clase encapsula el algoritmo que permite configurar los elementos de un nodo-colección a partir de la historia de nodos visitados ("session-based" en el modelo de Gazotto). El hecho de observar la historia de navegación a través de un nodo-colección podría ser utilizada para hacer explícito al lector el estado de la navegación en que se encuentra (por ejemplo para revisar los conceptos aprendidos al finalizar un guided tour en una hipermedia de aprendizaje). Otra posible utilización de esta estrategia sería como base para aplicar otras estrategias sobre ella (por ejemplo haciendo la intersección con los nodos pertenecientes a la clase de nodo "Pintura" que ya han sido visitados: IntersectionStrategy sobre SelectionStrategy y HistoryBasedStrategy).

- **Estrategia de elección manual de los componentes de una colección (PickUpStrategy)**

  Esta estrategia se utiliza cuando los componentes de un nodo-colección se quieren elegir manualmente, cuando ninguna otra estrategia automática puede ser aplicada; Garzotto la denomina "pick-up".

2.2.1.4. Jerarquía de Destino de Link

- **Destino de link (LinkEndpoint)**

  El destino de un link no está formado sólo por el o los nodos correspondientes, sino también por la representación del nodo a la que en particular se arribará tras navegar el link. Este conocimiento está encapsulado en instancias de la clase LinkEndpoint. Cabe destacar que el contexto en el que se produce la navegación queda invariante después de ella, a menos que las poscondiciones del link provoquen explícitamente un cambio de contexto. Es por esto que en el destino del link no hace falta especificar el contexto al que se arriba. Si la representación, por otro lado, no fuera determinada, se utiliza la representación por defecto que el nodo tenga definida para el contexto actual.

  Por otro lado, la cardinalidad de un link puede ser simple o múltiple. Para que el link pueda comportarse interactuando con su destino independientemente de su cardinalidad, se ha definido una jerarquía separada, con un protocolo uniforme completamente especificado en la clase abstracta LinkEndpoint de manera que cada subclase redefine el comportamiento de activarse acorde a si es simple o múltiple.

  La siguiente figura muestra la jerarquía definida a partir de LinkEndpoint.

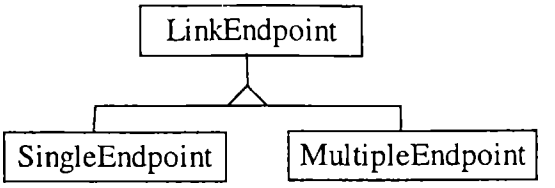


Figura 12. Jerarquía definida a partir de LinkEndpoint

Las responsabilidades abstraídas en esta clase abstracta son:

- mantener el/los nodo/s que forma/n el destino del link, así como el nombre de la representación a la que se arriba tras activar los nodos;

- activar el nodo destino en el contexto actual y la representación especificada.

- **Destino de link singular (SingleEndpoint)**

Representa un destino simple. En el caso en que este destino pertenece a una clase de nodos, y es computado recién al ser activado el link, en el endpoint se guarda sólo la especificación para obtener el nodo.

Las responsabilidades de esta clase son, en síntesis:

- activar el nodo en la representación especificada (redefinición del método abstracto);

- compararse con otras especificaciones de endpoints.

- **Destino de link múltiple (MultipleEndpoint)**

Representa un destino múltiple. Cuando es activado presenta un índice con la colección de destinos para que el usuario seleccione el que le interesa, y a ese se navega. También se guarda la forma de acceder a la colección cuando pertenece a una clase de nodos.

Responsabilidades de esta clase son:

- mantener el selector que denota el aspecto del nodo que se mostrará en el índice para seleccionar un destino de entre la colección;

- activar el destino, presentando el índice con la colección de nodos y navegando hacia el que se seleccione (redefinición del método abstracto);

- compararse con otras especificaciones de endpoints.

2.2.1.5. Jerarquía de Resolvedor de Destino de Link

- **Resolvedor de destino de link (EndpointSolver)**

Ya dijimos anteriormente que el link colaboraba con su "resolvedor de destino" para obtener el o los nodos hacia los que se debe navegar. La jerarquía definida a partir de la clase abstracta EndpointSolver es la que encapsula los distintos algoritmos de obtención del destino de un link. Veremos en la Sección 2.2.4.1 que este diseño se basa en un pattern descubierto a partir de este trabajo, llamado NavigationStrategy.

La Figura 13 muestra la jerarquía completa a partir de EndpointSolver.

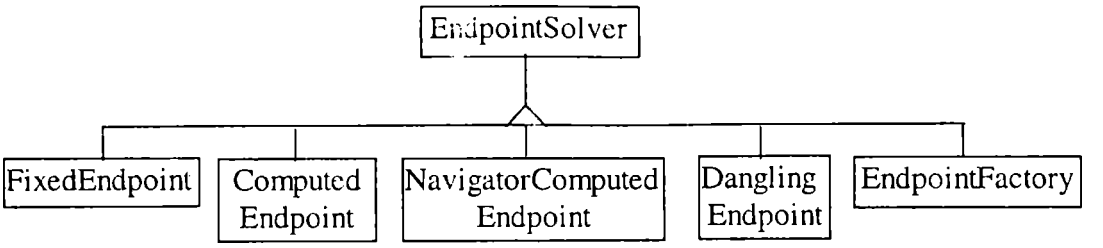


Figura 13. Jerarquía de EndpointSolver

La clase EndpointSolver define el protocolo uniforme que cada subclase redefine para obtener el endpoint del link, permitiendo que este último se desentienda de proveer múltiples sentencias condicionales sobre la forma de obtención. Tanto la jerarquía de HyperLink como la de EndpointSolver podrán entonces evolucionar independientemente, desacoplando las responsabilidades de cada una completamente. Además esta solución permite que un link cambie la forma de obtener su endpoint dinámicamente.

Las responsabilidades abstraídas en esta clase incluyen:

- crear y mantener la especificación del destino de link;
- calcular el destino de link en el momento en que es requerido;
- retornar la cardinalidad del destino.

- **Resolvedor de destino fijo (FixedEndpoint)**

Esta estrategia de obtención del destino del link es la más simple ya que mantiene directamente el o los nodos en cuestión. No hace falta realizar ningún cálculo ya que simplemente retorna el nodo que mantiene.

Esta clase será instanciada por cada link creado "a mano" o en forma individual por el usuario. La mayor ventaja de los links creados de esta forma es que reflejan más fielmente los asociaciones libres del usuario, en aquellas situaciones donde no se quiere o resulta poco razonable o incluso imposible implementar un bloque de código que obtenga el endpoint. Por ejemplo para definir links que mantienen el "design rationale", anotaciones, links de corto plazo o que no tienen un razonamiento explícito, resulta indispensable la posibilidad de creación "a mano" hacia un destino fijo.

A pesar de ser la estrategia más simple y que puede considerarse más eficiente en cuestión de tiempo de resolución, es la más restrictiva en cuanto a la definición de links, la más ineficiente en cuanto a espacio necesario, y la que puede provocar mayor instanciación de relaciones impropias, innecesarias e incluso inconsistentes. Una mayor discusión al respecto puede encontrarse en [Ashman+97], [Garrido+96b].

- **Resolvedor de destino computado para links clasificados (ComputedEndpoint)**

Esta clase resuelve directamente los problemas surgidos con la anterior, FixedEndpoint, ya que permite el dinamismo y la automatización de calcular el destino de un link recién en el momento en que resulta necesario.

La utilización de los links computados tiene mayor sentido, o resulta más obvia, en situaciones cuando se instancia, mediante un conjunto de links similares, una

misma relación entre clases de nodos, es decir una relación existente en el modelo de objetos. Un ejemplo sería la relación entre cada pintura con su pintor; no tendría sentido crear manualmente los links entre los pares de nodos pertinentes. En el framework los destinos computados se utilizan para definir el destino de una clase de links entre dos clases de nodos. Esto significa que el bloque de cómputo se basa en una relación a calcular sobre el modelo de objetos, que se evaluará de la misma forma con el origen (y posiblemente el destino) de cada link clasificado por la aquella clase.

El hecho de computar dinámicamente el destino de un link implica que la relación que se está instanciando tiene cierta semántica que permite definirla como un predicado, cuya evaluación proveerá su intanciación automática; otorga la posibilidad de adaptación instantánea ante un cambio en la relación entre los objetos subyacentes, obteniendo siempre la versión actualizada y por lo tanto consistente; permite además que si el nodo destino reside en una base de datos, recién se traiga a memoria cuando se desee navegar hacia él.

Los links computados se ocupan de reducir en cierta forma el esfuerzo requerido para crear y mantener links [Ashman+97]. Primeramente, remueven la necesidad de la instanciación manual. Segundo, la completitud de los links computados es alta, pues todos los links que mapean una relación (es decir los que pertenecen a la misma clase de link) son creados mediante un proceso idéntico. Otra ventaja muy importante consiste en permitir la utilización de links en aplicaciones más del tipo computacionales, donde las relaciones se crean dinámicamente como resultado de las operaciones que se van realizando, o a partir de queries.

- **Resolvedor de destino computado para links no-clasificados (NavigatorComputedEndpoint)**

  Para motivar esta clase con un ejemplo, supongamos un nodo Navigator definido sobre un browser que permite calcular itinerarios a partir de una lista de ciudades. Pensemos además que existe una clase de nodos para las ciudades. Sería muy útil tratar a cada elemento de la lista que se encuentra en el navigator como un ancla de link hacia la ciudad representada por ese elemento. En otras palabras, quisiéramos que al seleccionar un elemento de la lista y provocar la navegación se calcule, a partir de la clase de nodos representando ciudades, aquel nodo que represente el ítem seleccionado. Este cálculo requiere como parámetros la clase de nodo destino y el ancla seleccionada.

  La clase NavigatorComputedEndpoint aparece para permitir que el nodo destino de un link se calcule entre aquellos de una clase de nodos, y dependiendo de la selección actual en una lista mostrada en un Navigator. Este es el caso en que nada puede obtenerse del nodo origen del link para el cómputo del destino, porque aquel nodo no mapea un objeto del modelo sino simplemente una interface.

- **Resolvedor de destino no especificado (DanglingEndpoint)**

  Esta clase provee la flexibilidad de poder crear un link especificando sólo su origen, pero no su destino, cuando este último aún no ha sido creado o determinado. Un ejemplo de utilización de "dangling links" lo podemos encontrar en las páginas del web con una interface al estilo "wiki wiki" (http://c2.com/cgi/wiki?WikiWikiWeb).

Como expusimos previamente al presentar el propósito y la descripción del concepto de Link implementado por el framework (Sección 2.1.1.2.), la presencia de dangling-links es soportada por varios modelos de hipermedia (como DHM). Entre las situaciones de uso que se proponen encontramos: el destino del link será creado en un momento posterior o por otro usuario en un trabajo colaborativo; el destino del link ha sido borrado, ha dejado de ser válido, o no se encuentra disponible momentáneamente.

- **Resolvedor que fabrica el destino (EndpointFactory)**

La existencia de esta clase fue fundamentada en la Sección 2.1.2.- Extensión del modelo base, bajo el título "Creación dinámica de nodos y tardía de relaciones". Primeramente cabe destacar que esta forma de computar el destino de un link sólo se utiliza para relaciones *entre clases de nodos*. Para revisar el concepto, supongamos el caso en el que estamos creando un ancla de link desde un ObjectNode. Al crear un ancla durante el proceso de navegación, se le ofrece al usuario la posibilidad de elegir la clase de link a instanciar, en el caso de que hubiera varias. Al elegir la clase de link, se le delega a ella la responsabilidad de crear en el momento una instancia de ClassifiedLink hacia el destino que corresponda. La clase de link a su vez le delega a su especificación de destino esta responsabilidad. Las posibilidades que teníamos hasta este momento eran, o bien hacer que el destino fuera fijo a un nodo en particular, o calcularlo de acuerdo a los nodos existentes pertenecientes a la clase de nodo especificada en el destino. ¿Pero cómo hacer si el nodo no estaba aún creado y queremos que la navegación produzca la creación del destino? Estamos pretendiendo entonces crear el nodo destino y a su vez el objeto asociado, en el momento de activación del link.

Este servicio se provee de la siguiente manera: la clase de link guarda como destino una instancia de EndpointFactory. Cuando esta instancia recibe el pedido de recuperación del destino del link, provoca que la clase del modelo asociada con la clase de nodo del destino cree una nueva instancia (pasando el ancla como parámetro en el caso que fuera necesario). A su vez, gracias a que la clase de nodo observa el proceso de creación de instancias de la clase de objetos asociada, un nuevo nodo será creado automáticamente para mapear el nuevo objeto. El destino del nuevo link creado se fija así hacia este nuevo nodo.

Resumiendo entonces, esta clase permite que la navegación misma produzca la creación de nuevos objetos en el modelo. Lo mismo puede ser utilizado cuando el ancla ya existe sobre un botón y este es activado. Supongamos por ejemplo una aplicación financiera que permite proyectar las ganancias de una empresa en base a posibles inversiones. Existe un nodo por cada tipo de capital a invertir, que además permite setear nuevos parámetros de inversión. Existirá un botón 'Proyectar ganancia' que provocará la creación de un objeto Inversión en base a los parámetros, esto provocará a su vez la creación de un nodo asociado que será inmediatamente activado para mostrar al usuario la proyección resultado.

2.2.1.6. Jerarquía de Clase de Destino de Link

- **Especificación del destino de una clase de links (LinkEndpointClass)**

  Para guardar la especificación del destino a nivel de clase de links se utiliza alguna de las subclases de la clase abstracta LinkEnpointClass. La principal responsabilidad de esta abstracción es la de crear instancias de EndpointSolver y LinkEndpoint al momento de instanciar un nuevo ClassifiedLink. El conocimiento definido a este fin es la clase de nodo destino y el nombre de la representación a utilizar. Responsabilidades adicionales son las de retornar su conocimiento, compararse con otra linkEndpointClass, compararse con la especificación de una linkEndpointClass, y saber su cardinalidad (simple o múltiple).

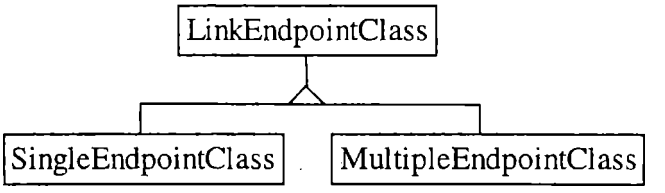


Figura 14. Jerarquía de LinkEndointClass

- **Especificación de destino singular de una clase de links (SingleEndpointClass)**

  Esta clase, como su nombre lo dice, especializa la representación del destino para una clase de links 1:1, es decir, singular. La principal responsabilidad redefinida es la de clonearse creando una instancia de SingleEndpoint, que es asociada al nuevo link a través de un ComputedEndpoint como resolvedor de destino.

- **Especificación de destino múltiple de una clase de links (MultipleEndpointClass)**

  Esta clase especializa la representación del destino para una clase de links 1:n, o sea múltiple. Así como su clase hermana, la principal responsabilidad redefinida es la de clonearse creando instancias de MultipleEndpoint que son asociadas a los nuevos links a través de ComputedEndpoints como resolvedores de destino.

2.2.2. *Biblioteca de componentes*

Esta sección incorpora las componentes definidas en el framework que no pertenecen a una jerarquía, es decir otras clases concretas que se instanciarán al utilizar un framework.

- **Hipermedia (Hypermedia)**

  Cada instancia de esta clase conformará una aplicación hipermedia distinta. Esta clase agrega todos los nodos, links, clases de nodos, clases de links, historia, y además conoce el primer nodo que se abrirá al abrir la aplicación hipermedia.

  Las responsabilidades de esta clase son:

  - mantener su identificación, que por lo general será su nombre;

  - proveer el protocolo necesario para crear clases de nodos, clases de links, nodos y links no clasificados;

  - proveer el protocolo para crear contextos y transiciones entre ellos;

  - abrirse iniciando una sesión activando el primer nodo;

  - mantener la historia de nodos visitados.

- **Manejador de la historia de nodos visitados (HistoryManager)**

  La historia de nodos visitados provee, como lo implementado en distintos ambientes, una lista ordenada con los nodos por los que el lector ha ido navegando y un acceso directo a ellos. La definición de HistoryManager en el framework permite además configurar la historia para que registre sólo los nodos visitados, o sólo los links activados, o ambas cosas. Además es posible definir un predicado que luego es evaluado con cada componente de la hipermedia que se activa, para determinar si lo registra o no. Esta funcionalidad se ha diseñado de manera transparente para los nodos, links y estructuras de acceso, a través del método abstracto `navigate` que provoca la registración. como se explicó bajo el título HypermediaComponent. Este diseño sigue el propuesto por el pattern Navigation Observer que será descripto en la Sección 2.2.4.2.

  Si el elemento a registrar es un nodo se guarda con él el contexto y la representación en la que fue accedido.

  Las responsabilidades de esta clase son:

  - registrar los componentes de hipermedia visitados en una sesión;

  - mantener la configuración de los objetos navegacionales a registrar;

  - reinicializarse;

  - filtrar la historia con cierto predicado de selección.

- **Ancla de link (Anchor)**

  Un ancla representa la forma de acceder a un link. Un ancla conoce tanto la forma con la que se muestra en la interface, como el link a activar en el momento que ella misma sea activada. Estas constituyen sus dos grandes responsabilidades.

- **Grafo de contextos (ContextsGraph)**

  Un grafo de contextos mantiene, para una hipermedia, los distintos puntos de vista o perfiles de usuario que accederán a la misma y que requieren una visión especial de ella. Por cada contexto o perfil se conocen además los restantes contextos a lo que un usuario con ese perfil podría cambiarse además del que ha sido definido expresamente para él. Los posibles cambios de contexto se representan mediante instancias de ContextTransition (ver Figura 15).

- **Transición de contexto (ContextTransition)**

  Cada instancia de esta clase representará un arco en el grafo de contextos. Esta clase provee además la posibilidad de definir acciones asociadas con el cambio de contexto, como podría ser resetear la historia de nodos visitados. Cuando un usuario decide cambiar de contexto la transición correspondiente es habilitada y esto produce la ejecución de las acciones asociadas.

- **Vista de Nodo (NodeView)**

  Una vista de nodo representa la forma en que un nodo de múltiples vistas se mostrará bajo cierto contexto o contextos. Una vista para un nodo particular se puede asociar a más de un contexto cuando la misma información será presentada a distintos perfiles de usuario. La siguiente figura muestra la relación existente entre el grafo de contextos y las vistas de nodos.

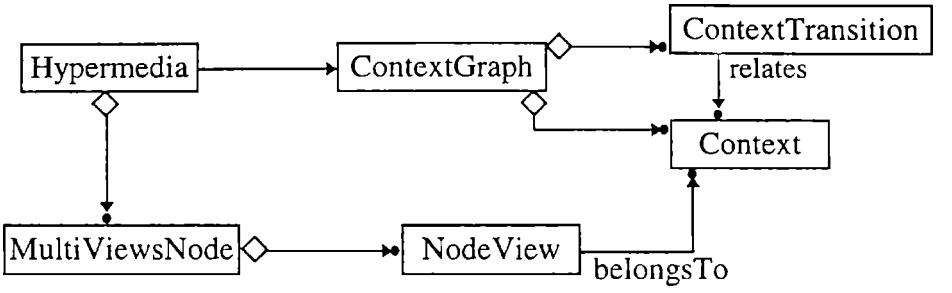


Figura 15. Contextos y Vistas de Nodo

Una instancia de NodeView mantiene los datos a mostrar, acciones que se podrán activar y anclas de links que dependen de uno o varios contextos en particular, con los que está asociada. Esto permite que la información de la entidad mapeada en el nodo se filtre dependiendo del perfil del usuario.

En el proceso de navegación, el contexto actual se va pasando como parámetro entre los nodos y links que se activan. Cuando un nodo recibe el mensaje de activarse, busca entre sus vistas la que corresponda al contexto actual y la activa.

Una vista es además una agregación de 'representaciones'. Una representación describe cómo se mostrarán los datos que la vista estipula. Se podría decir entonces que la vista denota el qué se va a mostrar y la representación el cómo se va a mostrar. Describiremos el concepto de Representación en el siguiente ítem.

Las responsabilidades de esta clase son entonces :

- Conocer los contextos en los que se utilizará.

- Conocer los datos del nodo a mostrar y las acciones que podrán activarse bajo los contextos asociados a la vista.

- Mantener las anclas que no están inmersas en un contenido del nodo si no que se representan mediante un widget por sí mismas, como sería en un botón.

- Crear y borrar anclas a ver en la vista, tanto inmersas o no en un contenido.

- Mantener el conjunto de representaciones con las que el nodo puede mostrarse en el contexto de la vista, y activar la representación que corresponda cuando la vista sea activada.

- Crear una nueva representación.

- Mantener la representación por defecto, que será la activada cuando no se especifica la representación en la que el nodo debe abrirse.

- Hacer de "mediador" (con el significado del pattern Mediator en [Gamma+95]) entre la representación activa y el nodo al que pertenece, ante el pedido de datos o comportamiento por parte de la representación y que no se encuentran en la vista particular sino en el nodo, por ser información de todas las vistas.

- **Representación de Nodo (Representation)**

Una representación para un nodo dentro de una vista describe la forma en que se verán los datos especificados en el nodo más los especificados en la vista.

Las instancias de esta clase actúan de modelo para la interface o ventana con la que se muestra. Una representación se crea con la especificación literal de la ventana que abrirá cuando ella misma sea activada (esto se explica en detalle en 2.3.1). Una vez abierta la ventana, la instancia de Representation que actúe como su modelo funcionará como "adaptador" entre el nivel de interface y el de hipermedia (ver Sección 2.2.3.6). Para esto redirige los pedidos de datos y acciones desde la interface a la vista de nodo a la que pertenece, y activa las anclas de links que han sido seleccionadas.

La representación automáticamente agrega a la ventana una barra de menúes que permite acceder a la historia de nodos navegados, seguir un link, hacer backtrack, cambiar de contexto y cambiar de representación.

Esta clase está muy relacionada con la implementación de interfaces gráficas de VisualWorks, ya que depende de poder contar con una especificación literal de la ventana, a la que mantiene, decodifica y abre en su momento. Aunque nada se haya implementado al respecto, creemos que una subclasificación de esta clase permitiría generar interfaces para otros ambientes, o código HTML.

- **Especificación de un dato más sus anclas (DataSpec)**

Esta clase representa la especificación de la forma de obtener un dato más las anclas inmersas en ese dato que hayan sido creadas.

La misma clase DataSpec se usa a distintos niveles y ella sabe copiarse o clonearse de un nivel a otro, siguiendo el pattern Prototype [Gamma+95]. Cuando esta clase es usada por instancias de NodeClass (lo mismo que NodeViewClass), la especificación de la forma de acceder al dato está contenida en una instancia de AccessorSpecification. Cuando esta clase es usada por instancias de ObjectNode

(o NodeView), contiene un AspectAdaptor cuyo modelo es el objeto asociado al nodo y cuyo aspecto es el selector que retorna el dato. Cuando en cambio es usada por otro tipo de nodo, la especificación del dato se mantiene en un ValueHolder sobre el dato mismo. Tanto AspectAdaptor como ValueHolder son clases provistas por VisualWorks, como subclases de ValueModel. Una descripción completa de las mismas puede encontrarse en [PP94, Woolf95].

Los datos que sean comunes a todas las vistas de un nodo se accederán mediante DataSpecs contenidos en el nodo mismo, mientras que los datos de una vista particular serán accedidos con DataSpecs de la vista.

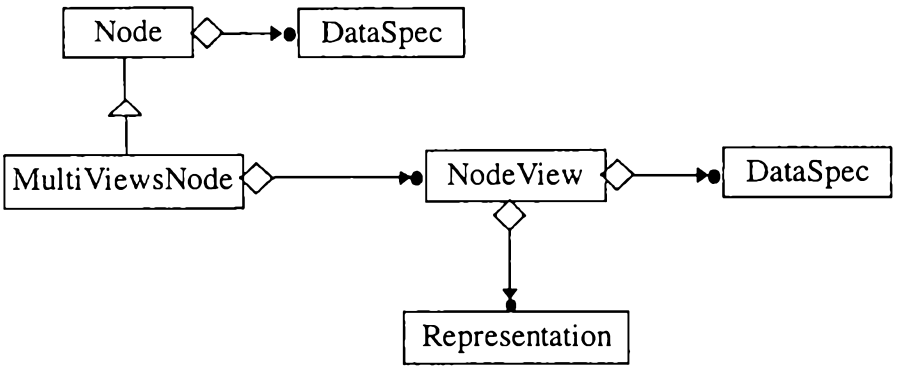


Figura 16. Agregación de DataSpec a dos niveles

- **Clase de Vista de Nodo (NodeViewClass)**

Sabemos a este punto que las instancias de NodeClass representan clases para los ObjectNodes y son las encargadas de crear instancias de estos últimos. Para esto una NodeClass guarda la especificación necesaria de las características que formarán a un nodo clasificado por ella. Entre las características que conforman un nodo están sus vistas, como vimos en el ítem anterior. La especificación de una vista de nodo a nivel de NodeClass se guarda en una instancia de NodeViewClass. A esta instancia se le delega además la responsabilidad de crear una instancia de NodeView que responda a su especificación. Resulta entonces que instancias de NodeViewClass pueden considerarse como tipos para las instancias de NodeView, aplicando nuevamente el pattern Type- Object (ver Figura 17).

Las responsabilidades de esta clase son entonces:

- Conocer los contextos con los que se asocia, para copiarlos en las instancias de NodeView que cree.

- Mantener la especificación de los datos, acciones y anclas para crear NodeViews con esta especificación.

- Mantener las representaciones que copiará en las vistas.

- Crear vistas de nodo que respondan a su especificación.

- **Clase de Ancla (AnchorClass)**

Esta clase viene a completar el nivel de clases que tipean la relación entre ObjectNode y HyperLink, arribando a una arquitectura como la que vemos en la

siguiente figura (las líneas de puntos indican que las instancias del origen de la flecha crean instancias de la clase del destino de la flecha).

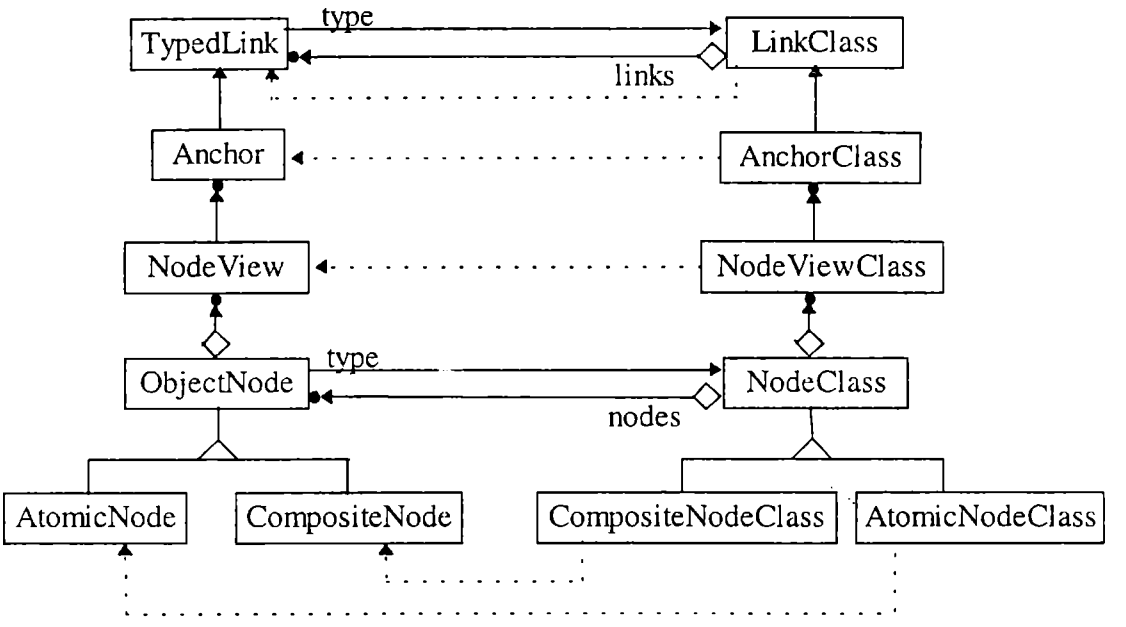


Figura 17. Niveles de clase e instancia

Una clase de ancla guarda en su especificación una clase de link, la que usará en la creación de anclas para asociarles el link correspondiente. Esta clase es usada por la clase NodeClass (o NodeViewClass) para guardar la especificación de las anclas no inmersas en un contenido, y por la clase DataSpec cuando se utiliza a nivel de NodeClass, para guardar la especificación de las anclas sobre el contenido del dato.

Las responsabilidades de esta clase son:

- Clonearse, enviando el mensaje correspondiente al nodo o a la vista para que cree un ancla que responda a la especificación de la clase de ancla.

- **Clase para la especificación de un dato a nivel de clase (AccessorSpecification)**

Esta clase se utiliza para guardar la especificación de la forma en la que el nivel de hipermedia accede a los datos o el comportamiento definido en el nivel de objetos. Las instancias de AccessorSpecification se clonean a sí mismas creando instancias de AspectAdaptor para que el nodo las utilice en el momento que se requieran.

Esta clase es colaboradora de DataSpec a nivel de clase de nodo.

AccessorSpecification guarda la clase de la aplicación, el nombre del mensaje a ser enviado y una indicación de si el mensaje es de instancia o de clase. Si el mensaje es de instancia, cuando el accessorSpec se clonee en un ObjectNode particular, creará un AspectAdaptor cuyo subject o modelo será la instancia de la clase especificada que está mapeada en aquel nodo. Si el mensaje es de clase, el subject será ésta misma.

2.2.3. *Patterns utilizados*

En esta sección presentaremos los patterns de diseño conocidos en la literatura actual que fueron utilizados en la arquitectura del framework. La utilización de estos patterns ha otorgado mayor solidez y confiabilidad al diseño, por tratarse de soluciones probadas y conocidas como correctas, flexibles, mantenibles y elegantes. Además, documentar el uso de estos patterns conocidos permite comunicar eficazmente las razones en las decisiones de diseño tomadas en esta arquitectura. Varios artículos se han presentado donde se documenta el framework de hipermedia en términos de los patterns constituyentes del mismo, como [Garrido+97a, Garrido+97b].

El formato que ha sido utilizado para esta sección es al estilo catálogo de patterns, lo que significa que tendremos una serie de subsecciones fijas para describir la utilización de cada pattern. En particular, el formato es similar al usado para describir un pattern mismo, con contexto, problema con su lista de fuerzas interactuantes, solución, consecuencias, y relaciones con otros patterns. En esta sección aplicaremos este formato para describir el *uso* de un pattern: contexto en el que fue utilizado, los factores o fuerzas que llevaron a su utilización, la solución específica usada en términos de la estructura del pattern, las consecuencias provocadas por la implementación del mismo, y las implicaciones que dieron origen al uso de otros patterns.

En esta sección se replantean muchos de los diseños vistos anteriormente, pero ahora con el propósito de precisar la *interacción* entre los componentes del diseño, el propósito, alcance y consecuencias de esa interacción.

Se describe abajo la *utilización* en el framework de los patterns: Observer, Composite, Type-Object, Decorator, Strategy y Adaptor. Los mismos fueron seleccionados para ser aquí presentados porque son los patterns que "generan" la arquitectura del framework (el concepto de patterns generativos se explica en 1.4.5)[3].

2.2.3.1 Diseño de la relación objeto ↔ nodo ↔ interface: Pattern Observer

- Contexto de uso:

El principal objetivo del framework de hipermedia es el de extender aplicaciones orientadas a objetos existentes, realzándolas con funcionalidad de hipermedia. Esto implica una completa integración de comportamiento desde ambos lados: navegación y funcionalidad de la aplicación. Así, los nodos proporcionan una nueva interface a los objetos, proveyendo distintas caras según el contexto actual y agregando anclas de link. En forma similar, los links proveen una manera de explorar relaciones entre los objetos del modelo subyacente.

- Problema:

¿Cómo se realiza el mapeo de los datos y el comportamiento encapsulados en los objetos de una aplicación existente hacia el nivel de hipermedia, y se soportan los cambios en el mismo?

[3] Cabe subrayar que no se describen estos patterns sino el uso que se le dió a los mismos. Con cada uno se proveen referencias a su especificación.

Los factores que compiten en el contexto anterior son:

- Los nodos deben mapear o asociarse a los objetos de la aplicación;
- Los nodos necesitan ser actualizados cada vez que el objeto asociado cambia;
- Los links necesitan ser creados con la creación de relaciones entre objetos;
- Las interfaces gráficas para los nodos necesitan también ser actualizadas cada vez que el nodo cambia;
- Los objetos de la aplicación subyacente no deberían preocuparse por la existencia de nodos;
- Los nodos no deberían ser responsables por las ventanas abiertas sobre ellos;
- El nivel de hipermedia también debe soportar nodos que no mapean objetos de la aplicación.

• Solución:

Usamos el Pattern Observer [Gamma+95] de la siguiente manera:

- La última de las fuerzas citadas arriba nos deja entrever que una subclasificación es necesaria para poder manejar separadamente los nodos que observan objetos de los que aparecen a nivel de hipermedia. Las subclases definidas fueron llamadas ObjectNode y HyperNode respectivamente. De la misma manera los links que mapean relaciones del modelo subyacente son instancias de ClassifiedLink, y los links agregados a mano en el nivel de hipermedia son instancias UnclassifiedLink, ambas subclases de HyperLink.
- Un ObjectNode es una agregaciones de NodeViews, y conoce el objeto(s) que mapea; cada NodeView define un DataSpec (slot de dato o comportamiento) para cada aspecto que se mapea del objeto en un contexto particular. Los ObjectNodes se convierten entonces en dependientes de los objetos asociados por medio de los DataSpecs que observan cada dato.
- Los nodos de una hipermedia podrían observar más de un objeto al mismo tiempo: por ejemplo suponga un objeto que representa una pintura. En la aplicación este objeto colabora con el objeto que representa el pintor. Aún si definiéramos clases de nodo "Pintor" y "Pintura", querríamos que el nodo "Pintura Las Meninas" contenga información del nombre de su autor, Velázquez, como mínimo. De esta manera los nodos pueden observar varios objetos fuertemente relacionados, sólo diciendo la forma de asociarlos en un mismo nodo.
- Los HyperNodes son también agregaciones de NodeViews, aunque no están asociados a ningún objeto. Los datos que muestran (como una página multimedial) son mantenidos por ValueHolders [Woolf95] o Proxies [Gamma+95] en el caso de datos multimediales guardados en archivos.
- Los links puede mapear objetos asociativos de la aplicación, es decir, objetos que expresan relaciones entre otros; en este caso un link se convierte en observador del objeto de la aplicación, y será instancia de ObjectLink, subclase de ClassifiedLink.
- La interface gráfica se transforma en dependiente de una representación particular con la que se está mostrando un nodo.

La jerarquía de Node obtenida en una primera iteración se ve como en la Figura 18.

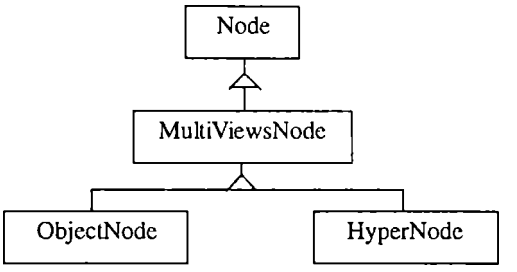


Figure 18: Jerarquía de Nodo preliminar

Por su lado, la jerarquía de Link se estableció como muestra la Figura 19.

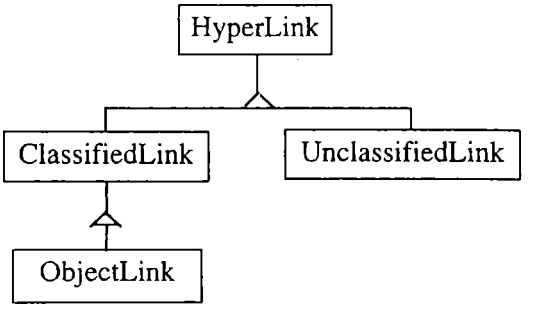


Figura 19: Jerarquía de Link

- Consecuencias
  - El uso del pattern Observer dictamina la estructura de la arquitectura global de una aplicación que use el framework de hipermedia.
  - Podemos combinar navegación con comportamiento convencional de una aplicación.
  - Los nodos son más que Observadores tradicionales, en el sentido que proveen diferentes dimensiones para mostrar objetos, y además agregan información navegacional a ellos.
- Implicaciones en cuanto al uso de otros patterns

El hecho de tener modelada la hipermedia en un nivel de objetos subyacente deja abierta la posibilidad de extraer un poco más de semántica de ese modelo. Una forma sería permitir la definición de nodos compuestos basados en jerarquías de composición del modelo. Así, un nodo pasaría a ser observador de un grupo de objetos relacionados entre sí por una jerarquía de parte-de. Esto se desarrolla en el siguiente ítem.

Otra cuestión interesante aparece con la necesidad de crear tipos de nodos (que observan las principales clases del nivel de objetos), lo que trataremos en 2.2.3.3.

2.2.3.2. Diseño de la agregación de nodos: Pattern Composite

- Contexto de uso:

Algunas metodologías de diseño y modelo de hipermedia proponen el concepto de nodo complejo, con la semántica de la relación "parte-de", para capturar una organización jerárquica de la información más allá de aquella basada en links. Otra agrupación de nodos propuesta es aquella con la semántica de conjuntos.

- Problema:

¿Cómo hacemos para mapear estructuras de composición del modelo de objetos y aprovecharlas para una mejor organización del conocimiento de la aplicación hipermedia? Además, ¿cómo proveemos la posibilidad de agrupar los nodos relacionados de cierta forma en conjuntos que provean un acceso a todos ellos?

Las fuerzas que compiten en el contexto anterior son:

- Los objetos de la aplicación subyacente pueden estar definidos en base a una estructura de composición, y los nodos que mapean esos objetos deberían poder reflejar esa estructura. El uso de links entre compuestos y componentes no reflejaría una estructura de composición.
- Los nodos que no mapean objetos de la aplicación, es decir los HyperNodes, estarán relacionados por links hipermediales, pero no deberían reflejar una estructura jerárquica. Es mejor reservar esa organización para los ObjectNodes, porque la definición de jerarquías de composición artificiales (aquellas que no están presentes en el modelo de objetos) pueden confundir al lector.
- Debería brindarse la posibilidad de agregar nodos que se relacionen de alguna forma, en conjuntos que provean un subespacio de navegación que guíe al lector en esta tarea.

- Solución:

El Pattern Composite [Gamma+95] fue usado dos veces en la jerarquía de Nodo (como muestra la Figura 20):

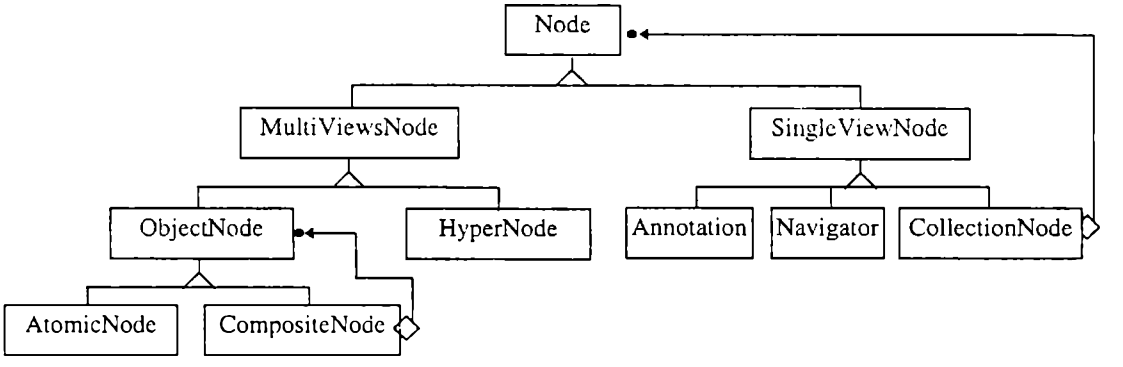


Figura 20: Uso del pattern Composite en la jerarquía de Nodo

- Los nodos-objeto pueden componerse formando agregaciones que reflejen la estructura subyacente. Con este fin ObjectNode fue subclasificada en AtomicNode y CompositeNode, siendo esta última la representación de una agregación de ObjectNodes (es decir de una jerarquía de parte-de).

- Cualquier tipo de nodo puede componer una colección con semántica de conjunto. Esta agregación de nodos será una instancia de CollectionNode. Esta instancia representará la misma agrupación de nodos cualquiera sea el usuario actual, es decir independientemente del contexto. Es por esto que los CollectionNodes tendrán una única vista, lo que queda determinado por ser su clase una subclase de SingleViewNode. Esto por supuesto no impide que los componentes puedan sí ser nodos de múltiples vistas.

- Consecuencias
  - Obviamente alcanzamos todas las consecuencias previstas por el pattern Composite.
  - Los CompositeNodes pueden ser usados cuando se necesita estructuración más allá de la navegación pura, reflejando una jerarquía de composición previamente definida por el modelo de objetos.
  - Los CollectionNodes son usados para definir un subconjunto de nodos relacionados, como una manera de organizar el subespacio de navegación.
  - CollectionNode puede ser usada para implementar "Contextos de Navegación" [Schwabe+96].
- Implicaciones en cuanto al uso de otros patterns
  - Como veremos en la subsección siguiente, la subclasificación de ObjectNode provocó una subclasificación de NodeClass.
  - La manera en que los CollectionNodes van a ser accedidos se define separadamente, y así el mismo collection-node puede ser accedido de diferentes formas. Existe consecuentemente una jerarquía separada de AccessStructure (ver Sección 2.2.1.1).
  - Un CollectionNode puede ser definido en base a distintas estrategias [Garzotto+94] y por esto el pattern Strategy [Gamma+95] fue usado para la hacer más flexible su creación (ver Sección 2.2.3.5).

2.2.3.3. Diseño de la clasificación de nodos y vistas-de-nodos: Pattern Type-Object

- Contexto de uso:

Para poder obtener una aplicación hipermedia usando el framework, los objetos de la aplicación son mapeados a nodos y sus relaciones a links. Los usuarios del framework deberían especificar qué *clases de la aplicación* quieren mapear a *clases de nodos*, y los nodos deberían ser generados automáticamente por cada instancia de aquella clase.

- Problema

¿Cómo automatizamos de alguna manera la creación de nodos aprovechando el modelo subyacente? Además, ¿cómo creamos clases de nodos sin provocar una explosión en la jerarquía de Node?

Los factores que pesan en el contexto anterior son:

- La definición de diferentes subclases de ObjectNode por cada conjunto de nodos que mapean instancias de la misma clase del modelo de objetos terminaría en una explosión de clases, la que sólo será un espejo de aquel modelo.
- La clase ObjectNode es subclasificada en AtomicNode y CompositeNode, independientemente de la aplicación base, como vimos anteriormente.
- Es preferible instanciar clases existentes en el framework a crear nuevas y múltiples clases cada vez que el framework es usado.
- Cuando una nueva instancia de una clase de la aplicación mapeada al nivel de hipermedia es creada, un nuevo nodo debería ser creado para mapear esa instancia.
- Todas las fuerzas mencionadas arriba en cuanto a los nodos se aplican de la misma forma a los links.
- Cada nodo puede tener diferentes vistas o "caras" con las que se muestra dependiendo del contexto actual. Esto implica que al crear un nodo de cierta clase deberían crearse con él las vistas de nodo que su clase define.

- Solución:

  El pattern Type-Object [Johnson+97] fue utilizado de la siguiente manera:

  - Se definió la clase NodeClass y sus subclases AtomicNodeClass y CompositeNodeClass, cuyas instancias cumplen el rol de clases para las instancias de las subclases de ObjectNode respectivamente. Cada NodeClass se relacionará con una o más clases del modelo, de las que observará la creación de instancias para crear ObjectNodes asociados automáticamente. El conjunto de NodeClasses definidas delinearán el modelo de datos y comportamiento de la hipermedia.
  - Para manejar el hecho de que un ObjectNode es una agregación de vistas fue necesario aplicar el mismo pattern nuevamente en forma anidada. De esta forma, se definió la clase NodeViewClass cuyas instancias actúan como clases para las instancias de NodeView. Cada NodeClass es entonces una agregación de instancias de NodeViewClass. Estas últimas tendrán la responsabilidad de crear NodeViews.
  - La clase LinkClass fue definida en forma similar, de manera tal que sus instancias actúan como clases para los ClassifiedLinks.

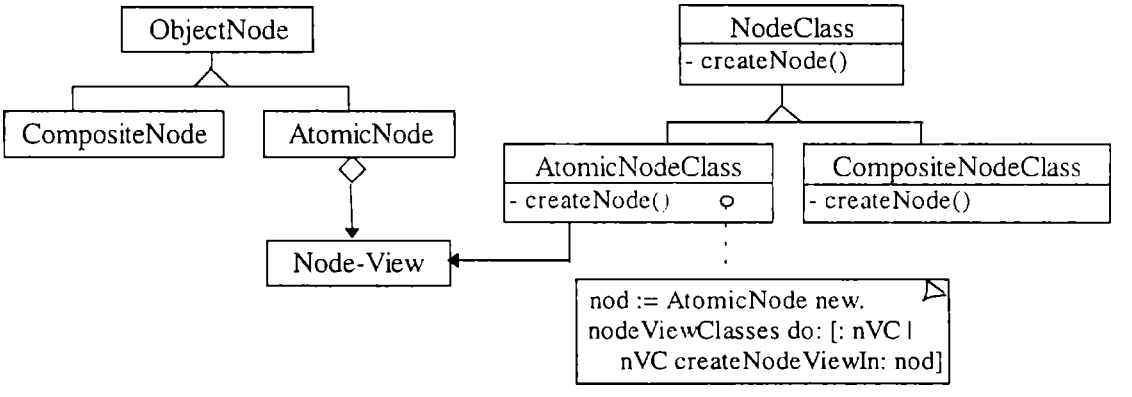


Figure 21: Relación entre ObjectNode y NodeClass

- Consecuencias
  - Se aplican todas las consecuencias previstas por el pattern Type-Object.
  - Fue posible implementar la automatización de la creación de nodos y sus vistas.
  - Se consiguió que la arquitectura del framework aparezca más sólida y estable, más cercana a una caja negra [Johnson+88] por el hecho de que el usuario no necesita conocer el framework en detalle para usarlo, lo que sería indispensable para poder subclasificar sus componentes.
- Implicaciones en cuanto al uso de otros patterns

La subclasificación de la clase NodeClass en subclases que se comportan como fábricas de instancias de la subclase de ObjectNode correspondiente, sigue el mismo propósito y estructura propuesto por el pattern Abstract Factory [Gamma+95].

2.2.3.4. Agregado de funcionalidad de hipermedia a una interface existente: Pattern Decorator

- Contexto de uso:

Muchas aplicaciones existentes tienen determinada una interface gráfica especial, cuya modificación no es viable, pero que quisiéramos extender con funcionalidad de hipermedia.

- Problema:

¿Cómo hacer para enriquecer con capacidad de navegación una interface gráfica existente, sin tener que modificarla?

Las fuerzas que interactúan en este problema son:

- Cuando una aplicación OO ha sido creada sin haber previsto una interface hipermedial, es altamente probable que provea una interface gráfica desde la que se puedan disparar las transacciones previstas en su funcionalidad. Al extender una aplicación como ésta con funcionalidad de hipermedia existen dos posibilidades: descartar la interface existente y definir una nueva preparada para soportar navegación, o conservarla pero integrada de alguna forma dentro del espacio de nodos y links definido por la hipermedia que se defina sobre la aplicación.
- La interface existente puede no ser modificable para poder remplazar los widgets usuales con widgets preparados para la navegación (es decir que permiten definir anclas de links, mostrarlas y activarlas). Aún cuando es posible modificar la interface, puede no ser la solución correcta cuando la aplicación quiere conservarse intacta para poder utilizarla fuera del espacio hipermedial.
- No es razonable subclasificar la clase de la interface provista, pues esto implicaría redefinir todas las responsabilidades de los nodos en cada nueva subclase.

- Solución:

Se utilizó el pattern Decorator [Gamma+95] de la siguiente manera:

- Se definió la clase Navigator en la jerarquía de Node. Un Navigator actúa como una vista transparente por encima de una interface previamente definida. En

realidad esta vista está formada por una conjunto de "wrappers" o Decorators que se crean sobre cada parte de la ventana sobre la que se quieren definir anclas de link. La utilización de la FH deberá hacerse en forma explícita, activándola en el momento de necesitarse para no producir colisiones con la funcionalidad prevista en la ventana.

- Los widgets extendidos hasta el momento sobre los que se pueden definir anclas son: las listas (que se convertirán en índices bajo demanda) y los textos, sobre los que se pueden agregar anclas de links no-clasificados o anotaciones.
- La forma en la que los Navigators son definidos implica que deberían tener una única vista. Es por esto que su clase fue definida como subclase de SingleViewNode, como podemos ver en la Figura 20.

- Consecuencias:

  Resulta posible extender aplicaciones existentes aún cuando provean una interface no-modificable ni descartable.

- Implicaciones en cuanto al uso de otros patterns

Las instancias de Navigator podrían ser consideradas también como Adapters [Gamma+95], dado que "adaptan" una interface definida fuera del ambiente de la hipermedia a un nodo que entiende el protocolo de navegación.

2.2.3.5. Estrategias de creación de nodos-Colección: Pattern Strategy

- Contexto de uso:

Existen distintas estrategias propuestas para la definición de nodos-colección.

- Problema:

¿Cómo soportar distintas estrategias de creación para un CollectionNode sin necesidad de utilizar sentencias condicionales?

Las fuerzas que surgen son:

- Necesitamos definir distintas variantes del algoritmo de creación de CollectionNodes.
- Sería interesante guardar sólo la especificación de los componentes de la colección cuando sea posible, y calcularla recién en el momento de activación del CollectionNode.

- Solución:

Se utilizó el pattern Strategy [Gamma+95] en forma casi directa, arribando a una estructura como la que se mostraba en la Figura 11.

En CollectionNode se definió un método de creación por cada estrategia distinta. Cada método instancia la estrategia que corresponde y delega en ella la responsabilidad de mantener la especificación de los integrantes de la colección. El nodo solo mantiene una referencia a la estrategia utilizada para su definición. Cuando el nodo es activado, éste le pide a su estrategia de definición que calcule sus integrantes, lo que podrá realizarse libre de toda sentencia condicional.

- Consecuencias:

Se lograron todas las consecuencias previstas por el pattern, como evitar la subclasificación de CollectionNode por cada estrategia de creación, eliminar las sentencias condicionales que llevan a un código obscuro, y proveer distintas implementaciones posibles.

- Implicaciones en cuanto al uso de otros patterns

No tiene.

2.2.3.6. Conexión entre los tres niveles de la arquitectura: Pattern Adapter

- Contexto de uso:

Los tres niveles de definición que la arquitectura del framework propone para las aplicaciones hipermedia se desarrollan en forma separada e independiente.

- Problema:

¿Cómo hacer para conectar el nivel de objetos con el nivel de hipermedia, y el nivel de hipermedia con el de interface, cuando los tres niveles se desarrollan independientes uno del otro?

- Solución:
  - Podemos considerar a los nodos como adaptadores, que saben cómo traducir la funcionalidad de navegación a los objetos del primer nivel, cuando estos últimos han sido desarrollados sin un protocolo que entienda los mensajes de la hipermedia.
  - Por el otro lado, la clase Representation ha sido implementada en el framework de manera que traduzca los mensajes de la interface gráfica asociada, a mensajes que el nodo entiende. Por ahora esta clase sólo adapta interfaces de VisualWorks, pero podría ser extendida para traducir interfaces hechas incluso fuera del ambiente de VisualWorks.

- Consecuencias:

El uso de este pattern en la definición global de la arquitectura permite implementar la separación de consideraciones a tener en cuenta en el desarrollo de cada nivel de una aplicación.

- Implicaciones en cuanto al uso de otros patterns

El resto de los patterns presentados anteriormente fueron utilizados en función de este último, que define la estructura global de la arquitectura.

2.2.4. *Patterns descubiertos*

A continuación presentamos dos patterns que hemos descubierto en el desarrollo del framework. Ambos son variaciones a patterns que aparecen en [Gamma+95], aunque aplicados en el dominio de hipermedia. Aquí no se presenta su uso específico en el framework sino que se describen en forma más general, como típicamente se describe un pattern. Incluso su estructura es abstracta, de manera que necesita ser instanciada por clases de un diseño particular para poder ser usada. Para un mayor desarrollo del formato de patterns ver [Gamma+95].

El formato que utilizamos para estos patterns es el mismo del catálogo de patterns de Gamma *et.al*, con secciones fijas, a saber: Propósito, Motivación, Aplicabilidad, Estructura, Participantes, Colaboraciones, Consecuencias, Implementación, Usos conocidos, Patterns relacionados.

Lo que se presenta en esta subsección ha sido publicado como Capítulo 11 del segundo volumen anual del libro que contiene los patterns más representativos que se presentan en la conferencia anual de patterns PLoP (Pattern Languages of Program Design Conference), Monticello, Illinois, U.S.A. [Rossi+96a].

2.2.4.1. Navigation Strategy

- Propósito

El propósito de este pattern es el de definir una familia de algoritmos que desacopla la activación de links hipermediales del cómputo de sus destinos, así permitiendo diferentes formas de obtener el destino, y su creación tardía.

- Motivación

En aplicaciones hipermedia convencionales (Microsoft's Art Gallery, por ejemplo), los links son fijados en la codificación hacia un destino particular. Cuando el destino de un link depende no sólo del nodo objetivo pero también de información del contexto, o si debe ser computado dinámicamente, es necesario realizar ciertos testeos en el nodo origen (o el ancla). Esta situación es más complicada aún en aplicaciones como herramientas CASE o ambientes de soporte de decisiones que permiten navegar a través de documentos de diseño, porque el destino de un link puede ser creado bajo demanda o permanecer no-especificado hasta una posterior definición. Suponga, por ejemplo, que se quiere navegar desde la tarjeta CRC [Beck+89] de diseño de una clase al browser de clase mostrando su implementación actual. Puede resultar necesario preguntar al manejador de versiones para obtener el destino del link. Más aún, si no existe una implementación de aquella clase, recorrer aquel link puede significar crear el nodo destino abriendo el browser en modo edición.

Una primera aproximación para resolver este problema es usar el pattern Strategy, cuyo propósito es encapsular diferentes algoritmos en una jerarquía separada, dejándolos variar en forma independiente de la clase cliente que los usa. De esta manera, cada link sería configurado con el algoritmo necesario. Pero como también se pretendía soportar la creación tardía de destinos de link, surgió la extensión del pattern Strategy con una subjerarquía de "Abstract Factories" [Gamma+95]. Con esta solución,

el algoritmo encapsulado en cada clase de fábrica se transforma en un "Factory Method".

El hecho de no tener una clase por cada clase de nodo, sino instancias que actúan como tal, provocó que en realidad se definiera una única clase fábrica que guarda la especificación de la clase de nodo que se instanciará al atravesar el link al que pertenece. Este pattern fue llamado "Navigation Strategy".

El diagrama de clases resultante al aplicar este pattern en el framework es el que podemos ver en la Figura 13.

- Aplicabilidad

El pattern Navigation Strategy es aplicable en el dominio de hipermedia cuando:

- se necesitan diferentes variantes de un algoritmo que compute el destino de un link;
- se necesita soportar la creación tardía de destinos de link.

En un contexto general, el pattern Navigation Strategy puede ser usado cuando existe la necesidad de establecer una relación entre dos o más objetos en diferentes momentos, es decir estática o dinámicamente, y cuando en último caso puede llevar a la creación tardía del objeto relacionado.

- Estructura

La estructura abstracta del Navigation Strategy puede verse en la siguiente figura:

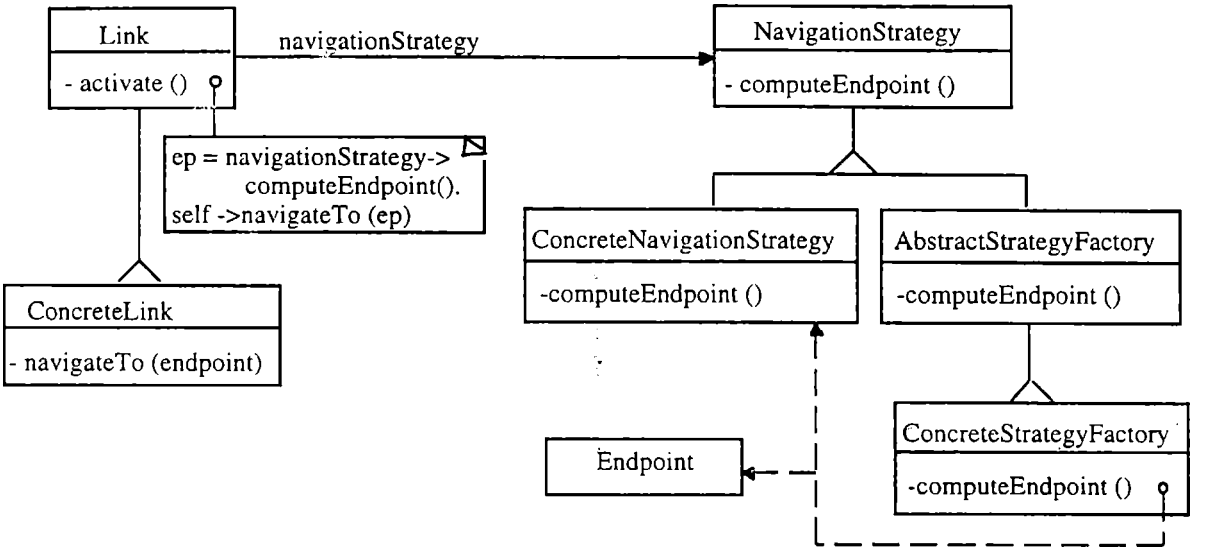


Figure 22: Pattern Navigation Strategy

- Participantes
  - Link mantiene una referencia a NavigationStrategy y define un "Template Method" [Gamma+95] para llevar a cabo su activación. Este método obtiene el destino o endpoint de su NavigationStrategy y navega hacia él.
  - ConcreteLink implementa el método `navigateTo`. Puede definir diferentes formas de realizar la navegación (por ejemplo, mostrando los atributos del link).

- NavigationStrategy declara una interface común para el algoritmo de cómputo del endpoint.
- ConcreteNavigationStrategy implementa el algoritmo que obtiene el destino del link (por ejemplo, puede ser destino fijo o computado).
- AbstractStrategyFactory obtiene el destino del link creándolo en ese momento, junto con una instancia de otra clase de ConcreteNavigationStrategy que será usada en sucesivas oportunidades.

- Colaboraciones
  - Link y NavigationStrategy interactúan para implementar la navegación a través de la red de nodos. Cuando es activado, el link pide a su NavigationStrategy que calcule su destino (pasando los argumentos necesarios) y luego realiza el proceso de navegación hacia el destino obtenido.
  - Las anclas de links (en los nodos orígenes) son los clientes que activan los links. A diferencia del pattern Strategy, estos clientes no son los responsables de la creación o selección de la ConcreteNavigationStrategy a utilizar; sólo constituyen una forma de acceder y activar un link.
  - ConcreteStrategyFactory crea tanto el Endpoint como una instancia de otra ConcreteNavigationStrategy para posteriores accesos.
- Consecuencias
  - NavigationStrategy ofrece las mismos beneficios del pattern Strategy, proveyendo una alternativa a las sentencias condicionales o a la subclasificación, y además permite la creación dinámica de nodos y links en una ambiente de hipermedia activo.
  - Esta separación en el proceso de navegación permite definir diferentes tipos de links (como aquellos que muestran información de sí mismos en el momento de ser navegados) y, por otro lado, definir diferentes tipos de destinos (como destinos simples o múltiples).
  - Las subclases de AbstractStrategyFactory pueden mejorar requerimientos de memoria defiriendo la recuperación del nodo destino (por ejemplo, cuando éste es mantenido en una base de datos), creando el Endpoint asociado en memoria sólo cuando es necesitado.
  - Finalmente, los mismos inconvenientes encontrados en Strategy pueden ser encontrados aquí: un número mayor de objetos y una comunicación más densa entre Link y NavigationStrategy (por ejemplo cuando el ancla es enviada como parámetro pero no es necesitada). Además, la subjerarquía de Abstract-StrategyFactory puede conducir a una proliferación de clases (como se discute en "Abstract Factory"), aunque esto puede ser resuelto usando el pattern "Prototype" en vez de "Abstract Factory" [Gamma+95].
- Implementación

Las consideraciones de implementación descriptas en [Gamma+95] para el pattern Strategy se aplican también a este pattern, aunque podemos mencionar algunas otras:

- *Intercambio de datos entre Link y NavigationStrategy*: la única información que se intercambia normalmente entre un link y su estrategia es el ancla de link (desde Link) y el endpoint (desde NavigationStrategy).
- *Endpoint y NodeClasses:* la clase Endpoint puede ser subclasificada en SingleEndpoint y MultipleEndpoint, de manera que cada link estará asociado con sólo un objeto Endpoint por intermedio de su NavigationStrategy. Además, el endpoint puede contener información de contexto o representación con la cual el nodo destino va a ser alcanzado.

  Por su parte, la clase AbstractStrategyFactory debería ser subclasificada por cada clase de nodo. Como esto puede resultar inadmisible, el pattern Prototype puede ser usado manteniendo sólo un prototipo a ser "cloneado" de la clase de nodo a utilizar para la creación.
- Cuando los nodos son extraídos de una base de datos, otras consideraciones deben ser tenidas en cuenta, como el mantenimiento de aquellos nodos. Estas consideraciones están más allá del alcance de este trabajo.

- Usos conocidos

Navigation Strategy es muy usado en actuales ambientes de hipermedia. Por ejemplo en Microcosm [Davis+94], un sistema de hipermedia abierto que provee un mecanismo de linkeo entre distintas aplicaciones, los links son siempre extraídos dinámicamente de una base de datos de links que contiene información sobre los offsets y tipos de las anclas.

En algunas extensiones propuestas a Netscape, por ejemplo [Hill+96], es posible definir nuevos links (privados) usando un mecanismo similar al presentado aquí. Más aún, la separación entre links y documentos linkeados permite la implementación de links "genéricos"(aquellos definidos en términos de contenido más que de ubicación), los cuales pueden ayudar enormemente a la definición de links comunes y a la navegación dirigida por el lector.

Algunas implementaciones del Modelo de Hipermedia de Dexter [Halasz+94] (como DHS [Grønbaek+94a]) proponen diferentes alternativas de obtener el destino de un link, similarmente a lo presentado aquí.

NavigationStrategy puede ser usado para mejorar el diseño de las aplicaciones hipermedia existentes. Por ejemplo, en los browsers de World Wide Web (WWW), el proceso de localización del destino de un link es normalmente una transacción no-atómica (que puede resultar no exitosa) y debe ser claramente separado del proceso de activación.

En el ambiente CASE presentado en [Alvarez+95], NavigationStrategy permite al diseñador tener links entre documentos animados dinámicamente. El destino de un link puede ser fijo a otro documento, disparar la creación de un documento, o incluso ser creado bajo demanda.

- Patterns relacionados:

NavigationStrategy es similar a Strategy en que permite al diseñador definir una familia de algoritmos para computar el destino de links, haciéndolos intercambiables y permitiendo que la clase Link sea extendida independientemente de aquellos algoritmos. A pesar de esto, difiere de Strategy en que incluye una subjerarquía de

fábricas en las cuales la estrategia actúa como un "Factory Method", así permitiendo la creación tardía de destinos y estrategias de navegación. AbstractStrategyFactory es también similar al pattern Acceptor [Schmidt95] en que ambos usan establecimiento tardío de la conexión.

Navigation Strategy también usa el pattern "Template Method" en Link para definir el algoritmo abstracto que realiza la navegación.

2.2.4.2. Navigation Observer

- Propósito

Desacoplar el proceso de navegación de la registración perceptible del mismo. Navigation Observer simplifica la construcción de viewers de la historia de navegación, separando los componentes de hipermedia (nodos y links) de los objetos que implentan el registro de navegación y su apariencia.

- Motivación

Las aplicaciones hipermedia deberían registrar el estado de la navegación en una manera perceptible para el usuario. A medida que la navegación progresa, este registro debe ser actualizado automáticamente. Por ejemplo, supongamos estar navegando a través de una aplicación hipermedia que muestra ciudades europeas, usando diferentes índices y links. Podemos alcanzar la misma cuidad a través de caminos de navegación diferentes y queremos poder saber qué ciudades han sido ya visitadas. Esta funcionalidad podría ser provista por un mapa de Europa que se mantenga siempre visible en pantalla, y sobre el que se vayan resaltando cada ciudad visitada durante la navegación. También podrían visualizarse los caminos navegados mostrando no sólo los nodos (ciudades) sino también los links (caminos). En general, los ambientes de hipermedia comerciales proveen una manera predefinida de mostrar la historia de navegación. Por ejemplo, Hypercard provee una lista de pequeñas figuras representando cada nodo o "card" que ha sido accedido.

Se podría implementar este comportamiento haciendo que los nodos representando ciudades se comuniquen con los objetos de interface que muestran el mapa a través de un mensaje del tipo `highlight(self)`. Esta solución haría muy dificultoso tener más de una forma de mostrar la historia de navegación, pues agregaría un gran acoplamiento entre nodos y viewers, y requeriría modificar los componentes de hipermedia por cada nuevo tipo de viewer que se defina.

La forma más conveniente de implementar este tipo de viewers de la historia de navegación es usando el pattern Navigation Observer. El mismo provee el registro percibible de los nodos y links visitados durante la navegación del espacio de la hipermedia. Además hace que este registro sea independiente de los nodos y links, cambiando el estilo de acuerdo a las necesidades y preferencias del usuario, de manera que el mismo diseño y estilo de interface pueda ser reusado en diferentes aplicaciones.

- Aplicabilidad

El pattern Navigation Observer debe ser usado en el dominio de hipermedia cuando:

- se necesita una manera de mantener la historia de navegación;

- se necesitan registrar no sólo los nodos pero también los links o estructuras de acceso en la historia;
- se necesitan diferentes viewers para la historia;
- se necesita soporte de backtracking en el camino hecho durante la navegación.

En un contexto general, el pattern Navigation Observer puede ser aplicado cuando se necesita registrar la ocurrencia de cierto evento y configurar esa historia con uno de muchos viewers diferentes.

- Estructura

La Figura 23 muestra la estructura abstracta del pattern Navigation Observer.

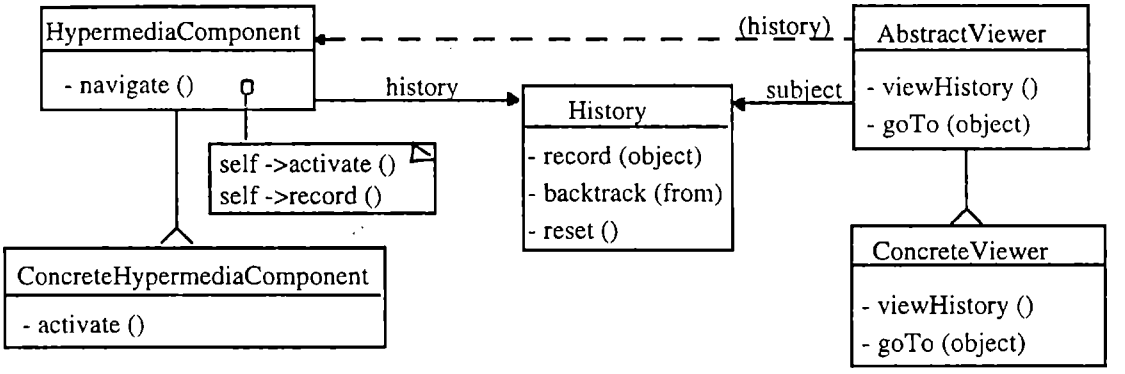


Figura 23: Pattern Navigation Observer

- Participantes
  - HypermediaComponent usa un "Template Method" de tal forma que cada vez que una instancia de una subclase es visitada notifica al objeto Historia.
  - ConcreteHypermediaComponent (Node, Link) implementa el método `activate`.
  - History registra la historia de navegación; es configurada con los ConcreteHypermediaComponents a ser registrados, o con predicados que se evalúan sobre aquellos; implementa el método `backtrack`.
  - AbstractViewer define una interface abstracta para mostrar la historia de navegación. Realiza operaciones de navegación como `goTo` hacia un componente de hipermedia de la historia. Esto se muestra con una línea de puntos en la Figura 23, ilustrando cómo el viewer puede activar un componente de hipermedia, obteniéndolo de la historia.
  - ConcreteViewer (List, Map, Graph) implementa los métodos `viewHistory` y `goTo`.
- Colaboraciones
  - Cada vez que un componente de la hipermedia es accedido, este envía el mensaje `record` a la historia, enviándose él mismo como parámetro, y la historia lo registra si corresponde.

- Cuando el usuario quiere percibir la historia en forma visual, el mensaje `viewHistory` es enviado a una instancia del correspondiente ConcreteViewer, que muestra la historia actual.
- ConcreteViewer interactúa con los componentes de hipermedia para realizar la operación `goTo`.
- El mensaje `reset` es enviado a la historia cada vez que el usuario desea re-inicializar la historia de navegación.

• Consecuencias

NavigationObserver ofrece los siguientes beneficios:

- Desacopla el proceso de navegación de su historia, y la historia de una manera particular de mostrarla.
- Este desacoplamiento permite la separación entre una manera particular de ver la historia de navegación para una aplicación, y la funcionalidad de hipermedia independiente de la aplicación (aquella de activar nodos y atravesar links).

Puede existir algún tipo de recarga en el viewer cuando esté interesado en cierto tipo de nodos (por ejemplo sólo las ciudades con cierta atracción turística). El viewer va a necesitar interpretar la historia para filtrar las ciudades o el tipo de nodos en los que está interesado.

• Implementación

Discutimos abajo algunas cuestiones relacionadas con la implementación del pattern Navigation Observer.

- *Configuración de la historia.* El objeto Historia puede ser configurado con un predicado a ser evaluado sobre los componentes de la hipermedia, si es necesario filtrar algunos de estos. De todos modos, si el predicado se vuelve muy complejo, sería más apropiada una subclasificación de la clase History.
- *Diferentes algoritmos de definición de la historia.* A pesar de que las historias pueden ser consideradas como simples "pilas" que registran cada visita a un componente de la hipermedia, puede ocurrir que algunas veces sea necesario proveer a los viewers de una historia "condensada". Esto puede ocurrir cuando el mismo objeto es visitado más de una vez o cuando ocurren backtracking complejos en un ambiente de múltiples ventanas activas. Una aproximación posible es la de definir un método distinto en History que retorne el registro de navegación descartando duplicados y ciclos. Otra posible solución es agregar la responsabilidad a los objetos viewers que analicen la pila cuando sea necesario.
- *Mapeo de viewers a la historia.* Esta cuestión es similar a la que afecta al pattern Observer [Gamma+95], dado que los viewers "observan" la historia. Esto puede ser implementado usando una tabla asociativa que mantenga el mapeo historia-viewer, o guardando referencias a viewers en cada objeto History (a pesar de que esto puede resultar muy costoso cuando hay muchas historias y pocos viewers).
- *Cuando actualizar viewers.* Los viewers pueden requerir información de la historia cada vez que necesitan mostrarse. También puede quererse actualizar una view cada vez que un componente de hipermedia es visitado. Estas dos opciones pueden ser usadas juntas, como se discute en [Gamma+95].

- *Dos o más historias.* Si múltiples usuarios acceden a la hipermedia concurrentemente, diferentes objetos History son necesarios para registrar cada sesión de navegación distinta. Este pude complicar la implementación pues el mismo componente de hipermedia puede ser visitado en diferentes contextos. En tales casos, los componentes de hipermedia deben estar al tanto del objeto History correspondiente a su sesión, o puede usarse un Mediator entre los componentes de hipermedia y la historia.

- Usos conocidos

Muchos ambientes de hipermedia proveen diferentes maneras de visualizar la historia de navegación. Por ejemplo, el componente de "ayuda" de Microsoft Windows muestra la historia de una sesión de ayuda como una lista de tópicos visitados. En forma similar, en algunos browsers de WWW como Netscape, es posible seleccionar una URL de la historia de navegación. A pesar de que Netscape sólo provee una lista textual de la historia, la estructura subyacente permite usar Navigation Observer para construir nuevos tipos de viewers.

Otros ejemplo, no relacionado con el dominio de hipermedia, aparece en ambientes OO que proveen formas de acceder y manipular la historia de ejecución. En Smalltalk, por ejemplo, diferentes tipos de "debuggers" (ConcreteViewers) pueden ser implementados accediendo a la pila de ejecución (History). En [Alvarez+95] se discute cómo construir animaciones que muestren la manera en que los objetos interactúan entre sí; allí la historia filtra objetos y métodos de acuerdo a la elección del usuario, y esos objetos son luego animados. El hecho de desacoplar objetos, historias y viewers ayudó a realizar una arquitectura más flexible y extensible para crear animaciones.

- Patterns relacionados

La relación entre viewers y la historia se asemeja a la propuesta por el pattern Observer [Gamma+95]. Además, History puede ser implementada como un Singleton, en el caso de ambientes no concurrentes, o como un Mediator que puede ser usado para desacoplar componentes de hipermedia de una historia particular cuando se trata con múltiples sesiones.

2.3. Implementación del framework

En esta sección hablaremos de las consideraciones tenidas en cuenta antes y durante la implementación del framework de hipermedia. No se intenta aquí presentar detalles de su codificación, pero sí dar un visión general sobre las ventajas e inconvenientes del lenguaje elegido, ejemplificando aquellas consideraciones con algunos diseños específicos.

Se incluye también en esta parte la especificación del tercer nivel de la arquitectura, el nivel de interface, por estar fuertemente ligado al ambiente de desarrollo. Explicaremos las extensiones y modificaciones realizadas sobre el framework de interface del ambiente.

Por último presentaremos la herramienta gráfica desarrollada para la fácil instanciación del framework.

2.3.1. Lenguaje utilizado; ventajas y desventajas del mismo; extensiones realizadas al nivel de interface

El lenguaje de implementación elegido para este trabajo es Smalltalk-80 [Goldberg+83]. El mismo es un lenguaje OO puro, es decir, que respeta consistentemente el paradigma OO; provee herencia simple, polimorfismo, binding-dinámico y es no-tipado.

Como dijimos en el comienzo de este trabajo, el objetivo perseguido fue el de extender aplicaciones OO, por considerar que la orientación a objetos es la tecnología de mayor desarrollo actual y futuro, y que mucho puede aprovechar y aportar del y al campo de hipermedia [Garrido+96a]. Con este objetivo, estamos convencidos de que la mejor forma de desarrollar software OO es con el lenguaje Smalltalk y utilizando el ambiente de desarrollo de VisualWorks™.

El ambiente provisto por VisualWorks brinda: categorización de clases y de métodos dentro de una clase, manejo de proyectos para trabajo colaborativo, manejo de cambios generales y por proyecto, debugging especializado, manejo de excepciones, pintador avanzado de interfaces llamado "canvas", browsers de clases, de jerarquías, de categorías de clases y métodos, entre las características más salientes [PP94].

Entre las **ventajas** obtenidas con la utilización del Smalltalk de VisualWorks podemos enumerar:

- *Implementación directa del diseño*: ningún artificio del lenguaje de implementación es necesario cuando se utiliza Smalltalk. Los componentes modelados en la etapa de diseño aparecen casi idénticos, aunque más detallados, en la codificación, por tratarse de un lenguaje de muy alto nivel.
- *Limpieza del código:* no son necesarias construcciones especiales como ocurre con lenguajes híbridos, ni es necesaria la destrucción explícita de instancias gracias al recolector de basura provisto por el ambiente.

- *Múltiples plataformas:* la misma imagen construida en una plataforma entre Windows, Macintosh, OS/2 o Unix, puede ser usada en cualquiera de las otras sin ningún cambio necesario. Esta característica hace que el framework sea completamente portable.

- *Mecanismo de dependencias:* este mecanismo provisto por el lenguaje permite la correcta implementación de "observadores" u objetos que se hacen dependientes de los cambios producidos sobre objetos observados o "modelos", sin necesidad de que estos últimos tengan que avisar a sus dependientes en forma explícita. Cualquier objeto puede tener dependientes, aunque las instancias de las subclases de Model están mejor preparadas para mantener dependientes en forma eficiente. Cada vez que un modelo cambia en alguna forma que puede ser de interés para al menos un dependiente, debe realizar un broadcast notificando de su cambio a *todos* sus dependientes, que a su vez decidirán si están interesados en el cambio. Este mecanismo de notificación entre un modelo y sus dependientes se denomina "changed/update" [Howard95]. Existen dos tipos de modelos dependiendo de su granularidad: *modelos de la aplicación* o "application models", y *modelos de valor o aspectos*, llamados "value models". Un "application model" es responsable de manejar las partes de una ventana, y es construido para una aplicación específica. Por su lado, un "value model" contiene un único aspecto de información que constituye su valor. Este concepto es tan importante en VisualWorks que todos los widgets (partes atómicas de una ventana) usan value models como modelos. Existen varios tipos de value models, cada uno representado por una subclase de ValueModel, como ValueHolder, AspectAdaptor y PluggableAdaptor. El concepto de "value model" se potencia además con el de "dependency transformer" para optimizar el mecanismo de dependencias [Woolf94].

  Este mecanismo permitió que los nodos se hicieran "observadores" de los objetos de la aplicación subyacente, sin tener que modificar estos últimos, y que a su vez los nodos fueran independientes de los objetos de interface, ya que no necesitan referenciarlos directamente.

- *Arquitectura en capas independientes:* una arquitectura en capas es aquella en la que el diseño de cada capa es independiente, e introduce cierto protocolo que restringe la interface de la misma [Campbell+91]. En Smalltalk, esto es totalmente factible de ser implementado, en parte gracias al mecanismo de dependencias visto anteriormente y en parte por la explícita separación que se hace entre los objetos del dominio y su interface. Esta separación es fundamental para el desarrollo de una arquitectura flexible, mantenible, reusable y modificable. El framework de hipermedia se ha podido desarrollar, gracias a esta característica, en tres capas o niveles independientes: nivel de objetos, nivel de hipermedia y nivel de interface.

- *Especificación de las ventanas:* Siguiendo con las consideraciones de interface, algo que resultó decisivo para la extensión y adaptación del framework de interface de VisualWorks fue su capacidad de construir descripciones literales a partir de una ventana construida con el canvas (herramienta gráfica para tal fin). Esa especificación se mantiene en un método de clase de la subclase de ApplicationModel que se cree para representar la ventana, en forma de arreglo literal. Recién en el momento de apertura de la ventana, una instancia de UIBuilder construye de aquel arreglo literal una

especificación formada por una colección de objetos anidados llamados "specs", a partir de los cuales se crean los objetos de interface implicados.

El proceso de construcción de interfaces puede ser extendido y adaptado gracias a su separación en etapas en el tiempo, aunque es necesario conocerlo profundamente. Las **extensiones** realizadas al framework de interfaces de VisualWorks fueron:

- Subclasificamos UIPalette en HUIPalette para poder agregar nuevos botones a la paleta de widgets que provee el pintador de pantallas, y cambiar widgets usuales por widgets adaptados a hipermedia; subclasificamos UIPainter en HUIPainter para que al editar e instalar una ventana, ésta pueda tomar parámetros de la hipermedia; por último subclasificamos UIPainterController en HUIPainterController de manera que al editar la interface para un Navigator, el menú de opciones permita definir anclas de links sobre algún widget.
- Interceptamos la salida del HUIPainter con un nuevo constructor llamado HSpecBuilder que a partir de la especificación de la ventana construye las representaciones de nodos y completa las vistas con los datos y comportamiento a mostrar. El HSpecBuilder colabora con las clases de specs de hipermedia para poder cumplimentar esta responsabilidad.
- Redefiniendo unos pocos métodos que la clase Representation hereda de ApplicationModel pudimos lograr que la especificación de la ventana se mantuviera a nivel de instancia y no de clase. Esto eliminó la necesidad de crear una clase por cada ventana. Volveremos sobre esta decisión más adelante.
- En el momento de abrir la ventana, la misma clase UIBuilder puede ser usada para el proceso de construcción de los objetos de interface pertinentes. Fue necesario agregar sí algunos métodos en UILookPolicy. Esta clase es la responsable de seleccionar el widget y el controlador correspondiente a la especificación, de acuerdo al soporte de ventanas actual. En realidad UILookPolicy usa la política por defecto, y cada subclase, que se corresponde con las distintas plataformas, utiliza su política específica. De esta forma se agregaron en UILookPolicy los métodos que conectan correctamente las specs de hipermedia agregadas con las views y controllers de hipermedia correspondientes. En la próxima subsección hablaremos de estas clases de views y controllers agregados.
- Para el caso de editar interfaces de nodos clasificados, extendimos las propiedades de los componentes que se agregan en la ventana de manera tal que éstos puedan observar un aspecto de cualquiera de los objetos mapeados por el nodo en cuestión (en las interfaces de VisualWorks comunes, lo único que se describe en las propiedades de un componente es el aspecto, porque se asume que el "sujeto" al que el aspecto corresponde es el application model donde será instalada la ventana; con la extensión que hemos realizado, desde la edición de la interface permitimos conectar los componentes con los objetos del dominio correspondientes).
- En el caso de editar interfaces de HyperNodes, como estos agregan información no presente en el modelo de la aplicación, es decir datos que no se pueden obtener de un

aspecto del dominio, se extendió la ventana de propiedades para algunos widgets de manera de poder ingresar los datos en el momento de edición. Estos datos son luego guardados en ValueHolders que mantiene el nodo.

- En el caso de editar interfaces de nodos Navigators, antes de la edición se realiza un parsing del arreglo literal que contiene la especificación de la interface de la aplicación ha ser extendida con hipermedia, intercambiando widgets normales por su correspondiente widget hipermedial del mismo tipo de dato. En la edición se permiten agregar anclas de links en estos últimos widgets. El resto del proceso es como en los otros nodos.

Como **inconvenientes** a los que nos enfrentamos por la utilización de VisualWorks podemos citar:

- A nivel de interface, la construcción de ventanas en VisualWorks hace necesario crear una nueva clase, generalmente subclase de ApplicationModel, por cada nueva ventana. Esto en realidad sucede en todos los ambientes de implementación del Smalltalk, aunque en VisualWorks, con la introducción de la filosofía de specs, Value Models y Dependency Transformers [Woolf94], se está tendiendo a darle menor responsabilidad al Application Model.

  La subclase de ApplicationModel es necesaria por tres motivos: mantener la especificación de la ventana, interactuar con el builder para construirla, y luego conectarla con el modelo del dominio y sincronizar los widgets entre sí. En este trabajo pudimos conseguir la creación de una única subclase de ApplicationModel llamada Representation, la cual constituye el modelo de todas las interfaces de nodos hipermediales. Esto se hizo posible de la siguiente manera: la especificación de la ventana se guarda a nivel de instancia en una variable de Representation, y no a nivel de clase; para la construcción de la ventana también pudimos lograr que las instancias de Representation interactúen con las de UIBuilder; la conexión de cada widget con el modelo se hace a través de AspectAdaptor o PluggableAdaptor, es decir Value Models. Estamos trabajando en la implementación de la sincronización entre widgets, en particular entre una lista y un texto en el que aparece más información del objeto seleccionado en la lista. Haría falta un poco más de trabajo para que cualquier widget pueda observar los cambios en otro, aunque lo hemos considerado más allá de esta presentación.
- En cuanto al soporte multimedial, en VisualWorks es casi nulo, pero como siempre, posible de ser implementado en otra extensión al framework.
- En cuanto a la persistencia, su implementación tampoco es trivial, pero aunque requiere un trabajo aparte la ventaja que se mantiene es que el modelo del framework no necesita cambiar; la capa de persistencia se puede construir en forma independiente.

2.3.2. *Definición de widgets adaptados a hipermedia*

Para poder soportar la creación de anclas de links sobre textos, listas, input fields y botones, evidentemente tuvimos que extender el conjunto de widgets existentes con aquellos que proveen esta funcionalidad.

Un *widget* en VisualWorks es una parte visual responsable de la representación de un componente, al que le da su apariencia y funcionalidad, y del que mantiene el estado [Howard95]. Un *componente* de VisualWorks es un "spec wrapper". Este contiene un widget, una copia del estado del widget, y una especificación literal llamada *"component spec"*. Este objeto "spec" constituye una descripción que permite construir una porción de interface.

En la edición de una ventana se pueden editar las propiedades de cada componente que se agregue. El conjunto de propiedades de un componente se obtiene de su objeto spec. Una de las propiedades que tuvimos que agregar en casi todos los specs fue el "sujeto" del que se obtiene el aspecto a observar. Como dijimos más arriba, al no definir una nueva clase de application model, y por el hecho de que un nodo puede mapear más de un objeto del dominio, proveemos desde la edición de la ventana la conexión del componente de interface con el objeto y el aspecto a observar, a través de un AspectAdaptor, sin tener que pasar por el application model. Basta entonces con seleccionar la clase de la aplicación (mapeada por la clase de nodo en cuestión) que define el aspecto a observar, y cada nodo-objeto se ocupará de crear el AspectAdaptor sobre la instancia de aquella clase que él esté mapeando.

La implementación de nuevos widgets en VisualWorks requiere, en el peor de las casos, la definición de tres nuevas clases:

- una nueva clase de spec, subclase de ComponentSpec o sus subclases;
- una clase de widget, subclase de aquella que representa al widget que se está extendiendo con hipermedia;
- una clase de controller, si es que hace falta un manejo especial de la entrada/salida.

De esta forma, los componentes que hemos agregado para permitir una interface navegacional son los que enumeramos a continuación, agrupados bajo la categoría 'Hypermedia-UISupport'.

- **ActionButtonHyperSpec**

  Subclase de ActionButtonSpec, esta clase sólo agrega la definición del sujeto del que se obtiene el aspecto que retornará la acción a realizar ante la opresión del botón. La vista y el controlador que se usan son los usuales: ActionButtonView y TriggerButtonController.

- **ButtonAnchorSpec**

  Subclase de ActionButtonSpec, esta clase permite la definición de botones que se comportan como anclas de links, es decir que al presionarlos se produce la navegación.

La vista y el controlador que luego son creados son los mismos que se utilizan para los botones que disparan acciones.

- **InputFieldHyperSpec, InputFieldHyperView e InputBoxHyperController**

Estas tres clases fueron creadas para poder soportar "input fields" con capacidad de definir anclas de links para ser usados en las ventanas de los nodos-objeto.

Tanto para los input fields como para los textos, que veremos más adelante, hemos cambiado la filosofía de edición. Un componente que muestra texto puede ahora encontrarse en uno de dos modos distintos: *edición* o *navegación*. Si el componente está en modo edición, al clickear con el mouse en un punto dentro de su extensión, simplemente se visualizará el cursor para poder editar; este es el modo por defecto de los widgets de texto. Con la posibilidad de cambiarse al modo navegación, al clickear con el mouse dentro del widget y sobre un caracter que forma parte de un ancla de link, se produce directamente la navegación, como sucede generalmente en los ambientes de hipermedia.

InputFieldHyperSpec agrega a las propiedades del componente la especificación del modelo del que se obtiene el aspecto, y la capacidad de poder definir "hotwords" que actuarán como anclas de links.

InputFieldHyperView tiene las siguientes responsabilidades: "scannear" el texto a mostrar, resaltando las hotwords definidas en su spec; activar la hotword que se encuentre en el punto donde se produjo un click del mouse y crear anclas para links no clasificados.

InputBoxHyperController es responsable de conocer el modo actual en el que se encuentra el widget (edición o navegación). Cuando se encuentra en el modo edición provee el menú que aparece generalmente para la edición de textos más tres nuevos ítem que permiten: crear un ancla de link ("create anchor"), una anotación ("annotate"), o cambiarse al modo navegación ("accept"). Cuando se encuentra en el modo navegación, provee el menú de acciones correspondiente al modelo, más los ítems que permiten navegar sobre la misma ventana ("go") o abriendo una nueva ("go on new window"), y el de cambiarse al modo edición ("edit").

- **InputValueHSpec**

Esta clase fue creada para ser usada en la edición de las ventanas de HyperNodes. Permite que se agregue un input field a la interface, que se ingrese su contenido y se definan las anclas de links, todo en modo edición.

- **SequenceViewHyperSpec y ListHyperWrapper**

Estas clases son usadas para agregar componentes del tipo listas de selección simple a los nodos-objeto.

SequenceViewHyperSpec agrega a las propiedades del componente la definición del "sujeto" del que se obtiene el aspecto, y la capacidad de poder definir anclas de link sobre los elementos de la lista, lo que la convierte en una especie de índice.

En este caso no hemos definido una nueva clase de view, porque sólo hizo falta definir un wrapper (o decorador) para la view usada comúnmente (SequenceView). Este wrapper provee los ítems de menú que permiten la navegación desde el ítem seleccionado en la lista, y activa el ancla correspondiente.

- **NavigatorListSpec y ListNavigatorWrapper**

  Estas clases son usadas para decorar con capacidad de navegación una lista de selección, perteneciente a una interface sobre la que se ha definido un nodo Navigator.

  NavigatorListSpec extiende las propiedades de aquel componente para poder definir anclas sobre él, y crear luego un ListNavigatorWrapper decorándolo. Este último permite que cuando la ventana está activa pueda navegarse desde un ítem de la lista.

- **TextEditorHyperSpec, TextEditorHyperView y TextEditorHyperController**

  Estas tres clases definidas para agregar widgets de texto a los nodos-objeto, se comportan de manera muy semejante a sus correspondientes para input-fields con capacidad de navegación.

- **TextValueSpec**

  Se comporta igual que InputValueHSpec para los HyperNodes, aunque para agregar widgets de texto.

- **NavigatorTextSpec y TextNavigatorWrapper**

  Estas clases se asemejan completamente a NavigatorListSpec y ListNavigatorWrapper, sus correspondientes para listas, aunque permiten decorar con capacidad de navegación los textos que estén definidos sobre la interface de un nodo Navigator.

  El TextNavigatorWrapper provee dos modos de operación: el modo normal provisto por el widget de texto que decora, donde el navigatorWrapper se hace "invisible" e inoperante, y el modo navegación, donde el wrapper muestra las anclas de links (o "hotwords" por tratarse de un texto), permite crear nuevas anclas de links o de anotaciones, y captura los clicks del mouse para provocar la activación del ancla.

- **AnchorSpec, LinkClassSpec y LinkSpec**

  Estas son clases de soporte usadas para la definición de anclas en los specs de componentes.

2.3.3. *Herramienta auxiliar desarrollada: Browser de hipermedia*

Un ambiente de construcción de aplicaciones OO, al cual Ralph Johnson define como *"toolkit"* [Johnson+88], es una colección de herramientas de alto nivel, que permiten al usuario interactuar con un framework para configurar y construir nuevas aplicaciones.

La idea de una herramienta de este tipo es la de permitir al usuario elegir componentes pre-armados, completarlos y conectarlos. Una de las ventajas de los frameworks "caja negra" (ver Sección 1.4.4) es que resultan mejores para ser utilizados como base de un toolkit, ya que justamente proveen componentes concretos que sólo deben ser conectados, y sólo en casos extremos necesitan ser extendidos creando nuevas subclases.

Una de las herramientas desarrolladas como una aproximación, y en una primera iteración hacia la construcción de un toolkit para el OO-Navigator es el llamado "Browser de Hipermedia" o "Hypermedia Browser". Su apariencia es muy similar a la de los browsers de Smalltalk: una serie de listados sincronizados, relacionados por una semántica de "parte-de" de derecha a izquierda. De allí su nombre. En la Figura 24 podemos observar la herramienta. La utilización de la misma será explicada en la Sección 3.5. cuando ya se hayan presentado las pautas de construcción de una aplicación hipermedia con el framework.

Actualmente se encuentra en desarrollo una herramienta gráfica que cumple la misma funcionalidad del Browser de Hipermedia, pero de una forma más amigable. La herramienta muestra la red de nodos y links a medida que se va construyendo, con distintos colores que identifican los tipos de nodos y links, entre otras características. Un usuario no experimentado con el framework encontrará la herramienta gráfica mucho más entendible. En cambio, un usuario ya experimentado, y en especial un programador Smalltalk, puede encontrarla por demás explicativa y por lo tanto preferir el browser, por estar acostumbrado al ambiente de programación de Smalltalk.

Otras herramientas del toolkit, accesibles desde el menú del Browser de Hipermedia, se ocupan de la definición de las características de un componente en particular, como clases de nodos, clases de links, contextos, etc.

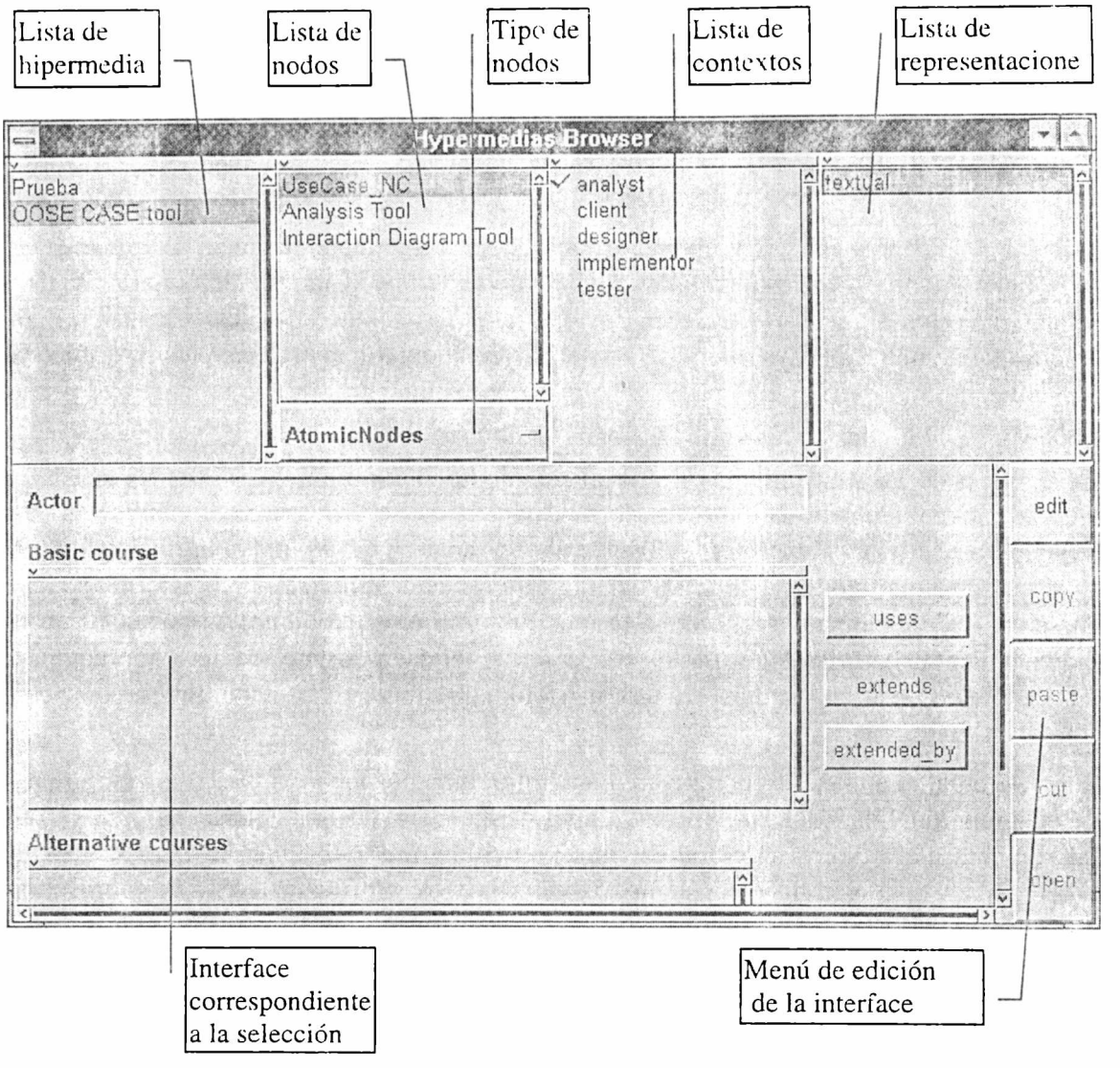


Figura 24: Browser de Hipermedia

a otra herramienta importante es la que permite construir una interface para un nodo. I erramienta, como se dijo anteriormente, es una extensión a la que ya provee el ambient isualWorks, llamada "canvas", para la creación de interfaces. El canvas fue extendido idgets que pueden conectarse con el nivel de hipermedia, y fue adaptado para qu ıncionalidad disponible dependa del tipo de nodo en cuestión (por ejemplo cuando se e interface de un Navigator no aparece la paleta de widgets, y cuando se trata de yperNode la paleta no provee ningún widget del estilo de botones, que permitar ctivación de acciones).

La Figura 25 muestra la herramienta de construcción de interfaces editand entana para una vista de una clase de nodos atómicos.

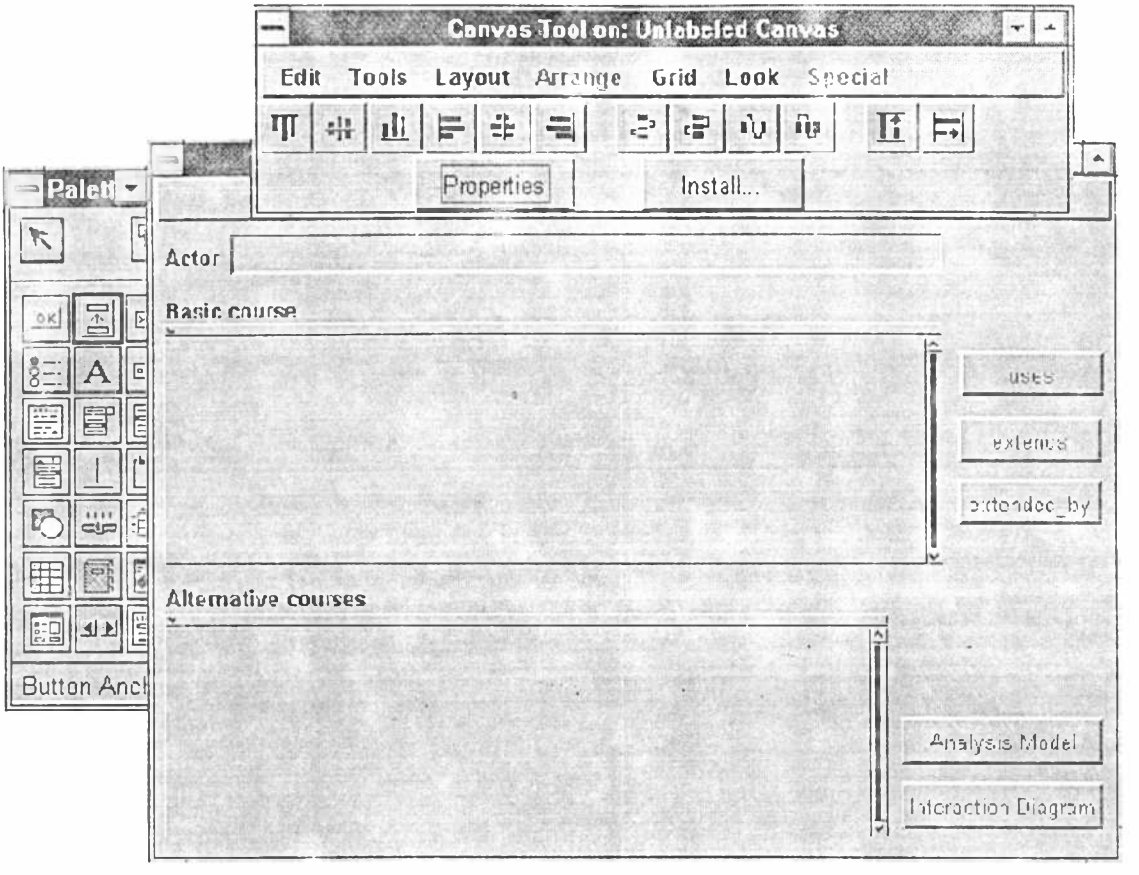


Figura 25: Herramienta de construcción de la interface de un nodo.

3. Utilización del framework

Este capítulo intenta presentar algunas pautas para el buen diseño de una aplicación que utilice el framework de hipermedia, y sugerencias para su mejor aprovechamiento.

La primera sección introduce las que le siguen, en el sentido que motiva el capítulo, explica el estado del arte en metodologías de diseño del dominio, y nombra los pasos a seguir en el uso del framework. Las secciones subsiguientes desarrollan cada paso en detalle.

Desde la segunda hasta la cuarta sección se presentan pautas para el correcto estudio del modelo de la aplicación a extender, y para la definición de los distintos componentes de la hipermedia resultante. En la quinta sección se describe el proceso de instanciación utilizando el "Browser de Hipermedia" que se presentó en el capítulo anterior.

La intención en este capítulo no fue la de definir una metodología de diseño de extensiones hipermediales, pero sí motivar la investigación en ese sentido. Se pretende que signifique algo más que un manual del usuario.

3.1. Cómo se instancia el framework

La instanciación del framework es un proceso formado por varios pasos de diseño, tanto del modelo de la hipermedia como de su interface. Detallaremos en las próximas secciones los pasos a seguir, a saber: estudio del modelo de la aplicación, descubrimiento de las relaciones entre componentes de la aplicación y creación de los componentes hipermediales. La última sección tiene más características de manual del usuario, ya que describe las herramientas que pueden utilizarse para una cómoda instanciación.

A pesar de que la funcionalidad de hipermedia es un concepto que ya ha sido foco de varios artículos y workshops en importantes conferencias, no existen hasta el momento pautas claras de integración con el comportamiento propio de las aplicaciones que son extendidas. Con este framework se intenta proveer un ambiente flexible para agregar fácil y rápidamente esta funcionalidad a aplicaciones OO, y así luego de varias pruebas y varias instanciaciones, llegar a lograr un marco de experiencia suficiente para el desarrollo de una metodología de extensiones hipermediales.

La metodología de diseño de hipermedia que más se aproxima a la intención aquí planteada es la Object Oriented Hypermedia Design Model (OOHDM) [Schwabe+95], que fue brevemente descripta en la Sección 1.4.2.3. En ella se plantea el desarrollo de un modelo de objetos conceptual como primera actividad. Lo que se pretende de este modelo es obtener un diseño modular y mantenible de los objetos de información a los que en una segunda actividad, y separadamente, se les agregan características de navegación. La diferencia es que en aquel modelo conceptual se resaltan los *datos* o valores encapsulados en los objetos, y no su comportamiento, como pretende el presente trabajo.

Por el momento se presentan aquí algunas sugerencias para la obtención de mejores resultados. Creemos que más que metodologías o heurísticas de diseño, lo que hace falta en el campo de hipermedia es el desarrollo de un lenguaje de patterns que proporcione soluciones standard. En este sentido hemos presentado algunos trabajos [Rossi+96b, Rossi+97], aunque van más allá del alcance de esta tesis y por eso no serán aquí expuestos.

3.2. Estudio del modelo de la aplicación

Para poder extender una aplicación, siempre es necesario conocerla en detalle. En el caso de extensiones usando el framework de hipermedia, no necesitamos conocer los detalles de implementación, pues no necesitaremos modificarlos, pero sí el modelo de diseño.

El primer paso a seguir previo a la extensión hipermedial, es, como el nombre de la sección lo indica, *el estudio cuidadoso del modelo de clases y objetos de la aplicación a ser extendida*. Recordemos que esta aplicación es la que formará el nivel

de objetos. El modelo de hipermedia se basará en aquel para definir los nodos y links, y del mismo obtendrá la mayor parte de los datos, comportamiento y relaciones. La metodología de diseño OOHDM se basa en una primera actividad de desarrollo de un modelo de objetos conceptual, como base para el modelo de la hipermedia [Schwabe+95]. Nosotros vamos un poco más allá, y de la aplicación no usamos sólo su modelo, sino su implementación, de manera que los nodos constituyan sólo un canal hacia los objetos desde la interface, y sólo se ocupen de mantener y responder a un protocolo de hipermedia y delegar el resto de las responsabilidades al sistema subyacente.

En el caso en que no exista una aplicación OO de la cual obtener la base del modelo de objetos, habría que plantear si tiene sentido desarrollarla, previo a su extensión con hipermedia. Esto dependerá de lo complejo de la aplicación, de su grado de comportamiento más allá del navegacional, y de la necesidad de contar con la aplicación fuera del espacio o ambiente de hipermedia. El framework es flexible hasta el punto extremo de permitir que sólo se definan nodos-hipermedia (HyperNodes) sin necesidad de un nivel de objetos subyacente, aunque esta hipermedia no podrá tener comportamiento alguno más allá de la navegación, y la mantenibilidad, reuso y adaptabilidad quedarán casi anuladas.

Suponiendo la existencia de la aplicación OO a ser extendida, veamos qué aspectos resaltar para ser observados posteriormente desde el nivel de hipermedia.

Ante todo el diseñador deberá decidir *a qué objetos quiere darle una interface hipermedial*. Esto significa observar cuáles son los objetos que encapsulan los datos que se quieren mostrar, y/o el comportamiento que se quiere disparar desde la interface de los nodos. En realidad el diseñador contará con un modelo de clases, y lo que decidirá es el conjunto de clases que serán mapeadas a clases de nodos, para que en definitiva sus instancias puedan ser observadas en nodos de un espacio hipermedial.

Así como no todo objeto será mapeado a un nodo, los nodos no necesitan observar únicamente un objeto, sino que pueden observar varios objetos fuertemente relacionados. Esto significa que la relación nodos → objetos puede ser múltiple, y no está definida sobre todo el conjunto de los objetos de la aplicación. Por otro lado, un objeto podría ser mapeado a más de un nodo, cuando se observa en relación con otros. Resumiendo, la cardinalidad de la relación objetos ↔ nodos es n:n, como lo muestra la Figura 26. Dicho en términos matemáticos, la relación es no inyectiva.

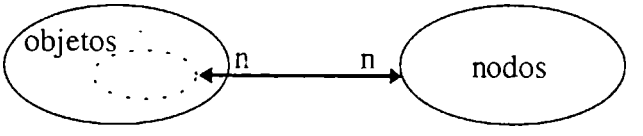


Figura 26: Cardinalidad de la relación objetos ↔ nodos.

Para poder decidir si dos objetos relacionados deben ser mapeados a un único nodo, o por el contrario, a dos nodos distintos unidos por un link, habrá que tener en cuenta si tiene sentido acceder a ambos objetos separadamente, o la información quedaría incompleta si no se muestra el conocimiento de ambos al mismo tiempo.

En cuanto a las relaciones entre objetos, existen varios tipos de relaciones a ser tenidas en cuenta, que estudiaremos en la próxima sección.

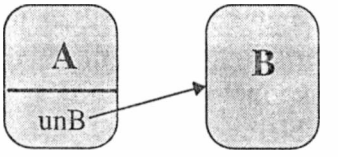
3.3. Descubrimiento de las relaciones entre componentes de la aplicación

El estudio de las relaciones existentes entre componentes del modelo de la aplicación es altamente importante para definir los links en forma significativa (es decir, que su existencia estará correctamente fundamentada por una relación en el modelo de objetos).

Algunas relaciones son explícitas y por lo tanto fáciles de encontrar, mientras que para otras costará más trabajo deducirlas. También pueden existir aquellas que no son directamente deducibles del modelo (porque no habían sido previstas, o porque sólo tienen sentido si se va a hablar de navegación), pero que resultarían muy útiles para el futuro usuario/lector de la hipermedia.

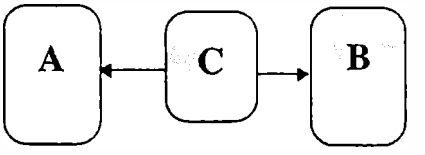
Entre las relaciones deducibles del modelo de objetos encontramos:

- *Relaciones explícitas de conocimiento*: un objeto A conoce a otro B, a través de una variable de instancia. Esta relación puede reconocerse directamente observando el diagrama de clases o de objetos, generalmente a través de flechas que conectan los objetos involucrados en la relación. En un diseño OO no todas las relaciones posibles están definidas por las referencias directas entre objetos, sino más bien por la clausura transitiva de las relaciones (como sucede con objetos contenedores intermedios). Para descubrir estas relaciones, la sección de "colaboradores" de las tarjetas CRCs puede ser un buen punto de inicio.



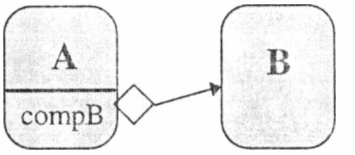
  Este tipo de relación será observada como un link si tiene sentido navegar a través de ella.

- *Relaciones modeladas mediante otros objetos:* en este caso estamos planteando la existencia de objetos asociativos, es decir aquellos que modelan una relación entre otros. En la jerga de patterns, estos objetos son llamados "adaptadores" o "mediadores". Esta relación puede descubrirse prestando atención a la parte del diagrama que luzca como en la figura, o revisando el propósito de una clase. A partir de esta relación conviene generar un link-objeto, u ObjectLink que observe el objeto C y obtenga de él la información que requiera.



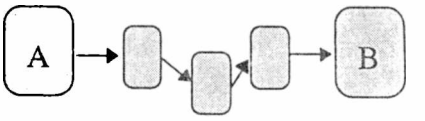
- *Relaciones inversas:* las relaciones inversas se obtienen al dar vuelta una relación existente entre dos objetos (como en el primer ejemplo sería la relación de B a A). Muchas veces las relaciones inversas también estarán presentes en el modelo, y por lo tanto serán descubiertas en la búsqueda del primer tipo de relaciones nombradas. En otros casos, la existencia de estas relaciones no tienen sentido o resultan redundantes en el modelo de objetos, pero sí hay que tenerlas en cuenta a la hora de ser navegadas. Estas relaciones se implementan como links cuyo predicado de asociación incluye al objeto destino.

- *Relaciones explícitas de agregación:* se debe prestar mucha atención a este tipo de relaciones para no confundirlas con aquellas de conocimiento. Se implementan de la misma forma, por medio de variables de instancia, pero la semántica es muy distinta: en este caso estamos hablando de relaciones de parte-de, como en el dibujo B es parte de A. Muchas notaciones de diseño utilizan formas distintas para distinguir una relación de conocimiento de una de agregación, como por ejemplo OMT [Rumbaugh+91], que utiliza en el segundo caso un rombo en el objeto contenedor como origen de la flecha de relación. Las relaciones de agregación **no** deberían ser mapeadas a links, para no confundir al usuario con simples relaciones de conocimiento, sino que el objeto contenedor debería traducirse a un nodo-compuesto, donde los agregados son las partes, y no hay links intermediarios para representar tal relación. Un más extenso razonamiento sobre este tema puede ser encontrado en el capítulo anterior.



Relaciones no explícitas del modelo pueden aparecer al observar el resultado del cómputo que realiza un objeto o el diseño en su tarjeta CRC, relacionando así *el productor con su producto*, aunque el productor no mantenga una referencia a su producto. Este tipo de relación se mapea a links cuyo destino es una fábrica de nodos y objetos asociados (instancia de AbstractStrategyFactory).

Otra relación no explícita, o indirecta, sería la que se obtiene luego de seguir un camino de varias relaciones. Por ejemplo, dado un camino de conocimiento como el de la figura, podríamos decidir que un link entre A y B puede ser muy útil de ser definido.



Relaciones no deducibles en lo absoluto del modelo de objetos sólo aparecerán por una falla en el mismo, tal vez ante un cambio de requerimientos. En este caso, si la aplicación puede ser cambiada, convendría agregar esta relación al modelo de objetos. Si la aplicación es cerrada en el sentido que no puede ser cambiada, entonces de todos modos pueden agregarse estos links.

Michael Bieber en su artículo [Bieber+97] presenta varios tipos de relaciones que pueden observarse, entre las que encontramos:

- *Relaciones de ocurrencia:* proveen acceso a todos los usos y vistas de una pieza de información. Ted Nelson llama a este tipo de relación "transclusion" [Nelson96]. Desde el punto de vista del framework de hipermedia, esta relación se traduciría a definir links entre todos los nodos que muestran el mismo objeto o partes de él.
- *Relaciones de proceso:* proveen acceso entre las distintas etapas o actividades de un proceso. Representan las interconexiones o flujo de datos entre grupos de trabajo, departamentos o aplicaciones de software manejando distintos aspectos de un producto o servicio. También incluyen la conexión de un objeto con el resultado de operar sobre él.
- *Relaciones estadísticas:* proveen acceso a elementos que tienen lugar bajo circunstancias similares, o elementos en los cuales uno no ocurre a menos que el otro ocurra.

- *Relaciones estructurales:* proveen acceso a objetos relacionados, basándose en la estructura interna de la aplicación. Estas serían para nosotros las relaciones explícitas del modelo.

3.4. Pautas para la creación de los componentes hipermediales

En esta sección presentaremos algunas pautas para la definición de los distintos componentes necesarios para crear una vista hipermedial. Cada subsección habla de un concepto particular, o si se quiere, una actividad dentro del proceso de instanciación del framework. El orden de las subsecciones puede seguirse como una secuencia de pasos, aunque no en todas las oportunidades este orden debe seguirse estrictamente.

Previamente a los pasos que se describen abajo, el responsable del proceso de instanciación deberá crear un objeto Hypermedia para una visualización particular de la aplicación. Esto lo hace mediante el Browser de Hipermedias, y a priori sólo necesita identificar el nombre de la misma.

3.4.1. Contextos

Luego de crear una nueva hipermedia, se debe *definir un contexto o punto de vista para cada perfil de usuario distinto, y las transiciones permitidas entre los distintos contextos.*

Este concepto fue descripto en la Sección 2.1.1.2., y su diseño en la 2.2.2. Para identificar los distintos perfiles de usuario de la hipermedia conviene agrupar, del conjunto general de usuarios, aquellos que realizarán tareas idénticas o similares sobre la misma. Recomendamos una categorización basada en los distintos usos del sistema más que en la información que cada usuario necesitará observar o le será permitido observar. Lo que a cada usuario le será significativo o válido observar surgirá de aquella categorización por tareas.

Nuestra definición de "perfiles de usuario" puede ser directamente relacionada con aquella de "actores" que la metodología OOSE [Jacobson+92] propone para la identificación de los distintos "casos de uso" de un sistema, con los que analiza el comportamiento esperado del mismo. En OOSE, un "actor" representa un rol específico que un usuario puede interpretar en el uso de un sistema. A cada "actor" de la hipermedia nosotros proponemos asociarle un *contexto*. Dentro de ese contexto aparecerán los "casos de uso".

Cada contexto que se crea debe ser identificado por un nombre, que bien puede ser el mismo del "actor".

Las *transiciones entre contextos* se crean para permitir que un mismo usuario pueda jugar *varios roles*. Cuando un usuario inicia una sesión con la hipermedia deberá hacerlo en un contexto particular y una vez en él, sólo podrá cambiar a alguna de las "vistas de nodo" asociadas con los contextos destino de las transiciones permitidas.

3.4.2. *Clases de nodos*

La definición de clases de nodos se realizará únicamente sobre la base del modelo de la aplicación subyacente. Una observación minuciosa del mismo, como se planteó en 3.2., permitirá decidir qué clases tendrán una interface hipermedial y con qué atomicidad lo harán.

Para las clases cuyas instancias se desean observar aisladamente en un nodo, se definirá una instancia de AtomicNodeClass asociada con cada clase. El nombre que identificará a la clase de nodos no necesita ser el mismo de la clase que observa. Otra decisión a tomar es la que surge de evaluar si se desea tener un nodo para todas y cada una de las instancias de la clase observada, o se quieren filtrar algunas instancias. De aquí que el tercer dato necesario en la creación de una nodeClass es el bloque de código que selecciona los objetos a observar (ver Figura 29). Ese bloque tiene un sólo parámetro que se corresponderá con cada objeto de la clase de la aplicación.

Para aquella clase (que llamaremos principal) cuyas instancias se desean observar conjuntamente con otras (instancias de una o más clases que llamaremos secundarias) que poseen información altamente relacionada, también se crea una instancia de AtomicNodeClass asociada con la clase principal y luego se agrega la asociación con las clases secundarias. La decisión de crear un mismo atomicNodeClass, o más de uno unidos por medio de links, dependerá de cuán fuerte sea la relación para que tenga o no sentido observar a aquellos objetos en forma separada en la vista hipermedial. También se ingresará un nombre identificatorio de la nodeClass y el bloque de código que selecciona los objetos a mapear a nodos.

Cuando luego se definan los aspectos que se observarán en una vista de nodo, aparecerán como disponibles aquellos que pertenecen a la clase principal y todas las secundarias.

En el caso de una clase que en el diseño de la aplicación aparece como una composición de otras clases, se creará una instancia de CompositeNodeClass asociada con la clase compuesta, a la que luego se agregarán las componentes como partes. Las interfaces de las vistas de los nodos compuestos se arman separadamente de aquellas de las partes, ya que se acceden separadamente, desde la compuesta.

3.4.3. *Hiper-nodos*

Los hiper-nodos surgirán de la necesidad de agregar información concerniente a la vista hipermedial, para presentar temas, o información multimedial, o proveer otros accesos.

Es conveniente destacar que aunque estos nodos podrían ser usados para agregar mucha de la información no disponible desde la aplicación, siempre es más recomendable que la información propia de la aplicación se agregue a la misma, es decir en el nivel de objetos. Sólo en los tres casos nombrados en el párrafo anterior, o cuando la aplicación no pueda ser modificada por cualquier motivo, los datos se agregarán en el nivel de hipermedia. Por supuesto estos datos no estarán disponibles cuando la aplicación sea accedida de cualquier otra forma fuera de la hipermedia en cuestión. Si otra hipermedia se definiera para la misma aplicación, aquellos datos deberían agregarse nuevamente si fuera necesario. Esta última consideración puede ayudar en la decisión de dónde conviene agregar un dato.

3.4.4. *Navegadores*

Por cada interface gráfica provista por la aplicación para acceder a su funcionalidad, o dicho de otro modo para disparar sus computaciones, habrá que decidir si conviene o tiene sentido mantenerla en la vista hipermedial. Esta decisión dependerá, entre otras cosas, de qué tan útil resultaba aquella interface para los usuarios que hasta el momento operaban sobre la aplicación, de cuánta funcionalidad hipermedial se quiera para la misma (ya que por ejemplo se reduce la posibilidad de crear anclas de links a sólo aquellas inmersas en un contenido, pues no se permiten agregar objetos de interface como sería un botón adicional), de la necesidad de mantener "dos modos distintos de operación" (ya que para poder navegar desde un Navigator hay que explícitamente conmutar al modo navegación).

3.4.5. *Colecciones*

En una aplicación hipermedia es importante proveer una organización del espacio de navegación en distintos grupos temáticos. Esto disminuye considerablemente el problema de "information overhead". Con esta finalidad se utilizan los nodos-colección.

Como ya se ha explicado, existen distintas estrategias de creación de nodos-colección, y que brindan flexibilidad en este proceso. La estrategia a utilizar dependerá de la disponibilidad de uno o más nodos-colección de partida (para aplicar una "set-based strategy", o "link-based strategy" sobre ellos); o de una clase de nodos sobre la que se pueda hacer una selección ("selection strategy"), y en otro caso se deberán elegir los componentes manualmente ("pick-up"). Cualquiera de las estrategias computadas se prefiere a la manual, pues las primeras soportan el dinamismo en la actualización de la hipermedia. Para una completa descripción de las distintas estrategias referirse a la Sección 2.2.1.3.

Como siguiente paso a la creación de un nodo-colección se deberá crear al menos una representación para el mismo, donde se agreguen los datos que serán mantenidos en el nodo. La creación de una interface gráfica para un nodo-colección es similar a la de un hiper-nodo, ya que la paleta de widgets disponibles es la misma. Una vez instalada aquella interface en una representación, se debe especificar la estructura de acceso que se pretende utilizar para acceder a los componentes del nodo, y automáticamente se agregan a aquella interface los widgets necesarios para soportar este acceso (una lista en el caso de índices y botones de 'next' y 'previous' en el caso de secuencias). Una posterior edición de la interface sería conveniente para ubicar los widgets a gusto del diseñador.

3.4.6. *Clases de links*

Las clases de links surgirán generalmente de relaciones existentes en el modelo, entre clases de la aplicación. Sólo habrá que observar entonces las relaciones de conocimiento entre clases, en la forma que explicamos en la Sección 3.3. También podrán aparecer clases de links desde clases de nodos a un nodo no-clasificado.

Para la creación de clases de links se debe especificar primeramente la clase de nodo origen. Cabe recordar que siempre tendremos una clase de nodo como origen para una clase de link, de manera que se cree automáticamente un potencial link clasificado

por cada nodo objeto. La Figura 31 muestra la herramienta que se utiliza para crear links.

El destino de una clase de link puede ser otra clase de nodos o uno o varios nodos no clasificados. Para uno u otro caso existe la posibilidad de que el destino sea único (single) o múltiple.

En el caso de que el destino sea otra clase de nodos, se deberá especificar el predicado de relación que será calculado dinámicamente, en forma de símbolo o de bloque de Smalltalk. Si el predicado es un símbolo, será considerado como un aspecto o mensaje del objeto en el nodo origen que retorna el objeto destino (en el momento de ser calculado se busca luego el nodo que representa a ese objeto destino para navegar hacia él). Ese mensaje podrá llevar como parámetro el ancla seleccionada para activar el link. Si el predicado es un bloque, este podrá tener uno, dos o tres parámetros. En el caso de uno o dos parámetros, el bloque deberá retornar el o los objetos destino, según sea un link singular o múltiple. El primer parámetro con el que se evalúa el bloque será siempre el objeto del nodo origen, y si existiera segundo parámetro, este será el ancla seleccionada.

En el caso de que la relación en el modelo se de al revés (caso de relaciones inversas explicado en 3.3.) necesitaríamos que el bloque se evaluara sobre los objetos del destino, de los que obtendremos el origen de la relación. Para eso se utilizan bloques con tres parámetros: el tercer parámetro tomará los valores de los objetos que pertenecen a la clase asociada al destino del link. Estos bloques toman la forma de predicados que retornan un valor booleano que indica si el actual parámetro del origen se relaciona con el actual parámetro del destino.

En el caso en que el destino sea un nodo particular, este quedará fijo por lo que no debe especificarse el bloque de relación.

En el caso de un destino múltiple, al momento de activación del link se le presentará al usuario un índice con todos los posibles destinos para que éste seleccione uno en particular. Se puede especificar mediante un selector qué aspecto del nodo se mostrará en ese índice. Por defecto muestra los nombres de los nodos.

3.4.7. Links no-clasificados

Los links clasificados serán creados automáticamente por sus clases de link, pero los links no-clasificados son los que serán creados "a mano" e individualmente. La herramienta de creación de estos links es la misma que en el caso de las clases de link, sólo que al abrirse sobre un nodo de hipermedia o sobre un navegador creará un link no-clasificado.

De la misma manera que se explicó en la subsección anterior, se deberá especificar si el destino del link será una clase de nodos o un nodo no-clasificado en particular. También se definirá si es simple o múltiple y su significado. El predicado de relación sólo se especifica cuando el destino del link es una clase de nodos. En este último caso el predicado será un bloque con un parámetro que representará el objeto asociado con cada nodo de la clase destino, o dos parámetros (el objeto destino y el ancla seleccionada).

Los widgets que pueden soportan ancla de links no-clasificados con destino definido a una *clase de nodo*, son sólo las listas de elementos (en el caso de Navigators)

y los botones. Una lista de elementos sobre la interface de un nodo navegador podrá entonces verse como un índice hacia nodos-objeto.

3.4.8. Vistas de nodo

Las vistas de nodo, como dijimos anteriormente, son las "caras" que el nodo muestra ante cada tipo de usuario, representado por un contexto. Una vista de nodo se construye primeramente identificando el o los contextos en los que se utilizará. No es necesario crear una vista por cada contexto definido, en el caso en que un nodo muestre la misma información para más de una vista.

Una vista de nodo se compone luego de un conjunto de "slots" de valores y un conjunto de slots de acciones. Cada slot se asocia a un aspecto del objeto, que luego es usado como mensaje al mismo.

Cuando se está definiendo una clase de nodos, la vista que se cree será una instancia de NodeViewClass, es decir una clase-de-vista-de-nodos. Luego ésta creará una NodeView con los mismos aspectos en cada nodo de la clase de nodos.

Para crear una vista de nodo basta seleccionar de la lista de contextos del browser de hipermedia (parte 4 en la Figura 27), los contextos en los que la misma será usada.

3.4.9. Representaciones

Varias representaciones pueden ser definidas dentro de una vista de nodo, para el caso de nodos de múltiples vistas (atomicNodes, compositeNodes y hyperNodes), o para un collectionNode. Al menos es necesario crear una representación por cada vista de nodo, pues es la que aporta el conocimiento de cómo mostrar la información del nodo. Esta instancia será además el *modelo* para la interface.

Al crear una representación se le asigna a la misma un nombre que la identificará dentro de la vista. Luego de esto se abre automáticamente el canvas para construir la interface con la que se mostrará la representación en cuestión.

3.4.10. Interfaces

El armado de interfaces no difiere mucho de aquel de interfaces normales en VisualWorks. Se utiliza el "canvas" usual (ver Figura 25), aunque la paleta de posibles elementos de la ventana a sido extendida con elementos adaptados a hipermedia. El conjunto de botones de la paleta (y por lo tanto los specs que se crearán con cada uno) también depende del tipo de nodo en cuestión. En el caso de interfaces para nodos Navigator no aparece siquiera la paleta, ya que no es posible modificar aquella interface agregándole nuevos elementos.

Cuando se define la interface para cualquier tipo de nodo excepto el Navigator, y al agregar un widget a la misma, se deben ingresar sus propiedades, a las que se accede por medio del menú invocado con el botón derecho del mouse. En las propiedades de un widget deben especificarse: aspecto que observa, clase del modelo del cual obtiene el valor, nombre del aspecto que retorna el menú de acciones, y en el caso de widgets adaptados, también se definen desde allí las anclas de links.

3.4.11. Anclas de link

Las anclas de links pueden definirse en widgets separados, específicamente botones adaptados para funcionar como zona sensible de un ancla de link, o pueden definirse inmersos dentro del contenido de otro widget. Por el momento el framework soporta un texto, un input field o una lista como posibles contenedores de anclas. La decisión por la primera o la segunda opción en la definición de anclas dependerá de distintos factores:

- *disponibilidad de un widget que en su contenido referencie al link:* en este caso conviene definir el ancla dentro del contenido, ya que el mismo provee un contexto en el cual el link cobra mayor significado;

- *definición de áreas de agrupación de links:* muchas veces conviene agrupar los botones para navegación en una misma área, y los botones para el disparo de acciones de la aplicación en otra; esta solución reduce la confusión que puede provocar la mezcla de funcionalidades (del nivel de objetos y del nivel de hipermedia) en un nodo.

Las anclas separadas y las que se definan sobre un widget de lista deben agregarse en el momento de crear la interface, es decir que lo debe hacer el autor o diseñador de la hipermedia. Por el contrario, las anclas inmersas en un texto pueden agregarse en ese momento o posteriormente en el momento de navegación (cuando en realidad se ve el texto del widget). Esto significa que tanto el autor como el lector de la hipermedia podrán crear hotwords. Las anclas para anotaciones sólo se crean durante la navegación.

3.5. Utilización del Browser de Hipermedias para la instanciación del framework

En esta sección se describirá la funcionalidad del Browser de Hipermedia presentado en el capítulo anterior.

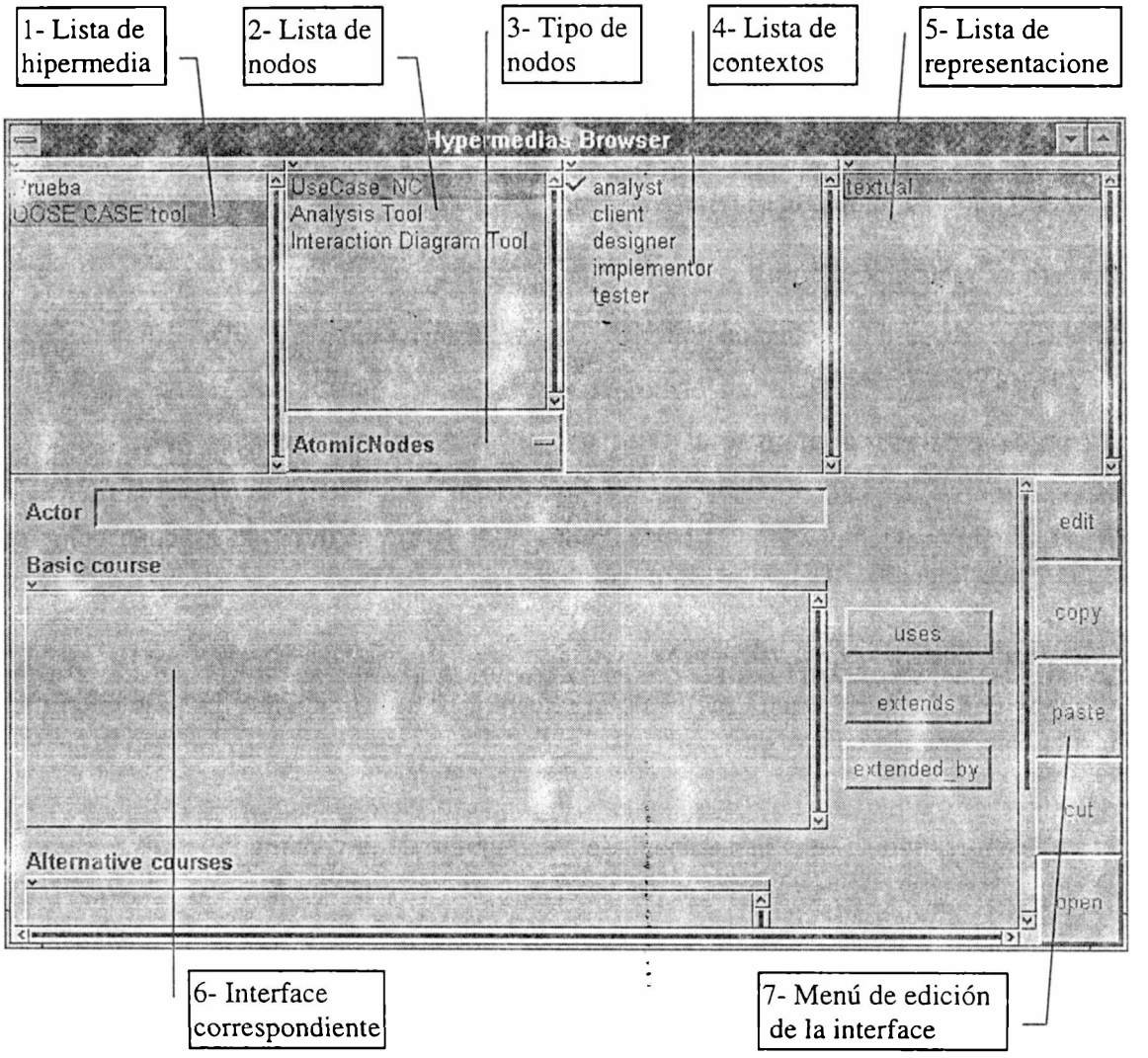


Figura 27: Partes del Browser de Hipermedia

- Lista de hipermedias

En esta lista se muestran las aplicaciones hipermedias definidas en la imagen de VisualWorks abierta.

2- Lista de nodos

Esta lista muestra los nodos que pertenecen a la hipermedia seleccionada en (1), del tipo seleccionado en (3). Si el tipo es 'Atomic nodes' o 'Composite nodes', lo que se muestra en este listado son en realidad las clases de nodo atómico o compuesto, respectivamente.

3- Tipo de nodos

Esto es un combo-box que determina los tipos de nodos a visualizar en (2) para la hipermedia seleccionada.

4- Lista de contextos

En este listado se presentan los contextos definidos para la hipermedia seleccionada en (1). Aquí se permiten múltiples selecciones para que al definir una vista para el nodo del listado (2), la misma pueda asociarse con más de un contexto.

5- Lista de representaciones

Muestra las representaciones disponibles en la vista seleccionada en (4), para el nodo en (2) y la hipermedia de (1).

6- Interface correspondiente

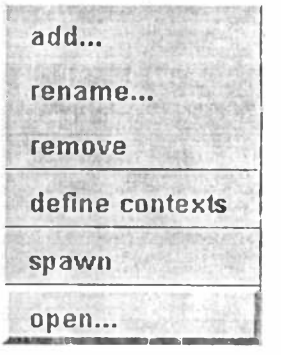
Esta será la interface asociada con la representación en (5). Mientras no haya una representación seleccionada, esta parte de la ventana va mostrando la especificación de lo que haya seleccionado hasta el momento.

7- Menú de edición de la interface

Menú de edición normal para partes de la interface. Por el momento no se pueden usar los botones 'copy', 'cut' y 'paste'.

Veamos ahora en detalle el menú asociado a cada parte de la ventana.

1- Menú de hipermedias



add...

Agrega una hipermedia. Se pide ingresar el nombre de la misma.

rename...

Cambia el nombre a la hipermedia seleccionada.

remove

Borra la hipermedia seleccionada y todo su contenido.

define.contexts

Permite definir contextos para la hipermedia y transiciones entre los mismos con la ventana que muestra la Figura 28.

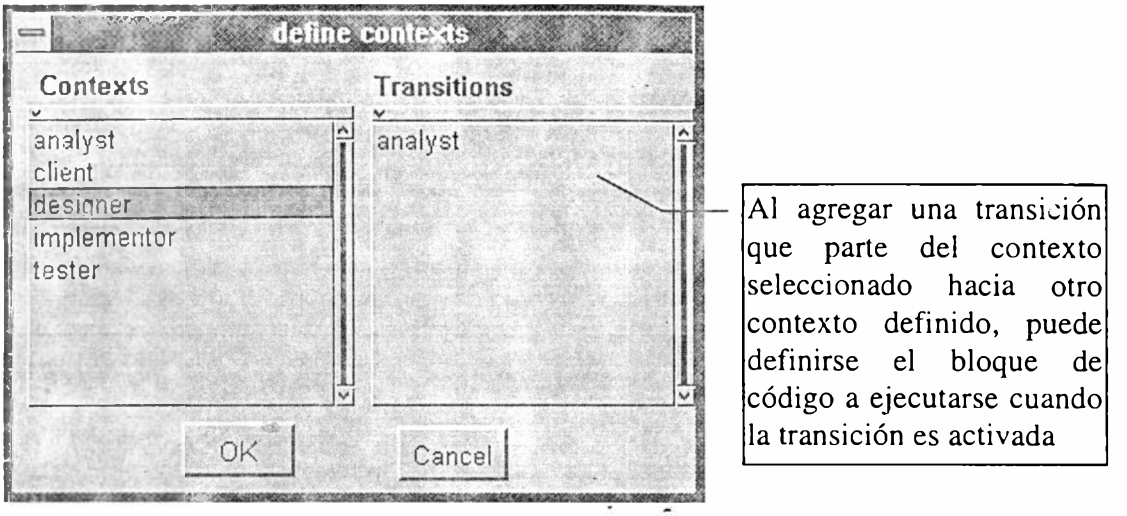


Figura 28: Ventana para la definición de contextos

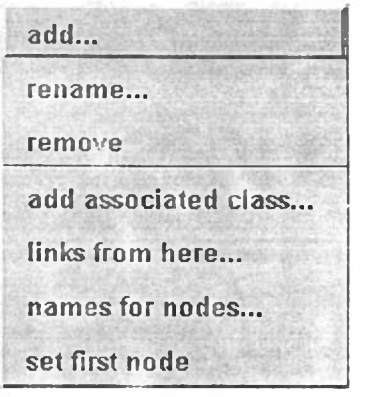
spawn

Abre un nuevo browser de hipermedias, mostrando únicamente los datos de la hipermedia seleccionada.

open

Abre la hipermedia seleccionada en el primer nodo. Requiere que se ingrese el contexto en el que se abrirá la hipermedia.

2- Menú de nodos



add...

En el caso que el tipo de nodos seleccionado en (3) sea 'atomic node' o 'composite node', agrega una nueva clase de nodos a la hipermedia, con la ventana que muestra la Figura 29. De otra manera agrega un nodo del tipo que indica el ítem (3).

rename...

Cambia el nombre del nodo ó la clase de nodos seleccionada.

remove

Borra el nodo o la clase de nodos seleccionada, los links desde y hacia ella, y todo su contenido.

add associated class...

Este ítem sólo aparece habilitado cuando el tipo de nodos en (3) es 'Atomic Nodes'. Abre otra ventana que permite agregar una nueva clase asociada con la clase de nodo seleccionada. En dicha ventana además del nombre de la clase a asociar se debe ingresar el predicado de asociación o join.

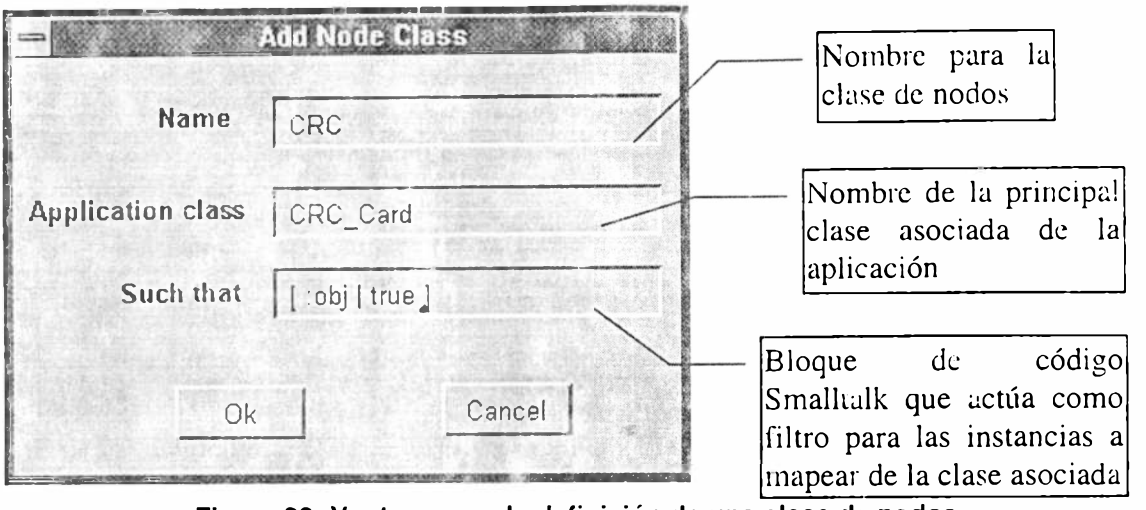


Figura 29: Ventana para la definición de una clase de nodos

add sub-nodeClass...

Este ítem sólo aparece cuando el tipo de nodos en (3) es 'Composite Nodes'. Abre otra ventana que permite agregar una clase de nodo existente como parte de la clase de nodo seleccionada en (2). También se debe ingresar el predicado de relación entre los objetos de la aplicación representados por las nodos en cuestión.

links from here...

Abre la ventana que muestra la Figura 30, en la que pueden observarse los links cuyo origen es el nodo seleccionado. En el caso de que el tipo de nodos sea una clase de nodos, lo que se observa en aquella ventana son clases de links. Por ejemplo, la Figura 30 muestra las clases de links cuyo origen es la clase de nodo "UseCase_NC".

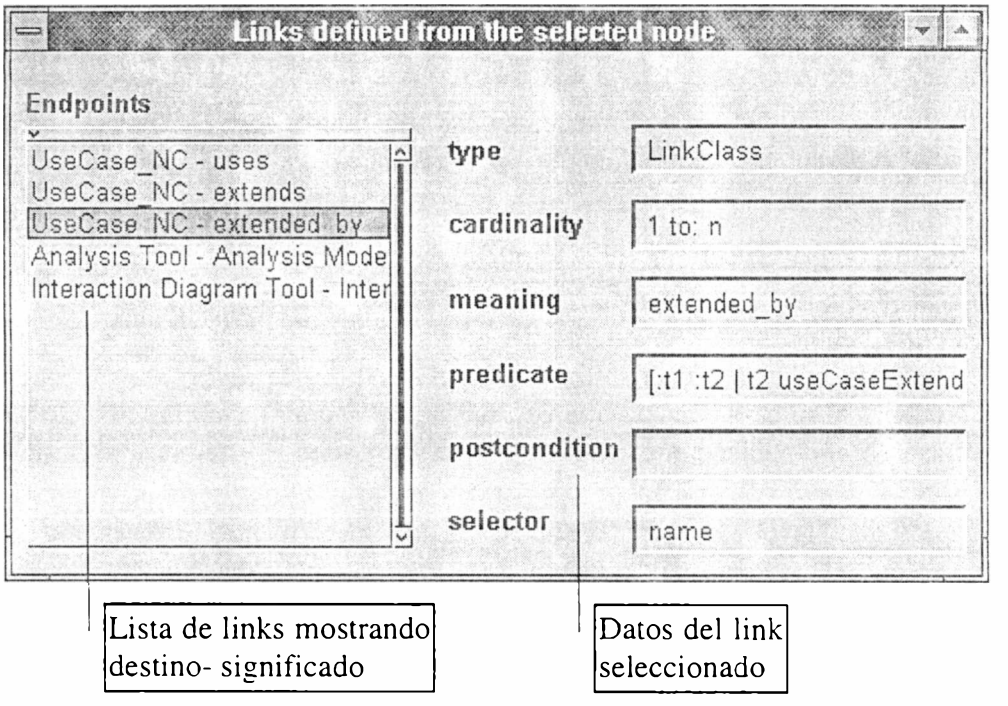


Figura 30: Ventana de descripción de links

La lista de links a la izquierda tiene asociado un menú que permite agregar nuevos links o borrar el seleccionado. Al seleccionar la opción 'add', aparece la ventana que se muestra en la Figura 31 para crear nuevos links. En el ejemplo se está creando una clase de links desde "UseCase_NC" hacia la clase de nodos "Analysis Tool", cuya cardinalidad será unitaria, su significado "to analysis" y el bloque de código (no mostrado íntegramente) relaciona los objetos de la clase origen y destino, tal que el destino conoce al origen.

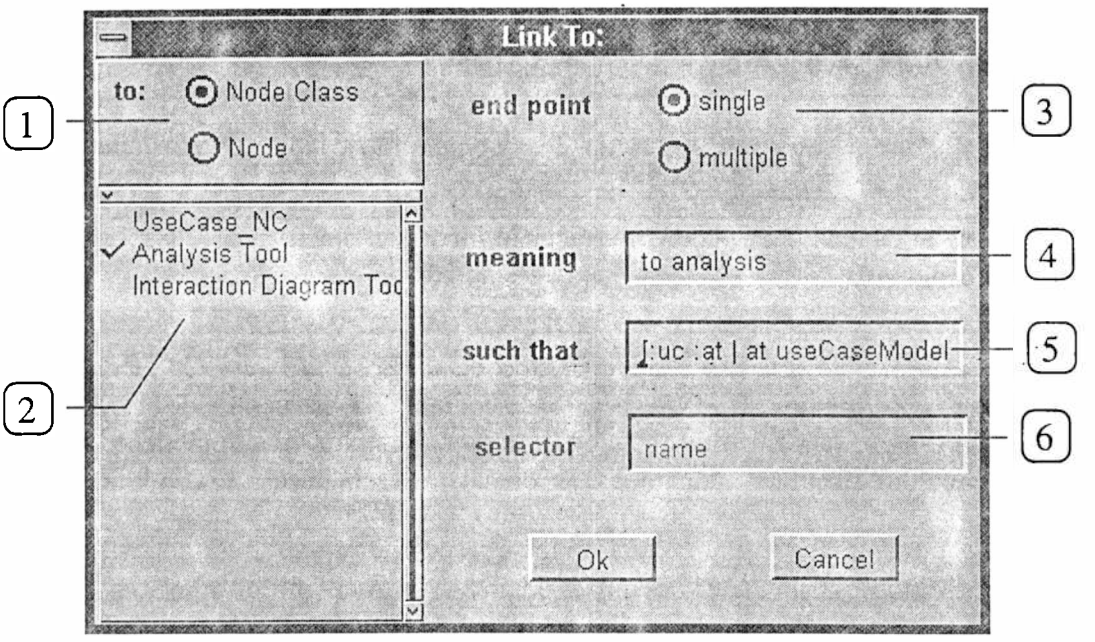


Figura 31: Ventana de creación de links

En los radio-buttons (1) se chequea si el destino será otra clase de nodos o uno o varios nodos no clasificados, los cuales se seleccionan luego de la lista (2), que muestra nodos o clases de nodos, según lo chequeado en (1).

En los radio-buttons (3) se chequea la cardinalidad del link, que podrá ser singular (single) o múltiple. El significado se especifica en (4).

En el caso de que el destino sea otra clase de nodos, el predicado de relación que será calculado dinámicamente deberá ser definido en (5) en forma de símbolo o de bloque de Smalltalk.

En el caso de un destino múltiple, el aspecto del nodo que se mostrará en el índice de posibles destinos al momento de activación del link se puede especificar en (6).

name for nodes...

Este ítem sólo aparece habilitado para las clases de nodos. Permite cambiar el nombre que mostrará cada nodo perteneciente a la clase seleccionada, obteniéndolo de un aspecto que se pide ingresar, el cual enviará al objeto asociado.

set first node

Setea el primer nodo de la hipermedia. Esto debe hacerse antes de abrirla, para que la misma sepa que nodo activar primero.

5- Menú de representaciones

add...

Agrega una representación. Se pide ingresar el nombre de la misma, y automáticamente se abre el canvas para construir la interface. La Figura 25 mostraba esta herramienta de construcción de interfaces cuya paleta ha sido extendida con widgets adaptados a hipermedia.

rename...

Cambia el nombre de la representación seleccionada.

remove

Borra la representación seleccionada, su contenido y la interface asociada.

edit

Abre el canvas para poder modificar la interface asociada con la representación seleccionada.

access structure...

Este ítem sólo aparece cuando el tipo de nodos en (3) es 'Collection nodes'. Abre una ventana de diálogo para seleccionar la estructura de acceso que se quiere agregar a la representación seleccionada, para los componentes del nodo colección. Esto automáticamente agrega los widgets necesarios en la interface.

No tiene sentido describir aquí el canvas para la construcción de interfaces, ya que su funcionalidad es bastante extensa y está muy bien cubierta por el Tutorial de VisualWorks. Veamos únicamente los cambios realizados sobre la misma.

♦ La paleta

La Figura 32 muestra las dos paletas definidas, una para clases de nodos y otra para hiper-nodos. En ella se señalan los widgets sobresalientes.

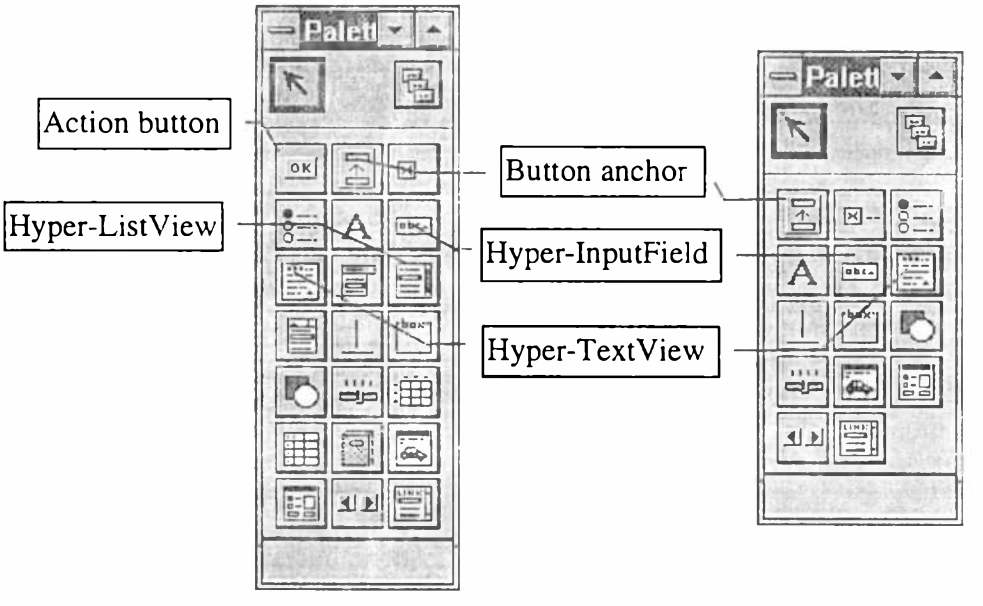


Figura 32: Paletas para la creación de widgets en la interface de una clase de nodos (a la izquierda) y en la interface de un hiper-nodo (a su derecha)

♦ Propiedades de los widgets

Cuando se agrega un widget por medio del botón de la paleta correspondiente, y luego de haber sido ubicado, el siguiente paso es la definición de sus propiedades. En las propiedades de los widgets adaptados a hipermedia aparecen: la clase del modelo que define el aspecto a observar desde el widget, y el botón para la definición de anclas. En la Figura 33 se muestra la ventana de propiedades que aparece para un TextEditorHyperView (widget textual con capacidad de soportar hotwords).

En (1) se especifica el nombre del mensaje que será enviado al objeto de la aplicación para obtener el valor del widget. La clase que define aquel mensaje será la que se selecciona en (4) (esta lista sólo contiene las clases asociadas al nodo en cuestión).

Si se quiere mantener un menú de acciones a activarse con el botón derecho sobre el widget, se debe ingresar en (2) el nombre del mensaje que al ser enviado al objeto de la aplicación que retorna el menú y las acciones asociadas.

El identificador de un widget es por defecto su aspecto. En (3) puede agregarse otro nombre para el widget que será utilizado por la vista de nodo.

Para definir anclas de links inmersas en el contenido del texto, basta con seleccionar el botón "Hotwords" (5). La ventana que aparece en ese caso es la que muestra la Figura 34. En la misma aparecerán a la derecha los distintos links de los cuales el origen es el nodo en cuestión. Al seleccionar un ítem de esa lista y elegir "add" del menú asociado, se pide ingresar la palabra que actuará de hotword en la activación.

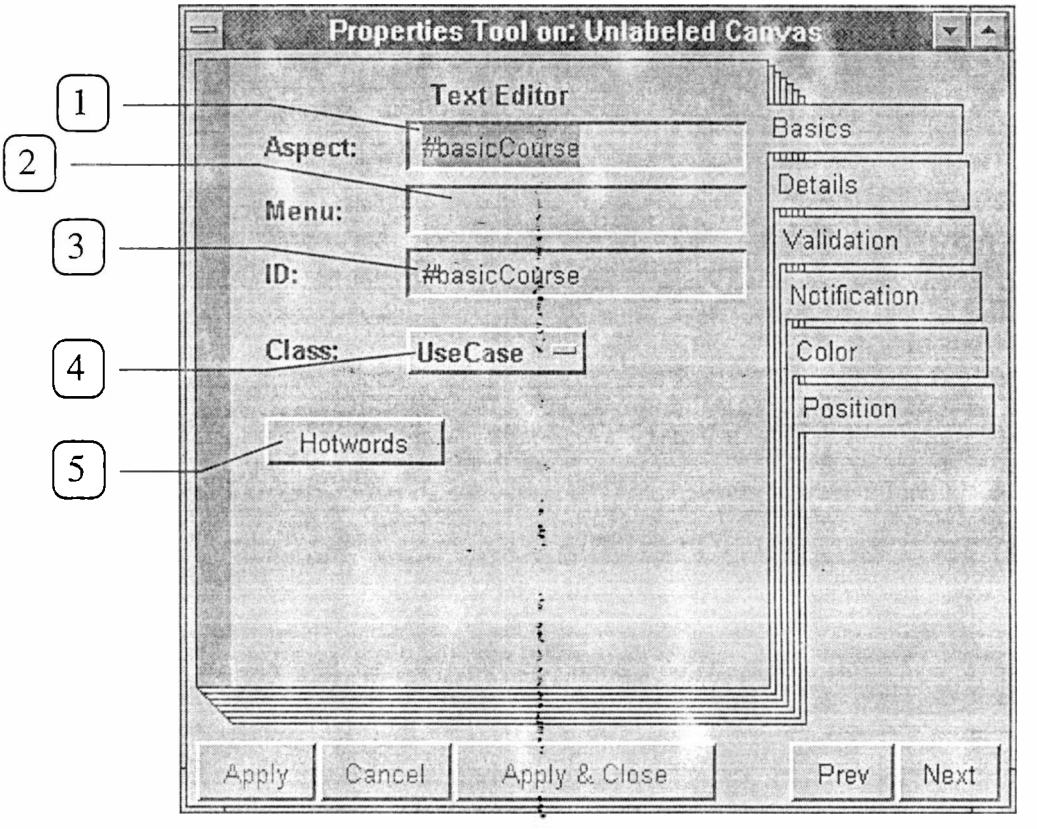


Figura 33: Propiedades de un TextEditorHyperView

En cuanto a la definición de hotwords en el momento de la navegación, primeramente cabe recordar que tendremos que conmutar al modo 'edición' pues el widget se abre siempre en modo 'navegación'. Luego se deben seleccionar la o las palabras que conformarán la hotword. En el menú del modo edición, además de las operaciones básicas, aparece 'create anchor for link' (crear ancla de link) y 'annotate' (anotar). Sólo queda entonces decidir por alguna de estas dos opciones. Si se elige la primera y nos encontrábamos editando la interface para una clase de nodos, se presenta una lista con las posibles clases de link a instanciar. Si en cambio se trata de un nodo no-clasificado, en la lista aparecen todos los posibles destinos. En el caso de elegir 'annotate' aparece una ventana de texto para agregar la anotación. Luego de definir la hotword habrá que conmutar nuevamente al modo 'navegación' para poder activarla.

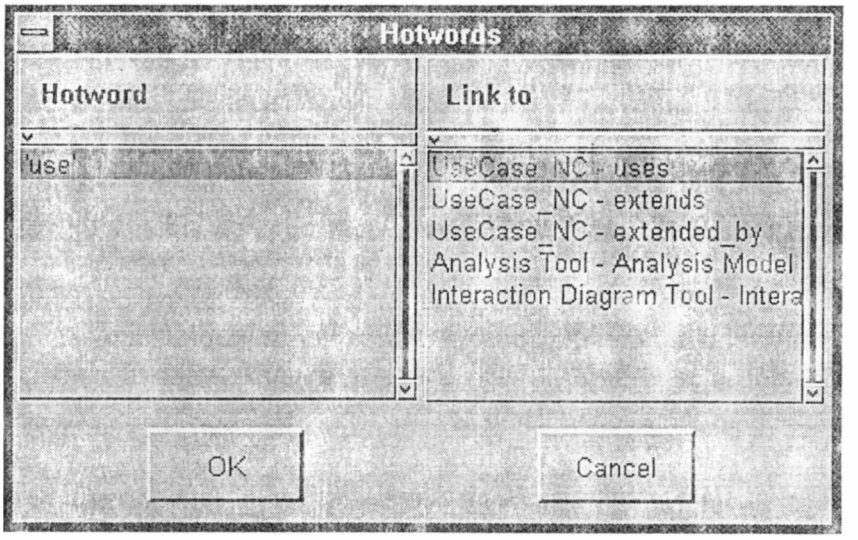


Figura 34: Ventana de definición de hotwords

4. Desarrollo de un ejemplo de utilización del framework

En este capítulo se desarrolla un prototipo para un ambiente CASE con el propósito de transferir de manera más completa los conceptos que se fueron exponiendo en el trabajo y ejemplificar las pautas expuestas en el capítulo anterior.

4.1. Modelo de la aplicación subyacente

Primeramente se presenta un modelo reducido de la aplicación que luego será extendida con funcionalidad de hipermedia, ejemplificando el proceso de instanciación del framework.

Supongamos contar con una herramienta CASE para la metodología OOSE [Jacobson+92] modelada en el paradigma de orientación a objetos. Revisemos las primeras etapas de esta metodología.

El Proceso de Análisis de la metodología OOSE incluye un Modelo de Requerimientos y un Modelo de Análisis.

> El Modelo de Requerimientos está compuesto de un Modelo de Casos de Uso, una especificación de interface y posiblemente un Modelo de Objetos del Dominio.
>
> > Un Modelo de Casos de Uso se construye a partir de *actores* y los *casos de uso* que estos disparan. Cada *caso de uso* se describe con un texto que explica su curso básico y algunos cursos alternativos si es necesario. Los casos de uso se relacionan entre sí mediante las relaciones "usa" y "extiende".
>
> El Modelo de Análisis requiere que se identifiquen objetos de *interface*, *entidades* y *de control*, en cada caso de uso. Esos objetos pueden ser descriptos en mayor medida especificando sus responsabilidades y colaboraciones.

El Proceso de Construcción involucra un Modelo de Diseño y un Modelo de Implementación.

> El Modelo de Diseño se compone de *bloques* que surgen a partir de los objetos descubiertos en el Modelo de Análisis. Además requiere la construcción de un Diagrama de Interacción por cada caso de uso y algunos otros diagramas.

Concentrémonos entonces en los Casos de Uso y sus relaciones con el Modelo de Casos de Uso y el Modelo de Análisis. La Figura 35 muestra un diagrama de clases que modelaría esta parte del diseño, usando la notación de [Rumbaugh+91].

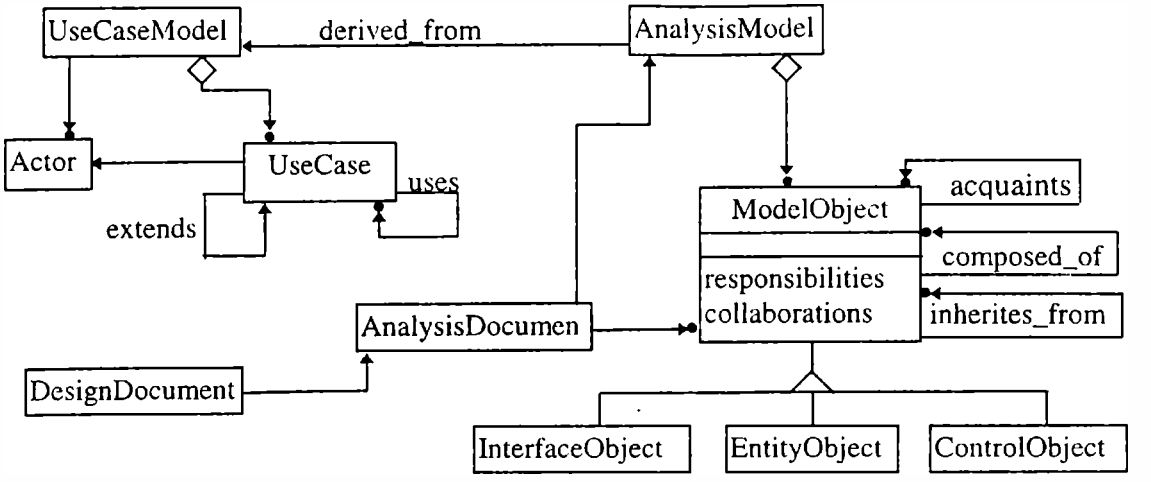


Figura 35: Diagrama de clases simplificado del Modelo de Casos de Uso y Modelo de Análisis de la metodología OOSE

4.2. Modelo de la hipermedia

¿Cómo podríamos realzar una aplicación de este tipo con FH? Por ejemplo, podríamos definir diferentes puntos de vista relativos a cada rol en el desarrollo de una aplicación; podríamos agregar navegación entre los modelos y entre los casos de uso que se relacionan, desde los sustantivos en la especificación de un Caso de Uso hasta la tarjeta CRC del modelo de objetos que le corresponde; se podría agregar una herramienta de ayuda con anclas a conceptos relacionados; permitir anotaciones en los modelos y diagramas, etc.

Veamos entonces el proceso de instanciación del framework como una serie de pasos para la extensión del CASE.

1) ***Creación de una instancia de la clase Hypermedia con un nombre que la identifique.***

Primeramente creamos una instancia de la clase Hypermedia para una visualización hipermedial particular de la aplicación. En este paso sólo necesitamos especificar el nombre de la instancia, como por ejemplo "OOSE CASE-tool". Esto lo podemos hacer utilizando el Browser de Hipermedia, mediante la opción "add..." del menú de hipermedias.

2) ***Definición de puntos de vistas por cada perfil de usuario y de transiciones permitidas entre puntos de vista.***

Podríamos pensar, como dijimos anteriormente, en la definición de un punto de vista por cada rol en el desarrollo de una aplicación. Así tendremos los roles: analista,

diseñador, implementador, y tester. Las transiciones entre roles deberían ser de cada rol al siguiente, pero no al anterior.

3) ***Definición de clases de nodo y nodos.***

Como dijimos en el capítulo anterior, es importante que hagamos una profunda observación del modelo de la aplicación para derivar de allí las clases de nodos que conformarán el modelo de la hipermedia.

De acuerdo con esto, podemos derivar las siguientes clases de nodo de nuestro modelo acotado del CASE:

- a simple vista surge que la herramienta debería tener un nodo por cada Caso de Uso que se defina; por lo tanto, partiremos por derivar de la clase UseCase una clase de nodo, que podríamos llamar UseCase-NC;

- luego observamos la clase Actor directamente relacionada con UseCase, y aparece que, además de una clase de nodo Actor-NC que muestre la información de cada actor, sería importante ver el nombre del actor en el nodo del caso de uso que el mismo genera; de aquí que asociemos la clase Actor a la clase de nodo UseCase-NC;

- el rombo en la flecha que sale de UseCaseModel hacia UseCase nos dice que la primera clase es una agregación de la segunda; la mejor forma de mapear una composición del nivel de objetos en el nivel de hipermedia es definiendo una clase de nodo compuesto, que llamaremos UseCaseModel-NC; la clase de nodos "parte" será entonces UseCase-NC;

- una jerarquía que aparece en el modelo es la de ModelObject; ésta es una superclase abstracta, y por lo tanto no tendremos instancias de ella en la aplicación ni nodos derivados de ella en la hipermedia; las instancias que sí deberíamos mapear a nodos que representen su tarjeta CRC son las de InterfaceObject (de donde surge InterfaceObjectCRC-NC), EntityObject (mapeada a EntityObjectCRC-NC), y ControlObject (a ControlObjectCRC-NC).

- de la misma forma que UseCaseModel-NC, se define AnalysisModel-NC observando la clase AnalysisModel como compuesta, y asociando como partes a las clases de nodo previamente definidas: InterfaceObjectCRC-NC, EntityObjectCRC-NC y ControlObjectCRC-NC.

- por último podemos considerar clases de nodo AnalysisDocument-NC y DesignDocument-NC, asociadas a las clases de la aplicación del mismo nombre.

Veamos ahora aquellos nodos que aparecerán en el nivel de hipermedia como complemento a los nodos clasificados; estamos hablando de los **hiper-nodos**. Seguramente querremos definir un nodo "carátula" o presentación del sistema. Otro uso posible aparece con la definición de nodos que provean "ayuda" para la utilización de las herramientas o la construcción de diagramas; para esto necesitamos asumir que esta información no tiene sentido mantenerla a nivel de la aplicación, por ejemplo porque al ser construida en forma netamente hipertextual no sería utilizada en otro contexto que no sea el de hipermedia.

Si seguimos un poco más allá de lo representado en el modelo de la Figura 35, y suponemos que la herramienta CASE utiliza el browser de clases provisto por VisualWorks™ para permitir la codificación de cada objeto del modelo de diseño, podemos pensar en definir un nodo **Navigator** sobre aquel browser de clases. Esto permitirá al programador, por ejemplo, definir links desde las clases que aparecen en el browser hacia sus tarjetas CRC, además de crear anotaciones en el código a gusto.

Ejemplos de **nodos-colección** serían: todos los ModelObjects que pertenecen a una misma jerarquía, o que interactúan en el mismo pattern, todos los que colaboran con un modelObject particular, la colección de todos los documentos de análisis y diseño de un UseCaseModel, etc.

Resumiendo las clases de nodo y nodos definidos, tenemos:

♦ Clases de nodos atómicos (AtomicNodeClasses):

UseCase-NC	a partir de UseCase como clase asociada principal y Actor como secundaria;
Actor-NC	a partir de Actor;
InterfaceObjectCRC-NC	a partir de InterfaceObject;
EntityObjectCRC-NC	a partir de EntityObject;
ControlObjectCRC-NC	a partir de ControlObject;
AnalysisDocument-NC	a partir de AnalysisDocument;
DesignDocument-NC	a partir de DesignDocument.

♦ Clases de nodos compuestos (CompositeNodeClasses):

UseCaseModel-NC	a partir de UseCaseModel como clase compuesta y asociando como parte a UseCase-NC;
AnalysisModel-NC	a partir de AnalysisModel como clase compuesta y asociando como partes a InterfaceObjectCRC-NC, EntityObjectCRC-NC y ControlObjectCRC-NC

♦ Hiper-nodos (HyperNodes):

Carátula.

Los necesarios para la ayuda.

♦ Nodos Navegadores (Navigators):

FullBrowserNavigator

ClassBrowserNavigator

MethodBrowserNavigator

♦ Nodos-Colección (CollectionNodes):

HierarchiesNode	join entre todos los collectionNodes definidos por la selección sobre nodos InterfaceObjectCRC, EntityObjectCRC y ControlObjectCRC que pertenecen a la misma jerarquía ;
PatternsNode	ídem para los nodos que representan objetos que interactúan en el mismo pattern ;
CollaborationsNode	ídem para los nodos que representan objetos que colaboran con un modelObject particular ;
DocumentsNode	join entre la colección de documentos de análisis que pertenecen a un UseCaseModel y la colección de documentos de diseño del mismo UseCaseModel.

4) ***Definición de clases de link y links.***

Observemos primeramente las relaciones del modelo de la aplicación que queremos extender a clases de links. Por supuesto tendremos en cuenta aquellas relaciones que conectan clases ya mapeadas a clases de nodos de la hipermedia.

De esta forma surgen:

1) "To the UseCase Model"
entre el hiper-nodo "Carátula" y la clase de nodos "UseCaseModel_NC";
destino múltiple a todos los nodos de la clase del destino;

2) "Uses-LC"
entre "UseCase-NC" y ella misma;
destino múltiple dado por la relación [:ucOrigen | ucOrigen usedUseCases]

3) "Extends-LC"
entre "UseCase-NC y ella misma;
destino singular dado por la relación [:ucOr | ucOr extendedUseCase]

4) "Extended-by-LC"
entre "UseCase-NC y ella misma;
destino múltiple dado por la relación [:ucOr :ucDest |
ucDest extendedUseCase == ucOr]

5) "Used-by-LC"
entre "UseCase-NC" y "Actor-NC";
destino múltiple dado por la relación [:ucOrigen | ucOrigen actors]

6) "Triggers-LC"
entre "Actor-NC" y "UseCase-NC";
destino múltiple dado por la relación [:actor :ucDest |
ucDest actors includes: actor]

7) "To Analysis-LC"
entre "UseCaseModel-NC" y "AnalysisModel-NC";
destino singular dado por la relación [:ucm :am | am useCaseModel == ucm]

8) "To UseCaseModel- LC"
entre "AnalysisModel-NC" y "UseCaseModel_NC";
destino singular dado por la relación [:am :ucm | am useCaseModel]

9) "To AnalysisDocument-LC"
entre "AnalysisModel-NC" y "AnalysisDocument-NC"
destino singular dado por [:am :adoc | am document]

10) "To DesignDocument-LC"
entre "AnalysisDocument-NC" y "DesignDocument-NC"
destino singular dado por [:adoc :ddoc | ddoc analysisDocument == adoc]

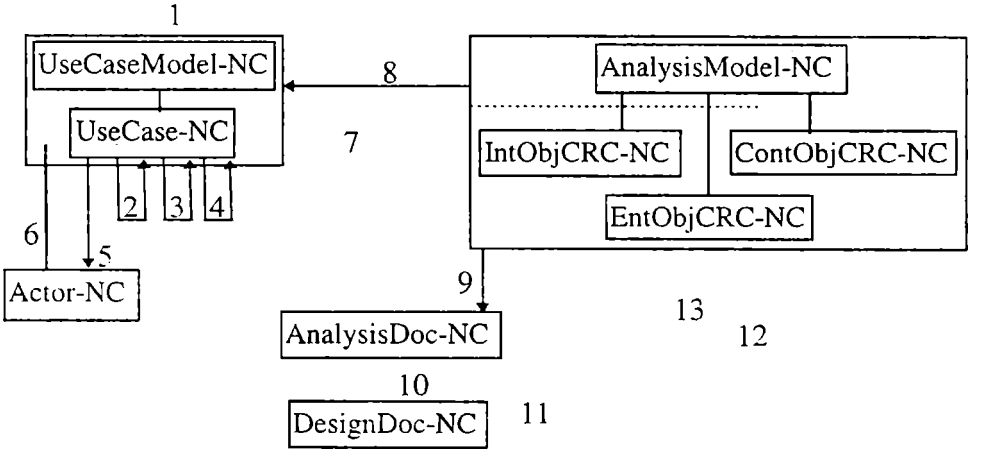
11) "To Implementation-LC"
entre "DesignDocument-NC" y "FullBrowserNavigator"
destino singular hasta el navigator.

12) "To Class Implementation-LC"
entre las clases de nodos que representan CRC's y "ClassBrowserNavigator"
destino singular hasta el navigator.

13) "To CRC"
entre el "FullBrowserNavigator" y las clases de nodos que representan CRC's;
destino singular dado por la relación [:nav :anch :crc |
crc name == anch]

etc.

La siguiente figura muestra las principales clases de nodos, nodos, clases de links y links, que conforman el modelo del nivel de hipermedia. La notación utilizada es la propuesta por OOHDM [Schwabe+95], donde las cajas representan nodos o clases de nodos, las flechas representan links, y se ha agregado la representación de un nodo compuesto, mediante una caja que agrupa el compuesto y sus partes, las que se ubican debajo de la línea punteada. Los números sobre las fechas identifican los links listados anteriormente.



5) ***Especificación de los datos y el comportamiento a ser expuesto en cada nodo bajo cada diferente punto de vista.***

Tomemos una clase de nodo como ejemplo : UseCase-NC.

El conocimiento de cada objeto UseCase que vamos a mostrar desde la vista de analista puede ser : actores, nombre, basicCourse. El conocimiento desde la vista de diseñador puede ser : nombre, basicCourse, alternativeCourses. El nodo no provee vista para el implementador.

Esto es sólo un mínimo ejemplo pues la especificación de este actividad para todo el modelo de la hipermedia es bastante extensa y dependerá del talento del diseñador.

6) ***Diseño de las distintas representaciones de los datos del nodo para cada punto de vista.***

Esta actividad se basa en el diseño de las interfaces gráficas para cada representación de los nodos, la disposición de los objetos de interface, la forma de activar el comportamiento en el caso de nodos-objeto (desde menúes o desde botones separados), la incorporación de las anclas (inmersas en el contenido de algún widget o separadas en botones de navegación). Dado que este paso sería muy extenso de describir aquí para cada interface definida para el CASE tool, se provee en el software que acompaña a este trabajo.

5. Resultados del trabajo

En este capítulo se presentan los resultados alcanzados con este trabajo, incluyendo las publicaciones científicas escritas durante el desarrollo del framework, como algunos ejemplos de su utilización.

5.1. Publicaciones

A continuación se listan cronológicamente las publicaciones científicas realizadas durante el desarrollo del framework, en las que fueron apareciendo resultados del mismo. Por cada publicación, además de la referencia, se describe brevemente su contenido. La mayor parte de ellas pueden ser accedidas desde la URL: http:\\www-lifia.info.unlp.edu.ar\~garrido.

- Un Framework Orientado a Objetos para Hypermedia
  Analía Amandi y Alejandra Garrido.
  Anales de CLEI'94: Conferencia Latinoamericana de Informática.
  México D.F., México. Setiembre de 1994.

  Este es el primer artículo publicado sobre el framework. Presenta el diseño preliminar del framework de hipermedia. Se explican los tres niveles de abstracción en el que se definen los componentes de la arquitectura, y la conexión entre los mismos.

- Extending Object-Oriented Applications with Hypermedia Functionality.
  Gustavo Rossi, Alejandra Garrido y Analía Amandi.
  Technical Report of The New Jersey Institute of Technology #95-10: First Workshop on Incorporating Hypermedia Functionality into Software Systems, realizado en conjunción con ECHT'94: European Conference on Hypermedia Technologies. Edimburgo, Escocia. Setiembre de 1994.

  Este es un artículo corto en el que presentamos nuestra experiencia previa en el uso de hipermedia en aplicaciones OO; discutimos el propósito u objetivo de construir un framework OO y explicamos sus características principales.

- Objects On Stage. Animating and Visualising Object-Oriented Architectures in a CASE Environment
  X.Alvarez, G.Dombiak, A.Garrido, M.Prieto y G.Rossi.
  Proceedings of the 6th. European Workshop on Next Generation of CASE Tools, realizado en conjunción con CAiSE'95. Jyväskylä, Finlandia. Junio de 1995.
  París: Université de Paris Press, 1995.

  Se propone una idea original de usar facilidades de visualización y animación en el contexto de un ambiente CASE OO. Primero analizamos el problema general: cuáles son los factores que hacen que una herramientas CASE sea o no exitosa. Luego describimos las características principales de un ambiente que remarca la utilización de animación y herramientas de visualización, y un framework OO para enriquecer los documentos del CASE y los objetos de la aplicación con facilidades de hipermedia.

- Architectural Design in an Object-Oriented Framework for Hypermedia
  Alejandra Garrido, Sergio Carvalho y Gustavo Rossi.
  Workshop on Framework Centered Software Development, realizado en conjunción con OOPSLA'95. Austin, Texas, Octubre 15-19, 1995.
  Un comentario del artículo puede encontrarse en: *Addendum to the Proceedings of OOPSLA'95. OOPS Messenger 6* (4). Oct. 1995. ACM Press.

  En este artículo corto discutimos la dificultad de construir frameworks OO y cómo los patterns de diseño pueden ayudar en este proceso, mostrando la utilización que se les dió en el framework para agregar HF a una aplicación OO.

- Design Patterns in an Object-Oriented Framework for Hypermedia
  Sergio Carvalho, Gustavo Rossi y Alejandra Garrido.
  Proceedings of The XV International Conference of the Chilean Society in Information Science. Arica, Chile. Noviembre, 1995.

  Aquí se discute el problema de construir una capa de software para extender aplicaciones OO con una interface hipermedial y estilo de navegación, en forma integral. Presentamos los componentes fundamentales del framework que provee la funcionalidad deseada y analizamos los patterns de diseño que generan la arquitectura del mismo.

- On the Automatic and Hand-made Link Creation at the Information System Level Functionality
  Alejandra Garrido y Harri Oinas-Kukkonen.
  Proceedings of The II International Workshop on Incorporating Hypertext Functionality into Software Systems, realizado en conjunción con Hypertext'96: 7th. ACM Conference on Hypertext. Washington D.C., U.S.A., 16 y 17 de marzo de 1996.

  Este artículo corto presenta las ventajas y desventajas que proveen una y otra aproximación para la definición de links: automática y manualmente.

- Design Issues while building computational hypermedia applications
  Gustavo Rossi, Daniel Schwabe y Alejandra Garrido.
  Proceedings of The II International Workshop on Incorporating Hypertext Functionality into Software Systems, en conjunción con Hypertext'96: 7th. ACM Conference on Hypertext. Washington D.C., U.S.A., 16 y 17 de marzo de 1996.

  Aquí presentamos la interconexión entre el framework de hipermedia y la metodología de diseño de hipermedia OOHDM.

- Design Patterns for Object-Oriented Hypermedia Applications
  Gustavo Rossi, Alejandra Garrido y Sergio Carvalho.
  Pattern Languages of Programs II, Capítulo 11. Editado por J. Vlissides, J. Coplien y N. Kerth. Addison- Wesley. 1996.
  Presentado en PLoP'95. Monticello, Illinois, U.S.A. Septiembre, 1995.

  En este capítulo del segundo volumen del libro de patterns se presentan dos patterns de diseño llamados 'Navigation Strategy' y 'Navigation Observer', mostrando cómo pueden ser usados para diseñar estructuras de navegación flexibles y extensibles.

- Towards a Pattern Language for Hypermedia Applications
  Gustavo Rossi, Daniel Schwabe y Alejandra Garrido.
  PLoP'96: 3rd. Conference on Pattern Languages of Programs.
  Monticello, Illinois, U.S.A. September, 1996.

  Aquí presentamos dos patterns sobre diseño de aplicaciones hipermedia, a nivel de navegación y a nivel de interface.

- Experiencias en el uso de patterns como herramienta de ayuda en el desarrollo de un framework de hipermedia
  Alejandra Garrido y Gustavo Rossi.
  Anales de las Primeras Jornadas de Ingeniería De Software.
  Departamento de Lenguajes de Computación y Sistemas, Universidad de Sevilla, España. Sevilla, Noviembre 14-15, 1996.

  Este artículo presenta las distintas dimensiones en el desarrollo de un framework en las cuales se propone la utilización de patterns como herramienta de gran ayuda.

- A Framework for Extending Object-Oriented Applications with Hypermedia Functionality
  Alejandra Garrido y Gustavo Rossi.
  The New Review of Hypermedia and Multimedia Journal, Volumen 2, 1996.
  Taylor Graham Publishing.

  Este artículo incluye la idea completa y la arquitectura del framework OO que provee funcionalidad de hipermedia, ventajas, componentes y jerarquías principales, se desarrolla un ejemplo y se relaciona el trabajo con otros del área.

- Design Reuse in Hypermedia Application Development
  Gustavo Rossi, Daniel Schwabe y Alejandra Garrido.
  Proceedings of Hypertext'97. Southampton, Inglaterra, Abril 1997.

  En este artículo presentamos un lenguaje de patterns para el dominio de hipermedia. Se proponen distintas dimensiones que cubre aquel lenguaje y se ejemplifica con los patterns hallados.

- Using Object-Oriented Models and Patterns in the WWW
  Gustavo Rossi, Alejandra Garrido y Daniel Schwabe.
  En proceso de referato para el Workshop on Software Engineering On the WWW, Boston, Mayo 1997.

  En este artículo corto presentamos una aproximación para construir aplicaciones hipermedia de gran porte, en particular aplicaciones para el web. Nuestra técnica combina el uso de un método de ingeniería de software orientada a objetos con un sistema de patterns de diseño de navegación e interface.

- Adding Hypermedia Functionality to Object Oriented Applications
  Francisco Vives, Pablo Zanetti y Alejandra Garrido
  En proceso de referato para: Fourth International Conference on Hypertexts and Hypermedia: Products, Tools, Methods. Septiembre 25-26, 1997. Organizado por: Universite Paris VIII, Laboratoire Paragraphe Hypertextes et Hypermedias, Paris, France.

  En este artículo se desarrolla un ejemplo de utilización de la metodología OOHDM para la modelización, y del framework de hipermedia para la implementación de una aplicación hipermedial. Además se propone la incorporación del concepto de "contextos de navegación" propuesto por OOHDM a la arquitectura del framework.

- Hand-made and Computed Links, Precomputed and Dynamic Links
  Helen Ashman, Alejandra Garrido y Harri Oinas-Kukkonen.
  En proceso de referato para : Hypertext - Information Retrieval - Multimedia 1997 (HIM '97), ha realizarse en Dortmund, Septiembre, 1997.

  Este artículo provee una extensa categorización de links manuales y computados, y entre estos últimos, pre-computados y dinámicos. Se describen las ventajas y desventajas de una y otra aproximación para la definición de links, con ejemplos de utilización conveniente.

5.2. Experiencias en el uso del framework

Durante la construcción del framework, y a medida que se iba iterando sobre su diseño, desarrollando las jerarquías de clases y creando clases abstractas, encontrando patterns que modelaran la interacción de los componentes y la fundamentaran, se fueron fabricando distintos ejemplos. Los dos que llegaron a ser mejor completados fueron un sistema de información académico y el prototipo de una herramienta CASE. Explicamos abajo brevemente cada una.

5.2.1. Sistema de información académico

El modelo del sistema de información académico fue tomado de las publicaciones de la metodología OOHDM, que lo presentan y lo desarrollan para explicar los pasos que la metodología propone (por ejemplo [Schwabe+95]). Se utilizó este ejemplo principalmente para chequear la viabilidad del framework para la implementación de los

conceptos propuestos por la metodología. El resultado fue exitoso, excepto por el concepto de "contextos de navegación" aún no provisto por el framework.

5.2.2. *Aplicación en herramientas CASE*

El problema de 'overhead' mental que aparece cuando tenemos que tratar con grandes cantidades de información semi-estructurada, como generalmente ocurre con el proceso de desarrollo de una aplicación, es ampliamente conocido. Este tipo de procesos están caracterizados por gran cantidad de componentes, donde cada uno existe por ciertas decisiones de diseño pobremente documentadas, está relacionado con otros componentes por diferentes tipos de relaciones, y tal que un cambio en su definición puede afectar muchas partes del sistema.

Para manejar un gran conjunto de información semi-estructurada e interconectada, la solución muchas veces propuesta es el uso de hipermedia. Algunas herramientas CASE proveen hipermedia para soportar las conexiones entre diferentes documentos en forma manual [Kelly+96]. Esto por supuesto no es suficiente. Lo que se necesita es un ambiente en el cual existan tipos de nodos y tipos de links predefinidos, de manera que los nuevos componentes que se agreguen tengan automáticamente su nodo asignado, que además permita la creación de links manuales y principalmente que no oculte el comportamiento o la funcionalidad de la herramienta CASE. Por el contrario, lo que resultaría más útil es la completa integración de la funcionalidad del CASE con la navegación hipermedial.

Un prototipo de una herramienta de este estilo es la que se presentó en el capitulo anterior. El hecho de la continua modificación que se presenta en un ambiente de desarrollo hace que una herramienta orientada a objetos, con el dinamismo que esto involucra, sea la más adecuada para su implementación.

Una característica importante que sería bueno agregar es el chequeo de consistencia sobre los links que se creen o generen entre componentes del proceso o artefactos de un modelo. La mayor parte de este chequeo debe ser provista por la misma funcionalidad del CASE, aunque puede existir la necesidad de crear tipos de nodos y links especializados. Esto es parte del trabajo futuro en la integración del framework con una herramienta CASE potente.

6. Conclusiones: Fundamentación de la utilidad del framework y posibles extensiones

En este capítulo se presentan las conclusiones del trabajo hasta aquí expuesto, la utilidad lograda hasta el momento y la que esperamos alcanzar, incluyendo el trabajo futuro.

Se incluyen también en el capítulo las diferencias y similitudes con otros trabajos relacionados. Esperamos que este trabajo motive otras extensiones sobre él que por el momento escapan a nuestras manos.

6.1. Conclusiones

En este trabajo hemos discutido la construcción de un framework orientado a objetos, el OO-Navigator, cuyo propósito es realzar aplicaciones OO con funcionalidad de hipermedia. Este área es particularmente crítica dada la aparición de una nueva generación de aplicaciones distribuidas que combinan características de aplicaciones hipermedia con aquellas basadas en transacciones convencionales. Podemos observar algunas de estas aplicaciones en el WWW. El principal objetivo del OO-Navigator es el de lograr la *integración del comportamiento de una aplicación con la navegación hipermedial sobre la misma.*

La construcción de este tipo de aplicaciones híbridas involucra tratar con diferentes dimensiones de diseño, como especificación del comportamiento de los objetos, componentes de hipermedia (como nodos, links o índices) y sus interfaces. Estas dimensiones son abordadas en el OO-Navigator en una forma modular y flexible, utilizando conceptos avanzados de la tecnología de objetos como frameworks y patterns.

Los frameworks OO constituyen el estado del arte para el reuso de abstracciones de diseño a gran escala, en un dominio de aplicación particular. Ya han sido usados por años en áreas que han alcanzado la maduración suficiente. El uso de frameworks permite separar un sistema en distintos niveles de abstracción, permitiendo así fácil *extensión, reuso y mantenimiento.*

Diseñar el framework no fue una tarea fácil. Muchas iteraciones fueron necesarias para obtener jerarquías sólidas. La documentación de las decisiones que provocaban cambios en el diseño era bastante pobre. Fue con el arribo de los patterns de diseño que la documentación del framework comenzó a tomar sentido para quienes se les intentaba transmitir el diseño.

El OO-Navigator fue implementado usando ParcPlace-Digitalk VisualWorks™, y ha sido usado para extender algunas aplicaciones OO existentes durante el proceso de construcción del framework (el prototipo de una herramienta CASE para OOSE, un sistema académico, el sistema de información de una biblioteca), para evaluar la adaptabilidad del mismo (ver Sección 5.2). También hemos construido un toolkit para ayudar al diseñador en el proceso de instanciación (Sección 3.5). Hemos extendido el framework de interface de VisualWorks, por ejemplo con widgets adaptados a hipermedia, agregando la posibilidad de navegar en la misma ventana en vez de abrir siempre una nueva, y cambiamos el proceso de construcción de ventanas (Sección 2.3).

En este lugar sería útil retomar las contribuciones que se listaron al comienzo de este trabajo, y desarrollar cada una fundamentando su validez.

6.2. Fundamento de las contribuciones enunciadas

Las contribuciones enunciadas al comienzo de este trabajo fueron:

- *Definición de un modelo unificado de conceptos de funcionalidad de hipermedia.*

  En el Capítulo 2, Sección 2.1, hemos presentado el modelo esencial desarrollado para luego, en base al mismo, diseñar los componentes del framework. La construcción de

este modelo unificado ha permitido contar con una abstracción real de las aplicaciones en el dominio de hipermedia sobre la que se basa el OO-Navigator.

La diferencia principal entre este trabajo y otras visiones orientadas a objetos de la construcción de aplicaciones hipermedia, como la propuesta por Lange [Lange94] o Mamann [Marmann+92], surge del hecho de que el framework de hipermedia constituye sólo un nivel de una aplicación más compleja que supone un modelo de colaboración con los niveles de objetos y de interface. De esta forma el OO-Navigator permite incorporar características de hipermedia en aplicaciones de negocios convencionales.

En [Grønbaek+94a] y [Arents+91] se proponen arquitecturas en capas similares a la presentada aquí, aunque no se aplican a la funcionalidad de hipermedia.

El trabajo más cercano es el desarrollado por Michael Bieber [Bieber+95, Bieber+97], que también maneja el concepto de agregar funcionalidad de hipermedia a aplicaciones computacionales, separando los cómputos del módulo de hipermedia. La diferencia es que no se ocupa de aplicaciones OO, sino mayormente de aquellas implementadas en bases de datos convencionales, y por lo tanto carece de un modelo abstracto basado en jerarquías de clases. Por el contrario se basa en reglas escritas en lenguaje funcional (por ejemplo Prolog) para la definición de tipos de nodos y links a partir de una base de datos de la aplicación.

- *Propuesta de la utilización de funcionalidad de hipermedia en aplicaciones OO.*

Mediante el uso del framework de hipermedia, una aplicación OO se beneficia con las posibilidades de:

- navegar entre ítems de información relacionados, aunque esas relaciones no estuvieran capturadas por la aplicación; muchas veces las relaciones tienen significado para un lector, pues la racionalidad de la relación no puede ser expresada mediante un predicado general;

- introducir mayor semántica a las relaciones entre componentes, con el agregado de atributos y comportamiento a las mimas;

- definir diferentes contextos para diferentes perfiles de usuario, brindando a cada lector la información que resulta apropiada a sus necesidades;

- resaltar la interface gráfica con la posibilidad de definir anclas de links sobre cualquier tipo de dato;

- permitir acceso rápido y fácil a la información deseada por medio de índices y backtracking;

- guiar en la transferencia de información que se adecue al perfil del lector con tours guiados;

- agregar anotaciones;

- contar con la historia de nodos y/o links navegados.

- *Aplicación de los principios de la tecnología de objetos al campo de hipermedia como base para su diseño, logrando los beneficios conocidos del paradigma OO.*

  El diseño OO ha demostrado ser la mejor forma de alcanzar reuso, pues permite definir por abstracción y composición los componentes de una aplicación, que luego son conectados para alcanzar el comportamiento esperado del sistema. Los datos de la aplicación son distribuidos entre los componentes, que se hacen responsables de mantenerlos encapsuladamente, y de llevar a cabo el comportamiento esperado. Esta organización de la información se asemeja a la organización no-lineal de aplicaciones hipermedia, en las que porciones discretas de datos son relacionadas con semántica de grafos [Garrido+96a].

  A raíz de lo expuesto en el párrafo anterior, una aplicación hipermedia puede obtener del paradigma OO lo siguiente:

  - reuso de conceptos de hipermedia;

  - una definición de links en base a relaciones basadas en un modelo semánticamente rico;

  - dinamismo mediante el agregado de computaciones, tanto las relacionadas con la información que se presenta en el nodo como la de generación automática de links;

  - la posibilidad de definir clases de nodos y links, y automáticamente crear todas sus instancias;

  - mantenibilidad en cada capa de abstracción de la arquitectura.

- *Construcción de una herramienta que permite extender cualquier aplicación OO con aquella funcionalidad.*

  Como este ítem expresa, se ha diseñado y construido un framework OO y un toolkit que permite su instanciación para implementar las ideas aquí presentadas.

- *Descubrimiento de patterns de diseño usados recurrentemente en aplicaciones hipermediales y en el diseño de la arquitectura construida.*

  En las secciones 2.2.3 y 2.2.4 se han descripto el uso de varios patterns en el desarrollo del framework. Los patterns transmiten las principales decisiones de diseño tomadas en el proceso y sus implicancias, justificando la arquitectura lograda. Se utiliza el mismo formato de patterns para explicar el uso de los mismos, por considerar que esto aumenta su expresividad.

6.3. Extensiones

La esencia de un framework es el modelo de interacción y flujo de control entre sus componentes [Johnson+91]. Un framework caja negra ideal es aquel que permite ser instanciado utilizando únicamente las clases ya definidas en él mismo. En la práctica, puede ser necesario agregar especializaciones particulares, y en ese caso crear nuevas subclases en las jerarquías existentes, adaptando así la aplicación. Esta es una de las razones por la cual el diseño de un framework es generalmente difícil: debe ser adaptable y extensible.

De esta manera, instanciar el framework de hipermedia implica decidir si las clases concretas provistas en él son adecuadas o deben ser especializadas. Ejemplos de necesidad de subclasificación serían: una nueva estructura de acceso, nodos con checkeo de algún tipo de restricción, links que mantengan su fecha de creación y/o última modificación, nodos-colección con una relación especial entre componentes, etc.

Otro tipo de extensiones son las que tienen que ver con agregar capacidades al framework que por falta de tiempo y por necesitar de restringir su alcance no han podido ser desarrolladas hasta el momento. Estamos hablando de ***trabajo futuro***. El mismo incluye, en otros puntos:

- soporte completo de multimedia y capacidad de realizar queries;

- permitir herencia entre clases de nodos y clases de links, reflejando las jerarquías del modelo subyacente;

- portar el framework a PPD VisualWave™, el cual permitiría que las aplicaciones OO puedan ser vistas desde un browser de WWW.

7. Bibliografía

En este capítulo se cita la bibliografía en orden alfabético por el primer autor. El signo '+' en las referencias denota que hay más de un autor involucrado en el artículo.

[Agosti+95] M. Agosti y J. Allan *editores*, "IR and Automatic Construction of Hypermedia. A Research Workshop". *Workshop in SIGIR'95*. Seattle, Julio 13, 1995.

[Alexander+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King y S. Angel. *"A Pattern Language"*. New York: Oxford University Press, 1977.

[Alexander79] C. Alexander. *"The Timeless Way of Building"*. New York: Oxford University Press, 1979.

[Alvarez+95] X. Alvarez, G. Dombiak, A. Garrido, M. Prieto y G. Rossi. "Objects on Stage. Animating and Visualizing Object-Oriented Architectures in CASE Environments" Proceedings of the International Workshop on Next Generation CASE Tools. CAiSE'95, Finlandia, Junio 1995.

[Anderson+94] K. Anderson, R. Taylor y E. Whitehead Jr. "Chimera: Hypertext for Heterogeneous Software Environments". *Proceedings of the ACM European Conference on Hypermedia Technology*. Edimburgo, Escocia, 1994, pp. 94-107.

[Arents+91] H. Arents y W. Bogaerts. "Towards an architecture for third-order hypermedia systems. *Hypermedia, 3* (2), 1994.

[Ashman+96] H. Ashman, T. Cawley, S. Davis y G. Chase. "Issues in the Use of External and Remote Services in Hypermedia Systems". *Proceedings of the Second International Workshop on Incorporating Hypertext Functionality Into Software Systems*. Washington D.C., U.S.A., 16 y 17 de marzo de 1996.

[Ashman+97] H. Ashman, A. Garrido y H. Oinas-Kukkonen. "Hand-made and Computed Links, Precomputed and Dynamic Links". *En proceso de referato para : Hypertext - Information Retrieval - Multimedia 1997 (HIM '97)*, ha realizarse en Dortmund, Septiembre, 1997.

[Beck+89] K. Beck y W. Cunningham. "A Laboratory For Teaching Object-Oriented Thinking". *Proceedings of OOPSLA'89*, New Orleans, Louisiana, 1989.

[Beck+94] K. Beck y R. Johnson. "Patterns Generate Architectures". *Proceedings of the 8th European Conference, ECOOP'94*. Lecture Notes in Computer Science, Vol. 821, 1994.

[Bieber+92] M. Bieber y S. Kimbrough. "On Generalizing the Concept of Hypertext". *MIS Quaterly*, Marzo 1992, pp. 77-93.

[Bieber93] M. Bieber. "Providing Information Systems with Full Hypermedia Functionality". *Proceedings of the 26th. Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA, 1993, Vol. III, pp. 390-400.

[Bieber+95] M. Bieber y C. Kacmar. "Designing Hypertext Support for Computational Applications". *Communications of the ACM, Vol. 38, No. 8*. August 1995, pp. 99-107.

[Bieber+97] M. Bieber y F. Vitali. "Toward Support for Hypermedia on the World Wide Web". *IEEE Computer*, Enero 1997.

[Booch92] G. Booch. "Designing an Application Framework". *Dr. Dobb's Journal*, 19 (2), pp. 24, 1994.

[Buschmann+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal. *"Pattern-Oriented Software Architecture. A system of patterns"*. John Wiley & Sons, 1996.

[Campbell+91] R. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany. "Choices, Frameworks and Refinement". In Luis-Felipe Cabrera and Vincent Russo, and Marc Shapiro, editor, *Object-Orientation in Operating Systems*, pages 9-15, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[Casanova+91] M. Casanova y L. Tucherman. "The Nested Context Model for Hyperdocuments". *Proceedings of Hypertext'91*. Diciembre, 1991.

[Conklin+88] J. Conklin y M.L. Begeman. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion". *ACM Transactions on Office Information Systems, Vol. 6* (4), 1988.

[Coplien95] J. Coplien. "A Generative Development-Process Pattern Language". *Pattern Languages of Program Design. Vol. I*, Addison Wesley. 1995.

[Davis+94] H. Davis, S. Knight y W. Hall. "Light Hypermedia Link Services: A Study of Third-Party Application Integration". *Proceedings of the ACM European Conference on Hypermedia Technology*. Edimburgo, Escocia, 1994, pp. 41-50.

[Gamma+95] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley. 1994.

[Garrido+96a] A. Garrido y G. Rossi. "A Framework for Extending Object-Oriented Applications with Hypermedia Functionality". *New Review of Hypermedia and Multimedia Journal*, Taylor Graham Publishing, Vol. 2, 1996, pp. 25-41.

[Garrido+96b] A. Garrido y H. Oinas-Kukkonen. "On the Automatic and Hand-made Link Creation at the Infomation System Level Functionality". *Proceedings of The II International Workshop on Incorporating Hypertext Functionality into Software Systems*, Hypertext'96. Washington D.C., U.S.A. Marzo, 1996.

[Garrido+97a] A. Garrido y G. Rossi. "Capturing hypermedia functionality in an object-oriented framework". *Submitted to ACM Communications*, 1997.

[Garrido+97b] A, Garrido y G. Rossi. "Using Patterns to Define a Framework's Architecture". Presentado en *The First Conference on Using Patterns*. Marzo 7-9, 1997.

[Garzotto+91] F. Garzotto, P. Paolini y D. Schwabe. "HDM - A Model for the Design of Hypertext Applications". *Proceedings of Hypertext'91*. Diciembre 1991, pp. 313-328.

[Garzotto+94] F. Garzotto, L. Mainetti y P. Paolini. "Adding Multimedia Collections to the Dexter Model". *Proceedings of the European Conference on Hypermedia Technology*, ECHT'94, Edimburgo, Escocia, Septiembre 1994, pp. 70-80.

[Goldberg+83] A. Goldberg y D. Robson. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Grønbaek+94a] K. Grønbaek y R. Trigg. "Design Issues for a Dexter-Based Hypermedia System". *Communications of the ACM 37 (2)*, Febrero 1994.

[Grønbaek94b] K. Grønbaek. "Composites in a Dexter-Based Hypermedia Framework". *Proceedings of the European Conference on Hypermedia Technology*, ECHT'94, Edimburgo, Escocia, Septiembre 1994, pp. 59-69.

[Haake+92] J. Haake y B. Wilson. "Supporting Collaborative Writing of Hyperdocuments in SEPIA". *Proccedings of CSCW'92*, Noviembre, 1992.

[Haake+94] J. Haake y N. Streitz. "Coexistence and Transformation of Informal and Formal Structures: Requirements for More Flexible Hypermedia Systems". *Proceedings of the European Conference on Hypermedia Technology,* ECHT'94, Edimburgo, Escocia, Septiembre 1994, pp. 1-12.

[Halasz+90] F. Halasz y M. Schwartz. The Dexter Hypertext Reference Model. In: J. Moline, D Benigui, and J. Garonas eds., *Proceedings of the Hypertext Standardization Workshop*, pp 145-166. Gaithersburg, Maryland, Enero 16-18, 1990. National Institute of Standards and Technology. U.S. Department of Commerce, National Technical Information Service.

[Halasz+94] F. Halasz y M. Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM, 37* (2), 1994, pp. 30-39.

[Helm+90] R. Helm, I. Holland y D. Gangopadhyay. "Contracts: Specifying behavioral compositions in object-oriented systems". *Proceedings of OOPSLA'90*, Ottawa, Canadá, Octubre 1990.

[Hill+96] G. Hill, W. Hall, D. DeRoure y L. Carr. "Appliying Open Hypertext Principles to the WWW". *Proceedings of the International Workshop on Hypermedia Design*, Montpellier, Francia, Junio 1995. Belin: Springer-Verlag, 1995.

[Howard95] T. Howard. *"The Smalltalk Developer's Guide to VisualWorks"*. SIGS Books, 1995.

[HTFI] New Jersey Institute of Technology: *The First Workshop on Incorporating Hypertext Functionality into Software Systems* (Edinburgh, Scotland, September 18, 1994). New Jersey Institute of Technology: Technical Report #95-10.

[Isakowitz93] T. Isakowitz. "Hypermedia, Information Systems and Organizations: An agenda for research". *Proceedings of the 26th Hawaii International Conference on System Science*, pp 380-389. Maui, Hawaii, Enero 5-8, 1993.

[Isakowitz+95] T. Isakowitz, E. Stohr y P. Balasubramanian. "RMM: A Methodology for Structured Hypermedia Design". *Communications of the ACM 38* (8), 1995, pp. 34-44.

[Jacobson+92] I. Jacobson, M. Christerson, P. Jonsson, y G. Overgaard. *"Object-Oriented Software Engineering. A Use Case Driven Approach"*. Addison-Wesley, 1992.

[Johnson+88] R. Johnson y B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming, 1* (2) 1988, 22-35.

[Johnson+91] R. Johnson y V, Russo. "Reusing Object-Oriented Design". *University of Illinois Tech Rep UJUCDCS 91-1696,* Champaign-Urbana, 1991.

[Johnson92] R. Johnson. "Documenting Frameworks with Patterns". *Proceedings of OOPSLA'92,* SIGPLAN Notices, 27 (10), pp. 63-76. Vancouver BC, Octubre, 1992.

[Johnson+97] R. Johnson y B. Woolf. "The Type-Object Pattern". *A aparecer en Pattern Languages of Program Design. Vol. 3.* Addison-Wesley, 1997.

[Kelly+96] S. Kelly, K. Lyytinen y M. Rossi. "MetaEdit+: A Fully Configurable Multiuser and Multitool CASE Environment". *Proceedings of the 8th. Conference on Advance Information Systems Engineering (CAiSE'96)*, Creta, Grecia, Mayo 1996.

[Krasner+88] G. Krasner y S. Pope. A cook-book for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming, 1* (3), 1988, 26-49.

[Lange90a] D. Lange. "A formal Model of Hypertext". J. Moline, D Benigui, and J. Garonas eds., *Proceedings of the Hypertext Standardization Workshop*, pp 145-166. Gaithersburg, Maryland, Enero 16-18, 1990. National Institute of Standards and Technology. U.S. Department of Commerce, National Technical Information Service.

[Lange90b] D. Lange. "A Formal Approach to Hypertext using Post-Prototype Formal Specification". *Proceedings of VDM'90,* Kiel, Abril 17-21, 1990, BRD. Lecture Notes in Computer Science, vol. 428.

[Lange94] D. Lange. "An Object-Oriented Design Method for Hypermedia Information Systems". *Proceedings of the 27th. Annual Hawaii International Conference on System Science*, Hawaii, Enero, 1994.

[Marmann+92] M. Marmann y G. Schlageter. "Towards a better support for hypermedia structuring: The HYDESIGN Model". *Proceedings of the ACM European Conference on Hypertext,* ECHT'92. Diciembre, 1992.

[Nelson96] R. Nelson. "Issues In Applicative Hyperization of Unwitting Systems". *Second International Workshop on Incorporating Hypertext Funcitonality into Software Systems,* en conjunción con Hypertext'96. Washington, USA, Marzo, 1996.

[Nielsen93] J. Nielsen. *Hypertext and Hypermedia.* Academic Press, 1990.

[Oinas95] H. Oinas-Kukkonen. "Developing Hypermedia Systems - The Hypertext Functionality Approach". In: Data Management Systems, *Proceedings of the Basque International Workshop on Information Technology*, BIWIT'95 (San Sebastian, Spain, July 1995). Los Alamitos, North Carolina: IEEE Society Press, 1995, 2-8.

[Opdyke+90] W. Opdyke y R. Johnson. "Refactoring: an aid in designing application frameworks and evolving object-oriented systems". *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applicarions (SOOPPA)*, 1990.

[PP94] ParcPlace Systems, *VisualWorks User's Guide*, 1994.

[Pree95] W. Pree. *"Design Patterns for Object-Oriented Software Development,"* Addison-Wesley, 1995.

[Rossi+96a] G. Rossi, A. Garrido y S. Carvalho. "Design Pattern for Object-Oriented Hypermedia Applications. *Pattern Languages of Program Design, Vol. 2*, capítulo 11, pp. 177-191. Vlissides, Coplien y Kerth editors, Addison-Wesley, 1996.

[Rossi+96b] G. Rossi, D. Schwabe y A. Garrido. "Towards a Pattern Language for Hypermedia Applications". *Proceedings of the Third Conference on Pattern Languages of Programs.* Monticello, Illinois, U.S.A. September, 1996. Available at *http://www.cs.wustl.edu/~schmidt/PLoP-96/workshops.html*

[Rossi+97] G. Rossi, D. Schwabe y A. Garrido. "Design Reuse in Hypermedia Application Development". *Proceedings of the Eighth ACM Conference on Hypertext* (Hypertext'97). Southampton, England. Abril 6-11, 1997.

[Rumbaugh+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. *"Object-Oriented Modeling and Design"*. Prentice Hall, 1991.

[Schmidt95] D. Schmidt. "Acceptor and Connector : Design Patterns for Active and Passive Establishment of Network Connections". *ECOOP workshop on pattern languages of object-oriented programs*, Aarthus, Dinamarca, Agosto 1995.

[Schwabe+95] D. Schwabe y G. Rossi. "Building Hypermedia Applications as Navigational Views of Information Models". *Proceedings of the Hawaii International Conference on System Science*, Hawaii, Enero 1995.

[Schwabe+96] Schwabe, D., Rossi, G. y Barbosa, S. "Systematic Hypermedia Design with OOHDM". *Proceedings of the ACM International Conference on Hypertext, Hypertext'96*, (Washington, Marzo 1996).

[Trigg88] R. Trigg. "Guided tours and tabletops: tools for communicating in a hypermedia environment". ACM Transactions on Office Information Systems 6 (4). Octubre 1988, pp. 398-414.

[Woolf94] B. Woolf. "Improving dependency notification". *The Smalltalk Report, Vol. 4* (3), SIGS Publications, Noviembre 1994.

[Woolf95] Bobby Woolf. "Understanding and Using the ValueModel Framework in VisualWorks Smalltalk". *Pattern Languages of Program Design Vol. 1*, Addison Wesley. 1995.

[Zellweger89] P. Zellweger. "Scripted Documents: A Hypermedia Path Mechanism". *Proceedings of ACM Hypertext'89*, Pittsburgh, PN, Nov. 1989.

8. Glosario

Se listan a continuación algunos términos y siglas utilizados en el trabajo.

Canvas: Herramienta provista por VisualWorks™ para la construcción de interfaces.

CASE: Computer Aided Software Engineering (Ingeniería de Software Asistida por Computadora.

FH: Funcionalidad de Hipermedia.

OHS: Open Hypermedia System (Sistema de Hipermedia Abierto).

OO: Orientado/a/s a Objetos.

PD: Pattern de Diseño.

PsD: Patterns de Diseño.

SI: Sistemas de Información.

Widget: objeto de interface gráfica responsable de la representación de un componente.

WWW: World Wide Web.